Exploring the Use of Code Generation Patterns for the Creation of Evolvable Documents and Runtime Artifacts

Herwig Mannaert and Gilles Oorts
Normalized Systems Institute
University of Antwerp, Belgium
Email: herwig.mannaert@uantwerp.be

Koen De Cock and Peter Uhnak
Research and Development
NSX BVBA, Belgium
Email: koen.de.cock@nsx.normalizedsystems.org

***Abstract*—Many organizations are often required to produce large amounts of documents in various versions and variants. Though many solutions for document management and creation exist, the streamlined automatic generation of modular and evolvable documents remains challenging. It has been argued in previous work that a meta-circular metaprogramming architecture enables a modular creation of source artifacts with very limited programming. In this contribution, a proof of concept is explored to generate modular LaTeX documents from runtime information systems through the use of a reduced version of this metaprogramming environment. The actual generation of several basic administrative document sources is explained, and it is argued that this architecture can easily be applied to generate other types of source artifacts using live runtime data.**

***Index Terms*—Evolvability; Normalized Systems Theory; Metaprogramming; Document Creation; Single Sourcing**

I. Introduction

Organizations are often required to produce large amounts of versions and variants of certain documents. While they have traditionally focused their efforts into streamlining technical product documentation [1], they are now also looking to build business value by creating personalized customer-faced documents [2]. At the same time, current information systems are producing massive amounts of relatively simple documents, e.g., invoices and timesheets, based on corporate data.

The streamlined and possibly automatic generation of such documents leads to concepts like modularization and single sourcing [1], both reminiscent of similar techniques used in the creation of software. Just like in software, dealing with versions and variants requires the design of document structures that deplete the rippling of changes in order to provide a level of evolvability [3]. Moreover, the use of parameter data during the instantiation of document variants seems similar to the inner workings of code generation environments.

In our previous work, we have presented a meta-circular implementation of a metaprogramming environment [4], and have argued that this architecture enables a scalable collaboration between various metaprogramming projects featuring different meta-models [5][6]. In this contribution, we investigate the use of a reduced version of this code generation environment within the generated software applications. More specifically, we explore the creation of evolvable artifacts, such as documents, where runtime data of the generated software application is used to instantiate the artifacts.

The remainder of this paper is structured as follows. In Section II, we briefly discuss some aspects and terminology related to the creation and single sourcing of documents, an important class of artifacts created by information systems at runtime. In Section III-A, we explain the basic concept of Normalized Systems Theory with regard to the design of evolvable artifacts. Section III-B recapitulates the architecture of our meta-circular code generation environment, and explains that this expansion of source code artifacts is not limited to programming code. Section IV presents how the generation environment can be configured to instantiate and expand runtime artifacts such as documents using live data. Finally, we present some conclusions in Section V.

II. Modular and Evolvable Document Creation

While organizations have traditionally focused their efforts in document management into streamlining product documentation [1], there is a widespread belief that personalized customer-faced documents can build business value by enhancing customer loyalty [2]. However, repurposing internal documents to be used for online purposes such as sales, marketing, product documentation and customer support has proven to be diffcult. Moreover, it is hard to find any best practices or repeatable models developed that address this challenge [1]. In this section, we briefly discuss some techniques and issues regarding the creation of evolvable documents.

A. Document Creation and Single Sourcing

A successful approach to handle any complex system or problem is modularization [7][8]. An example of such an approach in the area of document management is *Component Content Management (CCM)*, defined as *a set of methodologies, processes, and technologies that rely on the principles of reuse, granularity, and structure to allow writers to author, review, and repurpose organizational content as small components* [1]. One of the fundamental ideas of component content management is the separation of content and layout [9]. The granularity of a component in CCM is defined by the smallest unit of usable information [10]. Several standards exist that

 ISBN: 978-1-61208-850-1

define practical and technical implementation guidelines for creating modular and reusable content. According to Andersen and Batova [1], the most widely implemented standard is the *Darwin Information Typing Architecture (DITA)*.

Originally regarded as the broader discipline of CCM in the early 2000s, *single sourcing* has been defined as one of the fundamental aspects of CCM concerned with the design and production of modular, structured content. An elaborate description of single sourcing and its concepts, advantages, methodology, guidelines and practical examples, can be found in [11]. There are three fundamental aspects to single sourcing. First, content is made *reusable* by separating content from format. A second aspect is *modular writing*. Content is written in stand-alone modules instead of whole documents. This allows content to be assembled into documents from *singular source files that contain unique content*, the third aspect of single sourcing. Besides *assembling* the content modules into documents, i.e., combining source files in a hierarchical and sequential way with a distinct combination of audience, purpose and format, the modules need to be *linked*, i.e., connected to make them into coherent documents.

Enabling the content creators to focus on the actual substance of documents instead of having to deal with layout and publishing technologies, should lead to various advantages: saving time and money, improving document usability, and increasing team synergy [11]. Single sourcing recognizes two types of document creation. *Repurposing* entails merely reusing content modules for a different output format. *Re-assembly* on the other hand, is a more impactful way of reusing modules to develop documents for different purposes or audiences. Contrary to repurposing, re-assembly also includes changing the sequence of modules, the conditional inclusion, and the hierarchical level of inclusion.

B. Modular and Parametrized Document Generation

The emergence of concepts like modularization, CCM, and single sourcing regarding the management of certain classes of documents, e.g., technical documentation or personalized documents, is highly reminiscent of similar concepts in software codebases. Indeed, software developers have been striving for decades to modularize codebases, to separate concerns into singular source files, and to assemble source code modules into software applications, in a continuous effort — or quest — to reuse and repurpose these source modules.

Both documents and software source bases can have successive *versions* in time that contain additions, corrections or omissions to its content, and can be branched into concurrent *variants* when variations in content and/or purpose occur. Just like in software, dealing with versions and variants of a document requires the design of document structures to provide a desired level of evolvability. Evolvable documents are documents that do not hinder or limit the application of changes made to their structure or content. They are free from ripple effects that would cause changes to the documents to be highly difficult and costly [3].

Documents, such as technical and/or personalized documents, can have many concurrent variants. In technical documents, these variants range from the variation or even conditional presence of entire technical descriptions and procedures due to differences in the components of various installations, to simple parameter values like the serial number, color, or location of the documented installation or product. But short and simple documents, like letters, invoices or timesheets, can also be considered to have many variants due to different parameter values. This aspect of parameter-based or *model-based instantiation of document variants*, is highly reminiscent of environments for *code generation in software*.

III. Expansion of Evolvable Modular Structures

In this section, we discuss the expansion and assembly of evolvable modular structures. We introduce *Normalized Systems Theory (NST)* as a theoretical basis to design information systems —and conceptually other kinds of modular structures— with higher levels of evolvability, and its realization in a framework to generate and assemble programming code, and possibly other types of source artifacts.

A. Normalized Systems Theory and Evolvable Structures

NST was proposed to provide an ex-ante proven approach to build evolvable software [12], [13], [14]. It is theoretically founded on the concept of *systems theoretic stability*, a well-known systems property demanding that a bounded input should result in a bounded output. In the context of information systems, this implies that a bounded set of changes should only result in a bounded impact to the software. This implies that the impact of changes to an information system should only depend on the size of the changes to be performed, and not on the size of the system to which they are applied. Changes causing an impact dependent on the size of the system are called *combinatorial effects*, and considered to be a major factor limiting the evolvability of information systems. The theory prescribes a set of theorems and formally proves that any violation of any of the following *theorems* will result in combinatorial effects (thereby hampering evolvability) [12], [13], [14]:

- *Separation of Concerns*
- *Action Version Transparency*
- *Data Version Transparency*
- *Separation of States*

Applying the theorems in practice results in very fine-grained modular structures in software applications, which are in general difficult to achieve by manual programming. Therefore, the theory also proposes a set of patterns to generate significant parts of software systems which comply with these theorems. More specifically, NST proposes five *elements* that serve as design patterns for information systems [13][14]:

- *data element*
- *action element*
- *workflow element*

 ISBN: 978-1-61208-850-1

- *connector element*
- *trigger element*

Based on these elements, NST software is generated in a relatively straightforward way. Due to this simple and deterministic nature of the code generation mechanism, i.e., instantiating parametrized copies, it is referred to as *NS expansion* and the generators creating the individual coding artifacts are called *NS expanders*. This generated code can be complemented with custom code or *craftings* at well specified places (anchors) within the skeletons or boiler plate code. This results in the structural separation of four dimensions of variability [14][6]:

1) *Mirrors* representing data and flow models, using standard techniques like ERD (Entity Relationship Diagram) and BPMN (Business Process Model and Notation).
2) *Skeletons* expanded by instantiating the parametrized templates of the various element patterns.
3) *Utilities* corresponding to the various technology frameworks that take care of the cross-cutting concerns.
4) *Craftings* or custom code to add non-standard functionality that is not provided by the skeletons.

It has been extensively argued that the design theorems and structures of NST are applicable to all hierarchical modular architectures that exhibit cross-cutting concerns [15]. More specifically related to documents, the software theorems and element patterns of NST are very similar to the principles of CCM, that rely on reuse and fine-grained modular structures to allow writers to author, review, and repurpose organizational content as small components, and to the concept of single sourcing, demanding the separation of content and layout. Moreover, it has been shown that the application of NST to the design of evolvable document management systems leads to architectures that are in accordance with the principles of CCM and single sourcing [3], [16], [17].

B. Meta-Circular Code Generation or Artifact Expansion

NST has been realized in software through a code generation environment to instantiate instances of the various elements or design patterns. Due to the simple and deterministic nature of this code generation, i.e., instantiating parametrized copies, it is referred to as *NS expansion*. We have also argued that metaprogramming or code generation environments exhibit a rather similar and straightforward internal structure [5], [6], distinguishing:

- *model files* containing the model parameters.
- *reader classes* to read the model parameter files.
- *model classes* to represent the model parameters.
- *control classes* to select and invoke the generator classes.
- *generator classes* instantiating the source templates, and feeding the model parameters to the source templates.
- *source templates* containing the parametrized code.

As the NST metaprogramming environment was developed for the creation of web information systems, it has always included the generation of various building blocks, e.g., reader and model classes, which are similar to those of the code generation environment itself. This has made it possible to merge those generated code modules with the corresponding code generation modules, thereby evolving the metaprogramming environment into a meta-circular architecture [4]. This meta-circular architecture, described in [4], [5], [6], is schematically represented in Figure 1 and entails several advantages. First,

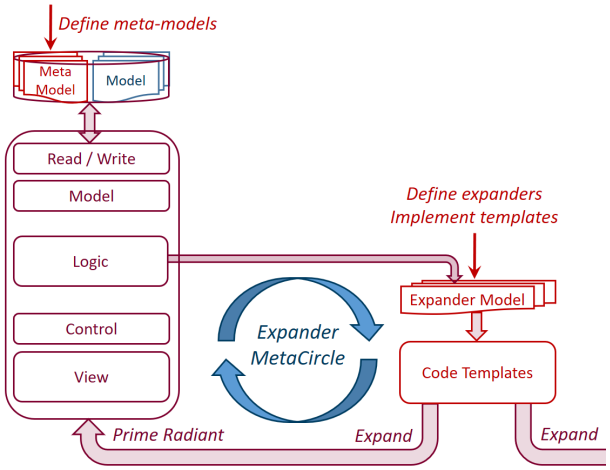

Fig. 1. The meta-circular architecture for NS expanders and meta-application.

this architecture enables the *regeneration of the metaprogramming code itself*, thereby avoiding the growing burden of maintaining the often complex meta-code, such as adapting it to new technologies. Second, it allows for a structural decoupling between the two sides of the code generation transformation, i.e., the domain models and the code generating templates. This also removes the need for contributors to get acquainted with the — basically non-existing —internal code structure of the metaprogramming environment, as additional expanders with corresponding coding templates can be defined and activated using a declarative control mechanism.

Moreover, the definition of additional meta-models and/or templates is not limited to programming code either. Instead of containing *Java* or *JavaScript* code, the templates may just as well correspond to hierarchical document modules such as chapters and sections, containing commands and settings of typesetting systems like Markdown/Pandoc or LaTeX. And any model representing parameter data and/or small content components may serve as a meta-model and drive the expansion or instantiation of the document. The meta-circular architecture does not require any explicit programming to support the new model entities representing the document. As we have seen, the various classes corresponding to the new model entities (XML readers and writers, model classes, control and generator classes) will be automatically generated.

IV. Exploring Runtime Expansion of Artifacts

In this section, we explore the assembly or expansion of parametrized documents using the NST meta-circular code generation environment.

 ISBN: 978-1-61208-850-1

A. Document Creation and Information Systems

As explained in Section II, an interesting duality exists between information systems and document creation. Information systems often support the creation of simple documents, such as invoices or timesheets, incorporating data that is entered and managed within the information system. At the same time, the streamlined creation of large amounts of document variants, for instance in the case of technical product documentation, requires some tooling to specify and manage the various parameters driving the creation of the document variants. In other words, information systems often create documents, and document creation systems usually require a supporting information system.

For the exploratory development targeted at the creation of documents using the NST meta-circular code generation environment, we have opted for the first scenario. The streamlined creation of variants of complex documents would require the definition of an elaborate meta-model describing the structure and domain parameters of the documents. The creation of such a model is out of scope of this contribution, but does not seem to pose a significant risk. Therefore, we decided to explore the generation of rather simple documents based on common data entities like invoices or timesheets. Nevertheless, this explorative development does address a possible and important technological hurdle. As these documents need to incorporate runtime data from the live information systems, e.g., the actual details of the various invoices, this proof of concept validates the expansion of artifacts based on runtime data from any information system expanded by the NST metaprogramming environment. In this way, the development can also serve as a validation for the expansion of other source artifacts based on live runtime data of information systems, such as marketing emails or sensor configuration files.

B. Declarative Control and Runtime Expansion

Consider two samples of a simplified data model for an administrative information system as presented in Figure 2.

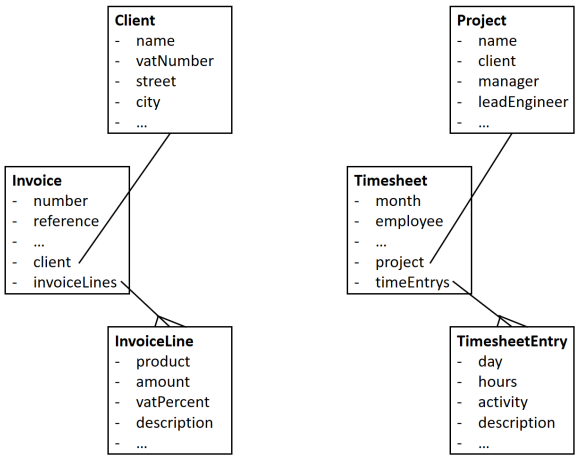

Fig. 2. Samples of a simplified data model for an administrative system.

- An *invoice* with some attributes, e.g., an invoice number and reference, containing a reference to a client, and consisting of several invoice lines.
- A *timesheet* with some attributes, e.g., the month and employee, containing a reference to a project, and consisting of several timesheet entries.

These data entities are expanded into *data elements*, collections of software classes as described in [6], by the NST metaprogramming environment, and incorporated in an information system. The expanded data elements or collections of classes include:

- Reader and writer classes to read and write the XML data files, e.g., *InvoiceXmlReader* and *InvoiceXmlWriter*, *TimesheetXmlReader* and *TimesheetXmlWriter*.
- Model classes to represent and transfer the various entities, and to make them available as an object graph, e.g., *InvoiceDetails* and *InvoiceComposite*, *TimesheetDetails* and *TimesheetComposite*.
- View and control classes to perform *CRUDS (create, retrieve, update, delete, search)* operations in a generated table-based user interface.

In the same way that the instances of the NST meta-model data elements are read and made available as an object graph at the time of code generation, the instances of the data elements represented in Figure 2 can be made available as an object graph at runtime in a generated information system. Incorporating the core templating engine of the NST metaprogramming environment [6] allows to evaluate the various attributes of the administrative data entities using *Object-Graph Navigation Language (OGNL)* expressions, and to feed them to the (LaTeX) templates that are used to create the invoice and timesheet documents.

As explained in [6], the expansion of artifacts, e.g., source code or document files, is based on a generic *ArtifactExpander* that uses declarative control to evaluate the model parameters and insert them into the source templates. Every individual expander generating a source artifact is defined in an *Expander* XML document. An example of the definition of such an individual expander to expand a LaTeX source file for an invoice is shown below. It is quite similar to the declaration of an expander creating a Java source file during code generation, but has a *TEX* source type and uses for instance the runtime invoice number to construct the filename.

```
<expander name="TexInvoiceExpander"
  xmlns="http://normalizedsystems.org/expander">
  <packageName>net.palver.latex.invoice</packageName>
  <layerType name="ROOT"/>
  <technology name="COMMON"/>
  <sourceType name="TEX"/>
  <elementTypeName>Invoice</elementTypeName>
  <artifact>Invoice-$invoice.number$.tex</artifact>
  <artifactPath>$expansion.directory$/
                $artifactSubFolders$</artifactPath>
  <isApplicable>true</isApplicable>
  <active value="true"/>
</expander>
```

The evaluation of the various instance parameters or attributes is based on OGNL expressions and defined in a separate *ExpanderMapping* XML document. This ensures the separation of content from format, as required by [9] to have reusable and evolvable documents. An example of the definition of such an individual mapping document for the invoice creation is shown below. Besides simple OGNL expressions, it allows to evaluate logical expressions, e.g., whether the invoice client is foreign for VAT purposes, and to make lists of linked objects and their attributes available, e.g., invoice lines.

```
<mapping
 xmlns="https://schemas.normalizedsystems.org/
                xsd/expanders/2021/0/0/mapping">
 <value name="info" eval="invoice.info"/>
 <value name="number" eval="invoice.number"/>
 <value name="client" eval="invoice.client.name"/>
 <value name="vatNr" eval="invoice.client.vatNr"/>
 <value name="street" eval="invoice.client.street"/>
 <value name="city" eval="invoice.client.city"/>
 <value name="isForeign"
  eval="!invoice.client.country.equals('Belgium')"/>
 <list name="invoiceLines"
       eval="invoice.invoiceLines"
       param="invoiceLine">
  <value name="info" eval="invoiceLine.info"/>
  <value name="product" eval="invoiceLine.product"/>
  <value name="amount" eval="invoiceLine.amount"/>
 </list>
</mapping>
```

The values as defined in the expander mapping document are passed to the LaTeX templates. As described in [5], the NST environment uses the *StringTemplate (ST)* engine library. This library supports the creation of a modular document structure by providing *subtemplate include* statements, enabling the document designers to adhere to the principles of single sourcing [11]. For instance, we share the declaration of various LaTeX packages and the definition of some basic commands through the use of the subtemplates `<basePackages()>` and `<baseCommands()>`. And the various invoice lines of an invoice (or timesheet entries of a timesheet) are created by instantiating a corresponding subtemplate for every list item through `<invoiceLines:invoiceTableLine()>` (or `<timesheetEntrys:timesheetTableLine()>`).

A reduced version of the NST metaprogramming environment was integrated into a runtime installation of an expanded information system that included the data elements represented in Figure 2. Based on live data from this runtime environment, tens of LaTeX sources for invoices and timesheets were successfully generated through the use of the expander declarations and parameter evaluations as presented above. It is clear that this expansion architecture allows information systems to create other type of source artifacts based on live runtime data. Indeed, as the NST expansion environment is agnostic with respect to the source type, e.g., able to create LaTeX source documents in exactly the same way as Java source files, the generation of other types of source modules is basically reduced to creating other types of templates. As possible use cases, we mention the creation of HTML emails for marketing purposes, and the assembly of configuration files that can be uploaded to remote IoT sensors or controllers.

V. Conclusion

Many organizations are often required to produce large amounts of versions and variants of documents in areas like technical documentation and accreditation. At the same time, corporate information systems are producing massive amounts of relatively simple documents based on corporate data. The streamlined and possibly automatic generation of such documents leads to concepts like modularization and single sourcing, which are similar to techniques used in code generation software. As in software, dealing with versions and variants requires the design of document structures to deplete the rippling of changes in order to provide a desired level of evolvability.

In our previous work, we have presented a meta-circular implementation of a metaprogramming environment, and have argued that this architecture can be used for code generation based on different and even newly defined meta-models. In this contribution, we have investigated the use of this code generation environment within the generated information systems at runtime. More specifically, we have explored the creation of evolvable artifacts, such as simple administrative documents, where runtime data of the generated software application is used to instantiate the artifacts.

We have shown in this contribution how a reduced version of the NST metaprogramming environment can be integrated within the runtime environment of an expanded information system, and how object graphs containing runtime data can be passed to source templates. We have demonstrated that we can use this runtime metaprogramming environment to successfully produce sources for administrative documents through declarative definitions and OGNL evaluations, without requiring dedicated software programming.

This paper provides different contributions. First, we validate that it is possible to use the NST metaprogramming environment to create another type of source code artifacts, e.g., LaTeX documents. Moreover, we have explained that this implementation adheres to several fundamental concepts regarding modular and evolvable document creation, like CCM and single sourcing. Second, we show that we can integrate a reduced version of the NST metaprogramming environment into a runtime information system expanded by the NST metaprogramming environment, and to generate source artifacts from live data within this running information system.

Next to these contributions, it is clear that this paper is also subject to a number of limitations. We have only demonstrated a single case of generating another type of source artifacts, i.e., LaTeX documents, outside the scope of the generation of software programming code. Moreover, the generated documents are quite simple, and in line with documents that are currently generated by mainstream information systems. Nevertheless, this explorative proof of concept can be seen as an architectural pathfinder, and we are planning to extend both the scope and

 ISBN: 978-1-61208-850-1

size of the generated documents, and the range of possible source types of artifacts that are generated.

REFERENCES

[1] R. Andersen and T. Batova, "The current state of component content management: An integrative literature review," IEEE Transactions on Professional Communication, vol. 58, no. 3, 2015, pp. 247–270.

[2] S. Abel and R. A. Bailie, The Language of Content Strategy. Laguna Hills, CA, USA: XML Press, 2014.

[3] G. Oorts, Design of modular structures for evolvable and versatile document management based on normalized systems theory. Antwerp, Belgium: University of Antwerp, 2019.

[4] 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.

[5] H. Mannaert, C. McGroarty, K. De Cock, and S. Gallant, "Integrating two metaprogramming environments : an explorative case study," in Proceedings of the Fifteenth International Conference on Software Engineering Advances (ICSEA), October 2020, pp. 166–172.

[6] 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.

[7] H. Simon, The Sciences of the Artificial. MIT Press, 1996.

[8] C. Y. Baldwin and K. B. Clark, Design Rules: The Power of Modularity. Cambridge, MA, USA: MIT Press, 2000.

[9] D. Clark, "Content management and the separation of presentation and content," Technical Communication Quarterly, vol. 17, no. 1, 2007, pp. 35–60.

[10] F. Sapienza, "A rhetorical approach to single-sourcing via intertextuality," Technical Communication Quarterly, vol. 16, no. 1, 2007, pp. 83–101.

[11] K. Ament, Single Sourcing: Building Modular Documentation. Norwich, NY, USA: William Andrew Publishing, 2003.

[12] H. Mannaert, J. Verelst, and K. Ven, "The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability," Science of Computer Programming, vol. 76, no. 12, 2011, pp. 1210–1222, special Issue on Software Evolution, Adaptability and Variability.

[13] ——, "Towards evolvable software architectures based on systems theoretic stability," Software: Practice and Experience, vol. 42, no. 1, 2012, pp. 89–116.

[14] 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.

[15] H. Mannaert, P. De Bruyn, and J. Verelst, "On the interconnection of cross-cutting concerns within hierarchical modular architectures," IEEE Transactions on Engineering Management, 2020, pp. 1–16.

[16] G. Oorts, H. Mannaert, and P. De Bruyn, "Exploring design aspects of modular and evolvable document management," in Proceedings of the Seventh Enterprise Engineering Working Conference (EEWC), May 2017, pp. 126–140.

[17] G. Oorts, H. Mannaert, and I. Franquet, "Toward evolvable document management for study programs based on modular aggregation patterns," in Proceedings of the Ninth International Conferences on Pervasive Patterns and Applications (PATTERNS), February 2017, pp. 34–39.

 ISBN: 978-1-61208-850-1