Bootstrapping Meta-Circular and Autogenous Code Generation

Herwig Mannaert

Normalized Systems Institute
University of Antwerp, Belgium
Email: herwig.mannaert@uantwerp.be

Koen De Cock

Research and Development
NSX bv, Belgium
Email: koen.de.cock@nsx.normalizedsystems.org

***Abstract*—Metaprogramming or automated code generation has been pursued for a long time, and is often considered crucial to increase programming productivity. It has been argued in previous work that evolvability of software is equally important, and that a meta-circular metaprogramming architecture may be crucial to addressing some fundamental evolvability issues in metaprogramming. At the same time, the field of software engineering struggles to provide firm technical guidance to computer programmers, and often reverts to heuristics and documented patterns. As metaprogramming is in general more complex than traditional programming, it seems even more crucial to provide technical guidance to metaprogrammers. In this contribution, the bootstrapping of an elementary meta-circular metaprogramming environment is investigated. Its main purpose is to serve as a pathfinder for the development of design patterns and techniques that can support and guide metaprogramming.**

***Index Terms*—Evolvability; Metaprogramming; Design Patterns; Meta-Circularity.**

I. Introduction

Metaprogramming or automated code generation has been pursued for a long time. Among other things, it is often seen as one of the most promising approaches to increase the productivity in computer programming. We have argued in our previous work that it is equally important to increase the evolvability of software systems [1], although this receives less attention within the Information Systems (IS) research area [2]. We have also argued that some fundamental issues hamper the evolvability of metaprogramming software [3], and that a meta-circular architecture seems suitable to address this. At the same time, it seems hard to provide strict guidance to computer programmers [4], and the field of software engineering often reverts to heuristic design patterns and craftsmanship practices. As metaprogramming is in general more complex than programming, and meta-circular metaprogramming even more complex, it seems imperative to provide some solid design patterns for such metaprogramming environments. In this contribution, we investigate the bootstrapping of a basic meta-circular metaprogramming environment, that could serve as a pathfinder to extract such envisioned design patterns, or even more fundamental techniques.

The remainder of this paper is structured as follows. In Section II, we briefly discuss automatic or metaprogramming, and its relationship with the broader field of software engineering. In Section III, we describe the overall architecture and the implementation setup of our elementary metaprogramming environment. Section IV presents the detailed procedure to bootstrap the meta-circular and autogenous metaprogramming environment. The details and characteristics of this environment are discussed in Section V. Finally, we present some conclusions in Section VI.

II. Metaprogramming and Software Engineering

In this section, we give a brief overview of the field of metaprogramming, and discuss its importance and position in the discipline of software engineering.

A. Automatic or Metaprogramming

The automatic generation of code, i.e., *writing code that writes code*, is probably as old as coding or software programming itself. Not unlike many other areas in information technology, several different terms are used to describe this activity, and their associated meanings may vary both over time and between authors. We briefly go through some terms and concepts, which we have explained in more detail in [3].

Though it has been argued for a long time that the mechanisms are quite similar [5], a distinction is often made between *code generation*, where a compiler generates executable code from a high-level programming language, and *automatic programming*, where source code is generated from a model or template. Related terms include *generative programming*, highlighting the similarity to automated manufacturing in the industrial sector [6], and *metaprogramming*, emphasizing the fact that this is a meta-level activity, and often defined as a programming technique in which computer programs have the ability to treat other programs as their data [7].

One of the main goals of automatic programming has always been to improve programmer productivity. Therefore, several terms in software development methodologies and tools are closely related to automatic programming. Methodologies like *Model-Driven Engineering (MDE)* and *Model-Driven Architecture (MDA)* focus on the creation and exploitation of conceptual domain models and ontologies, and assume the presence of software tools for the automatic generation of code. Such model-driven code generation tools that provide an environment for programmers to create application software through graphical user interfaces and configuration, are now

 ISBN: 978-1-61208-997-3

often referred to as *Low-Code Development Platforms (LCDP)* or *No-Code Development Platforms (NCDP)*. Despite all the terms and tools, the realization of automatic programming on an industrial scale remains not straightforward [8].

B. Engineering Metaprogramming

Software engineering is sometimes considered to be an eclectic field. To guide software developers, various paradigms, techniques, process models and tools exist. Even a vast amount of *software development methodologies*, i.e., a comprehensive guide to developing a system [9], have been available for decades. Though this resulted in a widespread belief that adherence to systems development methodologies is beneficial [10], the adoption of systems development methods remained limited [9] [10]. This led to more pragmatic strategies for concrete guidance, like the use of *design patterns* [11], solutions to common development problems that have proven their quality empirically, *agile methodologies* like SCRUM [12] that value self-organizing teams and adaptive collaboration with customers over strict planning and comprehensive processes, and *software craftsmanship* practices like clean code [13]. It seems that we have not found the fundamental laws of software that would play the role that the fundamental laws of physics play for other engineering disciplines [4], which could explain the renewed emphasis on best practices and heuristics in postmodern software engineering.

Though metaprogramming would seem in general to be more complicated than standard programming, structured techniques and guidance seem even scarcer for metaprogramming. Nevertheless, we have argued in our previous work that some fundamental issues need to be addressed to achieve productive and scalable adoption of automatic programming techniques [3]. First, to cope with the increasing complexity due to changes, we have proposed to combine automatic programming with the evolvability approach of *Normalized Systems Theory (NST)* providing (re)generation of the recurring structure and re-injection of the custom code [1], [3]. Second, we have proposed a meta-circular architecture to regenerate the metaprogramming code itself as well [14], [15]. The term meta-circularity dates back to Reynolds [16], and is related to the concept of homoiconic languages [17], enabling a program to be manipulated as data using the same language, and allowing the program's internal representation to be inferred just by reading the program itself.

Such a meta-circular architecture offers several potential benefits. It could avoid the growing burden of maintaining the often complex meta-code and continuously adapting it to new technologies [3], and could facilitate a more scalable collaboration between metaprogrammers through the exchange of meta-models. At least, a unified view on both the metaprogramming code and the source code being generated would reduce the cognitive load for the metaprogrammers, and would enable us to apply advancements in programming techniques simultaneously to both the generative and generated code.

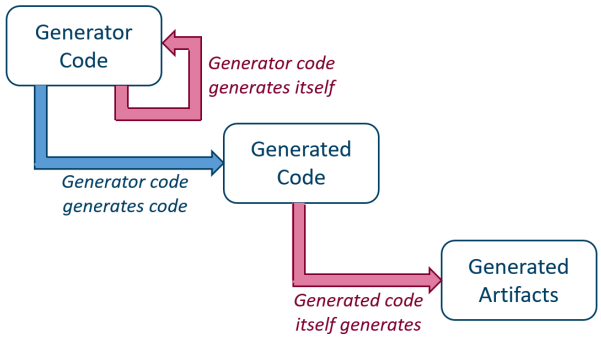

Figure 1. Representation of meta-circular and autogenous generation.

As one could expect meta-circular metaprogramming to be even more complicated than standard metaprogramming, it seems imperative to provide guidance to the metaprogrammers through structured concepts, techniques, and patterns. In this contribution, we investigate the bootstrapping of an elementary meta-circular metaprogramming environment in a very basic Java programming environment. The main purpose of such an elementary environment is to clarify the basic concepts, and to serve as a pathfinder to extract some future design patterns — and maybe even more fundamental techniques or basic primitives — to guide metaprogrammers.

III. An Elementary Metaprogramming Environment

In this section, we explain the scope, purpose, architecture, and implementation environment of the elementary metaprogramming environment presented in this paper.

A. Purpose and Overall Architecture

The scope of the code generation needs to be very basic but nevertheless realistic. As data models such as *Entity Relationship Diagrams (ERD)* are common and widespread, we decided to use basic representations of data entities as the models of the metaprogramming environment. The generated code needs to be able to represent such basic data entities, featuring both data attributes and relationships or references, and to import or read instances of these data entities. An example of such a data entity would be an invoice, having an invoice number and client as attributes, and references to the various invoice lines as relationships.

The goal is to create an elementary metaprogramming environment with a clear and simple structure, completely devoid of unnecessary complexities. At the same time, we want the metaprogramming environment to exhibit two additional fundamental characteristics. These fundamental properties are schematically represented in Figure 1.

- Meta-Circular generation:
  The generator code needs to be able to generate itself. This requires, of course, a bootstrapping process, i.e., the generator code needs to be handcrafted first before it can be enabled to (re)generate itself.

 ISBN: 978-1-61208-997-3

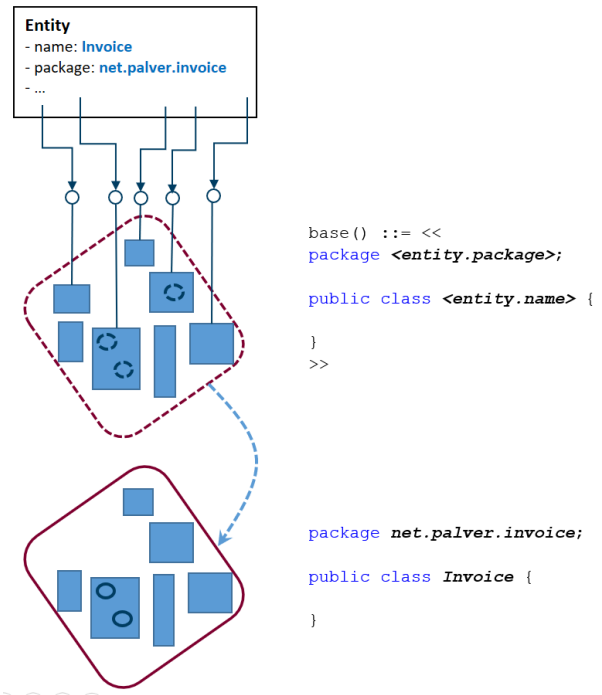

Figure 2. Instantiating a coding template with model parameters.

- Autogenous generation:
  The generated code itself needs to be able to generate. The artifacts that the generated code could generate should in general be related to the domain entity, e.g., invoice classes could generate an invoice document.

B. A Basic Implementation Environment

To represent data, both the model parameters such as entities and attributes, and the actual instances of the data entities such as invoice numbers, we decided to use the extremely simple and widespread format of *Comma-Separated Values (CSV)*. As templating engine, we have opted for *StringTemplate* [18], as it allows hardly any logic in the templates [19]. In this way, we ensure that the templating engine serves as a plain off-the-shelf tool to replace parameter strings with actual values.

To feed the data of the model into the templating engine, we use an elementary kernel function of the metaprogramming environment described in [3]. Based on this functionality, the model consisting of linked data instances is made available to the coding templates through *Object Graph Navigation Language (OGNL)* expressions [20] using the *Apache Commons OGNL* library [21]. More specifically, every model data *entity* is passed as a linked data tree to the template, where we can select individual properties like name by `entity.name`, or access a linked list of attributes by `entity.attributes` and loop through them. The templating engine simply replaces the parameters in the template with the actual values from the linked data tree. This procedure, a kind of *convolution* between a template and a linked data tree, seems to us one of the most basic mechanisms for code generation.

Figure 2 provides a schematic representation of such an elementary code generation. An instance of a model *entity*, with name *Invoice* and belonging to a package *net.palver.invoice*, is fed into a coding template for a base class. In this template, the values of the model entities are represented as parameters. The generator code will resolve these parameters and replace them with the actual values of the model entity, resulting in real source code for that domain entity.

IV. Toward Meta-Circular and Autogenous Expansion in Ten Steps

In this section, we present a detailed procedure to bootstrap in ten steps an elementary meta-circular and autogenous metaprogramming environment. The ten steps are grouped in four cohesive subprocedures.

A. Create Basic Generator Code

1) Read basic entity model: We create a base *DTO (Data Transfer Object)* class *EntityComposite* to represent the individual entities with basic attributes like name and package name, and an *EntityReader* that reads a CSV file *Entitys.csv*, and makes a list of DTO instances based on the entries. As a test, we print the list of instances after reading.

2) Expand basic entity class: We introduce a base class expander template *CompositeExpander* for a DTO class representing a domain entity, and a generator class *EntityGenerator* that reads the entities using the *EntityReader*, and feeds them to the template to create the DTOs. We check whether the base classes, similar to Figure 2, are correctly generated.

B. Generate Viable Data Entities

1) Support entity attributes: We provide support for attributes having a type and name through an *AttributeComposite* and *AttributeReader* class, include a variable list of attribute composites in the *EntityComposite* DTO, and introduce get-set-methods in the *CompositeExpander* template. The *EntityReader* is extended to read for every entity the list of attributes from a *<Entity.name>Attributes.csv* file. As a test, we define number and client as invoice attributes, and read some sample data.

2) Generate an entity reader: We introduce a *ReaderExpander* template based on the reader class that we have programmed, and extend the *EntityGenerator* to instantiate for every entity a reader class together with a composite class. The generated classes are compiled for an invoice entity, and tested by reading some sample data.

3) Introduce linked entities: We introduce *list type attributes* and extend the *CompositeExpander* template to support the variable definition and get-set-methods for such list fields. We also extend the *ReaderExpander* template to read these linked entities, in the same way that the *EntityReader* supports invoking the *AttributeReader*. We test this by defining an *InvoiceLine* entity, an *invoiceLines* attribute in the invoice entity, and reading some sample invoices and invoice lines.

 ISBN: 978-1-61208-997-3

C. Regenerate the Generator Code

1) Generate the meta-entities: The model to represent data models is implicitly based on data entities itself, being *Entity* and *Attribute*. So, we define them in the CSV files *Entitys*, *EntityAttributes*, and *AttributeAttributes*, and run the *EntityGenerator* to regenerate the composite and reader classes for entity and attribute. After comparing the generated classes with the origonal ones, we make them available.

2) Replace original base code: We now retire the original composite and reader classes, and replace them in our test setup. Using the newly generated composite and reader classes, and the still original *EntityGenerator* class, we perform regression testing by reading the models for invoice and invoice line, and generating their composite and reader classes.

3) Replace the generator code: We are now ready to generate the generator class itself through a *GeneratorExpander* template. To avoid hardcoding the specific expander templates to be used, we introduce an additional model entity *ExpanderPath* and define the actual expander templates to be used for the entity *Entity* in a CSV file *EntityExpanderPaths*. The generated *EntityGenerator* class will use this CSV file to guide the instantiation fo templates. We also introduce an *expandable* attribute on the entity, as this generator class does not need to be generated for the traditional domain entities. Figure 3 represents the code template for the generator class and its instantiation for the *Invoice* entity, similar to the base class instantiation represented in Figure 2.

D. Enable Autogenous Generation

1) Create autogenous generator: All Java source code has now been retired. A traditional domain entity like invoice can now be enabled to perform generation itself, i.e., autogenous generation, by simply setting the *expandable* attribute, resulting in the generation of an *InvoiceGenerator* class. This generator class will use the list of templates through a CSV file *InvoiceExpanderPaths*, and instantiate these templates. The artifacts to be created by these domain entities are not necessarily programming files. As a test example, we used an *InvoiceTexExpander* template, generating a Latex file for every invoice instance.

2) Generate derived artifacts: We are now able to generate additional artifacts for instances of domain entities like *Invoice* by simply defining additional templates and defining them in the *InvoiceExpanderPaths* CSV file. For instance, we have introduced *HyperText Markup Language (HTML)* and *Universal Business Language (UBL)* expansion templates to instantiate and exchange invoices.

V. Results and Discussion

Figure 4 presents a schematic overview of the various artifacts and their interrelationships. These artifacts include model parameter data files, domain entity data files, (generated) source code classes, runtime object instances, source code templates, and artifacts generated by the domain classes.

TABLE I.
SIZE OF THE REMAINING TEMPLATE SOURCE CODE.

Expander	LOC	#Bytes
Composite	29	735
Reader	35	1180
ExpansionContext	26	726
Generator	34	1230
Total	**124**	**3871**

The green area filling symbolizes the fact that the artifacts are generated, and the light blue contours indicate run-time objects instantiated by classes.

At the upper or meta-level, an *EntityReader* reads the entities and their attributes, and instantiates object instances of the *EntityComposite* class. These object instances of the domain entities are used by the *EntityGenerator* to instantiate the code templates and to generate domain classes like *InvoiceComposite* and *InvoiceReader*. At the central or domain level, a generated reader class like *InvoiceReader* reads the invoices and their invoice lines.

This basic code generation or metaprogramming framework has been extended to include the two additional characteristics.

- The metaprogramming becomes *meta-circular* by representing and reading the meta-entities in the same way as the domain entities, enabling the generation of the various classes at the meta-level itself.
- The metaprogramming becomes *autogenous* by generating a generator class for the domain entities as well, allowing to generate various artifacts for the instances of the domain entities themselves.

In the resulting meta-circular and autogenous code generation environment, there is not a single line of actual Java source code left. The only remaining source code consists of the four Java coding templates. Table I presents an overview of the size of the Java code in the four coding templates, detailing both the number of *Lines Of Code (LOC)* and the number of bytes. The *ExpansionContextExpander* template has not been mentioned yet, as it is a small technical helper class used mainly to setup some basic technical details, including the file paths for the CSV input files and the generated artifacts.

The small number of artifacts, and the very limited size of the final codebase, indicates that we have indeed created a quite elementary metaprogramming environment, that nevertheless exhibits meta-circular and autogenous code generation. The complete absence of any remaining source code is another confirmation of the elementary nature and limited complexity of the environment. In our opinion, this shows that the instantiation of coding templates through the evaluation of OGNL expressions in object data trees is a valuable metaprogramming pattern, and may even be close to being a fundamental technique for metaprogramming.

As mentioned before, the meta-circular nature of the metaprogramming environment avoids the complexity of two programming environments, i.e., one for the generator code

 ISBN: 978-1-61208-997-3

```
base() ::= <<
package $entity.packageName$;
public class $entity.name$Generator {
  public static void main(String [] args) throws Exception {
    String domainUri = "file:/"+args[0];
    ArrayList<$entity.name$Composite> instances = $entity.name$Reader.read$entity.name$s(domainUri);
    ArrayList<ExpanderPathComposite> expanders = ExpanderPathReader.readExpanderPaths(domainUri+"$entity.name$ExpanderPaths");
    ExpansionEngine expansionEngine = new ExpansionEngine(domainUri);
    Context expansionContext = expansionEngine.getExpansionContext();
    ArrayList<ExpanderComposite> expanderComposites = expansionEngine.getExpanders(expanders);
    for ($entity.name$Composite instance : instances) {
      $entity.name$ExpansionContext instanceContext = new $entity.name$ExpansionContext(instance, expansionContext);
      for (ExpanderComposite expanderComposite : expanderComposites) {
        expansionEngine.expand(instanceContext, expanderComposite);
      }
    }
  }
}
>>

package net.palver.invoice;
public class InvoiceGenerator {
  public static void main(String [] args) throws Exception {
    String domainUri = "file:/"+args[0];
    ArrayList<InvoiceComposite> instances = InvoiceReader.readInvoices(domainUri);
    ArrayList<ExpanderPathComposite> expanders = ExpanderPathReader.readExpanderPaths(domainUri+"InvoiceExpanderPaths");
    ExpansionEngine expansionEngine = new ExpansionEngine(domainUri);
    Context expansionContext = expansionEngine.getExpansionContext();
    ArrayList<ExpanderComposite> expanderComposites = expansionEngine.getExpanders(expanders);
    for (InvoiceComposite instance : instances) {
      InvoiceExpansionContext instanceContext = new InvoiceExpansionContext(instance, expansionContext);
      for (ExpanderComposite expanderComposite : expanderComposites) {
        expansionEngine.expand(instanceContext, expanderComposite);
      }
    }
  }
}
```

Figure 3. Representation of the instantiation of the generator code template for the *Invoice* entity.

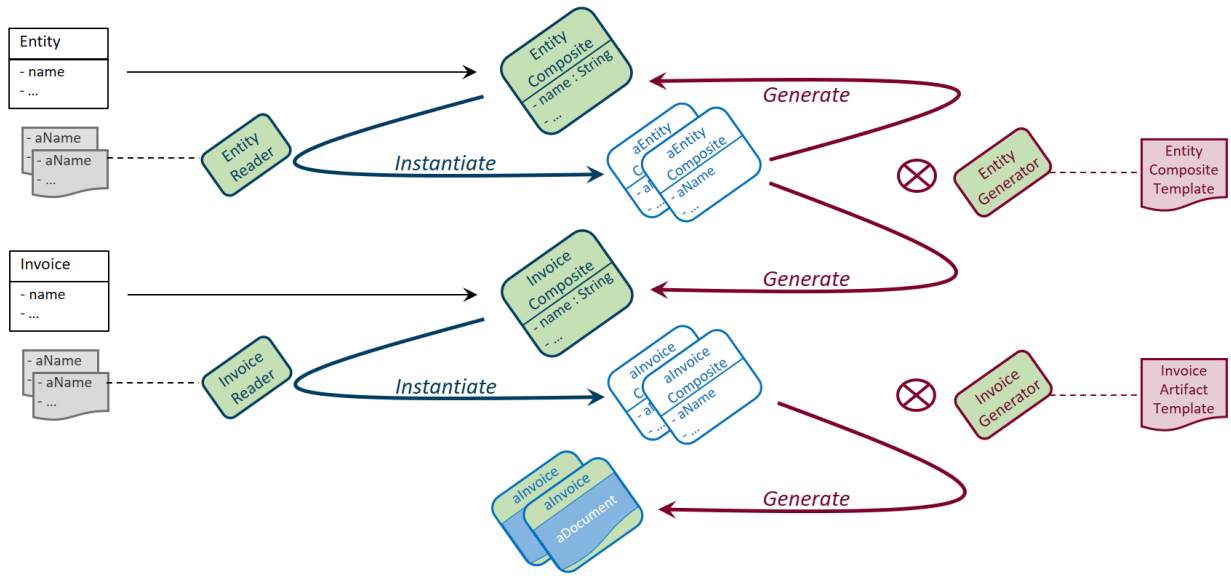

Figure 4. A graphical representation of the overall expansion.

 ISBN: 978-1-61208-997-3

and one for the generated code, and enables the simultaneous introduction of improved and/or extended programming techniques in *both generator code and the code that is generated*. Moreover, it fundamentally decreases the barrier to port such a (meta)programming environment to another programming platform and/or language. Indeed, porting the coding templates and target systems to another language would automatically port the metaprogramming software as well.

VI. Conclusion

Automated code generation or metaprogramming is often seen as an important, and maybe even crucial, approach to increase programming productivity. We have argued in previous work that, for reasons of software evolvability and scalable collaboration, such a metaprogramming environment should preferably exhibit a meta-circular architecture. As meta-circular metaprogramming might entail additional complexity, it seems desirable to provide guidance to metaprogrammers through structured concepts and techniques.

In this contribution, we have presented a detailed procedure to bootstrap an elementary but realistic meta-circular metaprogramming environment. Its purpose was to serve as an architectural pathfinder to clarify some basic concepts, and to support the future extraction of metaprogramming design patterns, or even more fundamental techniques.

The bootstrapping of this meta-circular metaprogramming environment is believed to make some contributions. First, the elementary nature of the environment offers a clear and structured view on both the concept of meta-circularity in code generation environments, and on the bootstrapping process that is needed to realize such an architecture. Second, we have also presented the emergence and basic mechanism of autogenous code generation, i.e., the generated code itself being able to generate artifacts. Third, the rather straightforward realization of meta-circular and autogenous code generation seems to indicate that replacing variables in coding templates through OGNL expressions in data instance trees might be a valuable pattern for metaprogramming.

Next to these contributions, it is clear that this paper is also subject to a number of limitations. The bootstrapping is performed for a single elementary metaprogramming in a single language environment. Additional work needs to be performed to extract and formulate more general design patterns and structured techniques for (meta-circular) metaprogramming. Nevertheless, we believe that this is a worthwhile pursuit, and we are planning to further explore this approach.

References

[1] H. Mannaert, J. Verelst, and P. De Bruyn, Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design. Koppa, 2016.

[2] R. Agarwal and A. Tiwana, "Editorial—evolvable systems: Through the looking glass of IS," Information Systems Research, vol. 26, no. 3, 2015, pp. 473–479.

[3] H. Mannaert, K. De Cock, P. Uhnak, and J. Verelst, "On the realization of meta-circular code generation and two-sided collaborative metaprogramming," International Journal on Advances in Software, no. 13, 2020, pp. 149–159.

[4] P. Kruchten, "Introduction: Software design in a postmodern era," IEEE Software, vol. 22, no. 2, 2005, pp. 16–18.

[5] D. Parnas, "Software aspects of strategic defense systems," Communications of the ACM, vol. 28, no. 12, 1985, pp. 1326–1335.

[6] P. Cointe, "Towards generative programming," Unconventional Programming Paradigms. Lecture Notes in Computer Science, vol. 3566, 2005, pp. 86–100.

[7] K. Czarnecki and U. W. Eisenecker, Generative programming: methods, tools, and applications. Reading, MA, USA: Addison-Wesley, 2000.

[8] J. R. Rymer and C. Richardson, "Low-code platforms deliver customer-facing apps fast, but will they scale up?" Forrester Research, Tech. Rep., 08 2015.

[9] C. Riemenschneider, B. Hardgrave, and F. Davis, "Explaining software developer acceptance of methodologies: A comparison of five theoretical models," IEEE Transactions on Software Engineering, vol. 28, no. 12, 2002, pp. 1135–1145.

[10] M. Huisman and J. IIvari, "The individual deployment of systems development methodologies," in Lecture Notes in Computer Science, A. Banks Pidduck, Ed., vol. 2348. Springer-Verlag, 2002, pp. 134–150.

[11] E. Gamma, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[12] K. Schwaber, Agile Software Development with Scrum. New Jersey, US: Prentice Hall, 2002.

[13] R. C. Martin, Design of modular structures for evolvable and versatile document management based on normalized systems theory. London, UK: Pearson, 2008.

[14] H. Mannaert, K. De Cock, and P. Uhnák, "On the realization of meta-circular code generation: The case of the normalized systems expanders," in Proceedings of the Fourteenth International Conference on Software Engineering Advances (ICSEA), November 2019, pp. 171–176.

[15] H. Mannaert, K. De Cock, P. Uhnák, and J. Verelst, "On the realization of meta-circular code generation and two-sided collaborative metaprogramming," International journal on advances in software, vol. 13, no. 3-4, 2020, pp. 149–159.

[16] J. Reynolds, "Definitional interpreters for higher-order programming languages," Higher-Order and Symbolic Computation, vol. 11, no. 4, 1998, pp. 363–397.

[17] C. Mooers and L. Deutsch, "Trac, a text-handling language," in ACM '65 Proceedings of the 1965 20th National Conference, 1965, pp. 229–246.

[18] "StringTemplate," URL: https://www.stringtemplate.org/, 2022, [accessed: 2022-06-15].

[19] T. Parr, "Enforcing Strict Model-View Separation in Template Engines," URL: https://www.cs.usfca.edu/ parrt/papers/mvc.templates.pdf, 2022, [accessed: 2022-06-15].

[20] "OGNL," URL: https://en.wikipedia.org/wiki/OGNL, 2022, [accessed: 2022-06-15].

[21] "Apache Commons OGNL - Object Graph Navigation Library," URL: https://commons.apache.org/proper/commons-ognl/, 2022, [accessed: 2022-06-15].

 ISBN: 978-1-61208-997-3