Gentleman:
A Light-weight Web-based Projectional Editor Generator

Louis-Edouard Lafontant
Université of Montréal
Montreal, Canada
louis.edouard.lafontant@umontreal.ca

Eugene Syriani
Université of Montréal
Montreal, Canada
syriani@iro.umontreal.ca

ABSTRACT

In the activity of software development and modeling, users should benefit from as much freedom as possible to express themselves, and this characteristic also extends to the tools they use. In recent years, projectional editors have proven to be a valid approach to obtain such capabilities by enabling language extension and composition and various notations. However, current solutions are heavyweight, platform-specific, and suffer from poor usability. To better support this paradigm and minimize the risk of arbitrary and accidental constraints in expressivity, we introduce Gentleman, a lightweight web-based projectional editor generator. Gentleman allows the user to define a model and projections for its concepts, and use the generated editor to create the model instances. We demonstrate how to define a projectional editor for Mindmap modeling, covering model definition, text and table projection, multi-projection, and styling to showcase its main features.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Graphical user interface languages**; *Interface definition languages.*

KEYWORDS

Projectional editing, model-driven engineering, language workbench

ACM Reference Format:
Louis-Edouard Lafontant and Eugene Syriani. 2020. Gentleman: A Light-weight Web-based Projectional Editor Generator. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, October 16–23, 2020.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3417990.3421998

1 INTRODUCTION

The practice of model-driven engineering (MDE) relies heavily on the use of models and domain-specific languages (DSL) [23] which offer, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [21]. Over the years, many tools have been created to support this activity, giving birth to a new category of tools labeled as *language workbench* [7]. They support the efficient definition, reuse, and composition of languages and their IDEs [6]. However, the current state of tooling has some limitations that slow down the adoption of MDE and its derived paradigms [11, 24]. Many challenges revolve around the modeling languages [20] and the tools used in the process such as modeling editors. We identify two limitations of modeling editors: (1) the level of expressivity and flexibility induced by the tool and (2) a deficiency in terms of usability, making their usage difficult for domain experts and practitioners alike.

Language workbenches offer a model editor that enables users to manipulate their models using the syntax of the DSL. Most editors are parser-based and can be classified into two categories. On the one hand, Free-form editors are typically used for textual DSL, like Xtext [4] and Spoofax [13]. On the other hand, syntax-directed editors are typically for graphical DSLs, like MetaEdit+ [17] and AToMPM [19]. The difference between the two lies in their parsing technique as they both rely on a parser to build an abstract syntax tree (AST) with the given input and validate the syntax. A projectional editor, however, does not rely on parsers. As a user edits a program, the AST is modified directly. Projection rules are used to create a representation of the AST with which the user interacts, reflecting the resulting changes [22]. Without a parser, it enables the support of notations that cannot be easily parsed, such as tables or mathematical formulas, and the composition of any language without introducing syntactic ambiguities. As demonstrated in [3], this is much harder to achieve with parser-based tools. The most promising projectional editor in the MDE community is currently Jetbrains MPS [5]. However, it is a heavy-weight editor that cannot be easily integrated in other tools.

In this paper, we present Gentleman, a lightweight web-based projectional editor generator, which aims to close the gap between models and domain experts. Gentleman allows the user to define a model and projections for its concepts, and use the generated editor to create the model instances. We demonstrate how to define a projectional editor for Mindmap modeling, covering model definition, text and table projection, multi-projection, and styling to showcase its main features. The tool demonstration is available online[1].

2 OVERVIEW OF GENTLEMAN

Given the metamodel of a DSL and projections associated with its elements, Gentleman generates a modeling editor tailored to the DSL. It is a lightweight web solution, a much-requested feature according to [2], and thus no installation is required. The editor comes with very little restriction and can be adapted to various

MODELS '20 Companion, October 18–23, 2020, Virtual Event, Canada
© 2020 Copyright is held by the owner/author(s).
ACM ISBN 978-1-4503-8135-2/20/10.
https://doi.org/10.1145/3417990.3421998

[1]https://youtu.be/rPYouGPThKY

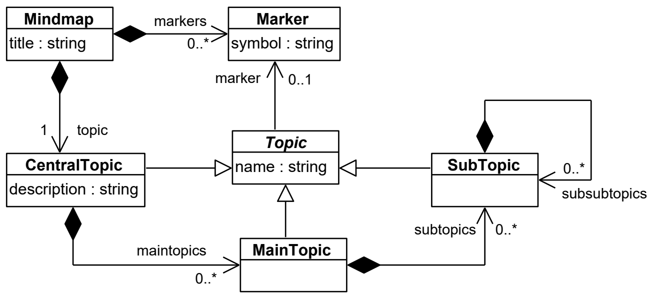


Figure 1: Mind map metamodel

HTML/CSS layouts and integrated into any web application. Gentleman relies on two main elements: concepts and projections define the metamodel and the concrete syntax of a DSL, respectively. The meta-languages used to define concepts and projections have been specified using Gentleman itself, making the editor bootstrapped. For convenience, the tool can import a metamodel, as an Ecore model, to automatically generate the concepts.

In the remainder of the paper, we will use a DSL for modeling Mindmaps as a running example. The metamodel is presented in Figure 1. A mind map is a structure used to organize information (topics) linked to and arranged around a central topic. The metamodel depicts a tree-like organization of main topics and sub-topics. Any topic can be assigned a marker.

2.1 Concepts

In Gentleman, the language engineer defines his DSL concepts using a collection of structures called *concept*. Conversely, a concept can be considered as the semantics of a projection. A collection of concepts and their relations are aggregated into a *model*. In Gentleman, we distinguish between four types of concepts: *primitive*, *concrete*, *prototype*, or *derivative* concepts. *Attributes* are used to associate concepts and can be encapsulated into *components*. Gentleman concepts borrow notions from object-oriented, prototype-based, and functional paradigms. For example, if the metamodel is defined by means of a class diagram, classes would be represented as concepts. To facilitate the creation of models, the language engineer can import any metamodel defined with Ecore into Gentleman. Table 1 outlines what some Ecore elements correspond to in terms of Gentleman concepts.

Ecore	Gentleman
EPackage	Model
EClass (abstract or interface)	Prototype Concept
EClass	Concrete Concept
EEnum	Derivative
EAttribute	Attribute
EReference (containment)	Component
EReference	Attribute

Table 1: Mapping between Ecore and Gentleman concepts

2.1.1 Attributes. The attribute of a concept (parent) describes its relationship with another concept (target); it is identified by a `name` (unique within the concept). An attribute may also possess a `description` and an `alias` that could be used to refer to the targeted concept. In this relationship, the target concept may be parameterized with constraints and augmented with additional properties to fit its parent concept usage. An attribute may be optional, meaning that an instance of the parent concept would be valid without a relation to the target concept. Every attribute declared inside a class or an association end in class diagrams are represented by concept attributes in Gentelman. In the Mindmap example, the attribute `title` and `marker` are declared as attributes of the concepts `Mindmap` and `Topic` respectively. The target concept of `marker` is the concrete concept `Marker`. In contrast, the target concept of `title` is the primitive concept `String`.

2.1.2 Components. The component of a concept (parent) is a group of related attributes extrinsic to the parent. They are not perceived as inherent to the concept itself and act as inner concepts visible only to its parent. This is similar to the notion of inner classes in object-oriented languages. Just as an attribute, a component is identified with a name, has an optional description, and an optional alias. For example, in Mindmaps, subtopics are components of a main topic because they are not intrinsic.

2.1.3 Primitive concepts. They are self-defined concepts and, therefore, not related to any other concepts: they have no attributes and thus no components. For better model integration, primitives are accessible globally to any model. Every concept can be resolved to a composition of primitives. Gentleman offers some predefined primitive concepts, like `String`, `Number`, `Boolean`, `Set` (collection of elements), and `Reference` (pointer to another concept). They contain specific properties on which constraints may be applied when defining an attribute to restrict the concept. For example, the string primitive of the `symbol` attribute has a constraint that it must be of length two. The language engineer can also define operations that may be used with instances of primitives, such as a text transformation with a string or arithmetic with numbers.

2.1.4 Concrete concepts. They represent the core concepts of the model and unlike primitives, they are specific to a model. Examples of such concept are `CentralTopic`, `MainTopic`, and `SubTopic`.

2.1.5 Prototype concepts. A prototype creates a base skeleton to provide reusability and extension to concepts of the model, similar to prototype-based programming. Any concept can reuse a prototype and would inherit its structure. Prototypes follow the Liskov substitution principle. If the target of an attribute, then any concept reusing it can also be the target. In this case, any property or constraint defined on the attribute or component would still hold. In Mindmaps, `Topic` is a suitable candidate for a prototype.

2.1.6 Derivative concepts. A derivative is a concept derived from another one (base). Every value that can be captured by a derivative must also be valid for its base concept. When the base is a primitive, it can serve as a form of specialization. For example, an enumeration would be translated into a derivative. In Mindmap, `Marker` can be represented as a derivative of `String`, restricting the list of accepted values. When the base is a concrete concept, the derivative could be used to define computed properties. For example, suppose `CentralTopic` is derived from `Mindmap`. Then, the `title` attribute of the latter could be the concatenation of the `name` attribute of the former prefixed with '`MM_`'.

2.2 Projections

A projection is a representation of a concept that can be visualized and interacted with in the graphical user interface (GUI). It can be applied to any part of a concept, such as the concept as a whole or an attribute. Note that the language engineer may define multiple projections for a concept. By doing so, we can obtain the right combination of visuals in any given situation. Gentleman offers predefined *layouts* found in modern GUI technologies [14]. It also provides data-specific controls in the form of *fields*, providing more structure to help users quickly scan and comprehend the information presented [12]. Both layouts and field can be customized with a *style*. At the moment, Gentleman only supports relative and tabular positioning with limited support for graphical elements. It does not provide the means to arbitrarily position elements, that would be possible with the use of HTML canvas, for example.

2.2.1 *Layout.*

A layout is concerned with the structure of a projection. It organizes elements presented in the GUI by indicating the location of its child elements. Gentleman defines layouts similar to those found in popular GUI frameworks such as Xamarin [10], SWT [9], and WPF [15]. For instance, the `StackLayout` piles elements horizontally or vertically, the `WrapLayout` group its elements in a block, and the `TableLayout` arranges them in a row or column-directed table. Every layout presents a container that can be configured to be collapsible, draggable, or resizable.

A layout child elements can be a text content, a layout, or a field. In the Mindmap instance presented in Figure 2, the organization of the elements is structured using the `StackLayout` to order them vertically. We use the `WrapLayout` to group them together, such as the heading `Mind Map <title="Planning">`. Each layout is further enriched with styling rules like color, border, and spacing values.

2.2.2 *Field.*

A field is concerned with manipulating the value of a concept; thus, it enables data input and output. It provides an abstraction for the underlying widget (control element) to promote reusability and portability. Gentleman uses a modular approach to select the right widget for the intent of the user. For instance, it selects a Textarea if the user intends to write text on multiple line or a single-line Textbox otherwise. Gentleman offers fields that cover the most fundamental widget components in GUIs [8]. Among others, a `TextField` allows the user to input characters for a `String` or `Number`. A `BinaryField` enables the user to alternate between two states of a concept value, such as for a `Boolean`. A `LinkField` allows the user to refer to another concept and bind its projection for a `Reference`. A `ChoiceField` enables the user to select one item in a predefined list for a `Prototype`. Each field offers specific customizations, such as the projection of each choice in a `ChoiceField` or the visual delimitation between items of a `ListField`. More advanced fields, like a `TableField` allow the user to manipulate structured data and provide the ability to add, remove, sort, and filter data. Fields also have generic properties to specify, for example, if they are read-only, disabled, or hidden.

In the Mindmap instance presented in Figure 2, the *title*, *name* and *description* attributes all target a `String` concept and, therefore,

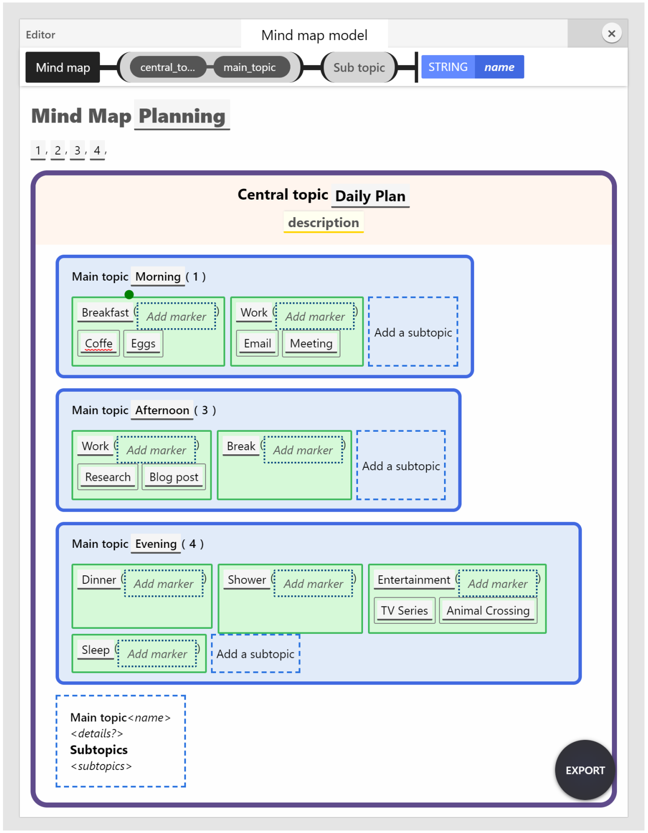


Figure 2: Instance of a Mindmap in Gentleman

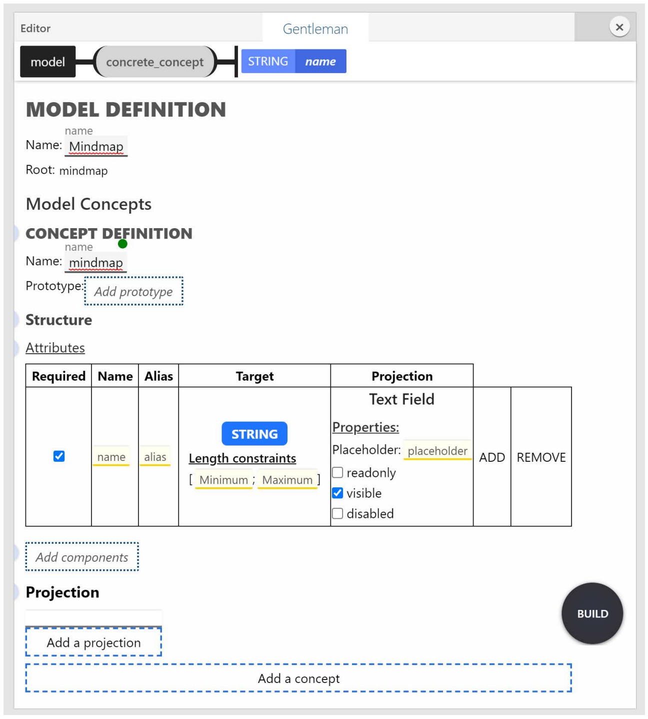
are rendered as `TextFields`. The marker added to the main topics are `LinkFields`, referencing the declared markers in the header. The main topics and subtopics are rendered as `ListFields` stacked horizontally. Additional main topics can be added to the central topic with the *Add* control action. Note how we customized this action field to display a template of the main topic structure.

2.2.3 *Style.*

Any projection can be complemented with style rules to describe its presentation. Styles can be defined directly in the Gentleman editor or imported and applied to a layout, text container, or field. For example, users can set the font, color, and alignment of text and the border of a table. To avoid repetition and encourage better integration, Gentleman leverages the browser technologies, offering full support for CSS class selectors. Layouts and fields also expose class selectors for their HTML elements. This enables the language engineer to declare global styles through CSS and specific context-based rules in Gentleman. Note that images can be added through the background property offered in CSS and Gentleman.

3 INTERACTING WITH THE EDITOR

Gentleman is designed to be lightweight and thus uses a minimalist approach to avoid any extraneous content that would otherwise distract and slow-down the user. The base editor only comes with a single toolbar with a button to close the editor and a status bar. Additional buttons can be added in the configuration of the editor.



MODEL DEFINITION

Name: Mindmap

Root: mindmap

Model Concepts

CONCEPT DEFINITION

Name: mindmap

Prototype: Add prototype

Structure

Attributes

Required	Name	Alias	Target	Projection		
✓	name	alias	STRING Length constraints [Minimum ; Maximum]	Text Field Properties: Placeholder: placeholder ☐ readonly ✓ visible ☐ disabled	ADD	REMOVE

Projection

Figure 3: Projectional editor to create a model in Gentleman

3.1 Usage scenarios

We distinguish three usage scenarios of Gentleman: (1) definition, (2) edition, and (3) reading. In (1), the user defines the concepts of a model and the projections as presented in Section 2. In this scenario, the target user is a language engineer or GUI designer involved in the DSL definition. For greater flexibility and reuse, he defines projections separately from concepts, thus providing good separation of concern. This is especially the case when the concepts are defined in an Ecore metamodel. In (2), the user creates or edits an instance of the DSL. The editor presents editable fields to add values to the AST of the model. Figure 3 shows a snapshot in this scenario. The purpose of (3) is to simplify the projection for users to read the model rather than edit it. It is a special case of (2) where fields are made read-only and interactive actions are disabled. For example, widgets to add, remove components, and empty optional attributes are not displayed. The target users of (2) and (3) are the domain experts of the DSL. To give users more flexibility during the editing activity, they can spawn as many editors as needed. This way, they can edit different concepts each in a dedicated space or use different projections simultaneously. The editors are juxtaposed next to each other and can be positioned as the user wants.

3.2 Tagging

Users can attach a *note* to any part of the content of the editor. It is not stored as part of the AST of the projection but as part of the editor. Notes can be tagged in the form of anchors. Users can search for tags to quickly navigate to specific locations of the model. One use of notes is to add comments on a model.

3.3 Storage

Import and export allow the user to preserve the current state of the model and editor or load a saved one. The storage medium is a JSON object representing the AST of the model and the projection configuration of each concept. It also stores configurations specific to the editor, like the toolbar configuration and comments. Recall that Gentleman is bootstrapped; therefore, it treats any instance being edited like a model. Internally, Gentleman does not distinguish between a model definition (a.k.a. metamodel), a projection definition (a.k.a. concrete syntax), or an instance. The export stores a reference to each concept and each projection. It is also possible to save the model in plain text with no formatting or projection by using the *print* functionnality.

3.4 Editor generation

When the language engineer has defined a model and at least one projection for each concept, he can automatically synthesize a projectional editor for his DSL. One attractive feature of Gentleman is its ability to preview a projection during the editing process. This allows the designer to view the presentation of the projection associated with the concept and how it integrates with other projections. It is also possible to edit the previewed projection to see how the design responds to different values entered and improve the user experience.

3.5 Context assistance

When the user interacts with a specific projection, a context in the status bar indicates the name and location of the current concept in terms of the structure of the model. At the top of Figure 2, we see that the currently active field corresponds to the name of a concret concept. The user can navigate through the model using the TAB key or mouse click. As a projectional editor, he can only modify editable projections. For example, in Figure 2, the user cannot remove the central topic; he can only set its name value or add/remove markers. During the interaction with a field, the user may request the accepted values that can be assigned to the field by hitting the common CTRL+Space key combination. The response depends on the state of the field and its concept: it pops a dialog showing information or a list of choices if an action is required. In Mindmaps, the marker attached to a main topic must have been defined as a central topic component. Therefore, at the level of the main topic, the context assistance lists the marker values defined at the central topic level. Requesting context assistance for the name of the main topic displays the attribute's meta-information, including constraints if they were set.

3.6 Model validation and feedback

After the user edits a field, an orange, red, or green badge is displayed next to it. They indicate that a value was modified since it was last in focus, that a constraint of the concept is violated, or that it is valid, respectively. In Figure 3 a green badge is displayed over the text field to indicate that the value entered is valid. When a constraint is not satisfied (e.g., the value is not unique), an additional description is fed back to the user, in the form of a dialog if it is purely informational or a choice dialog if further actions can be taken. As explained in Section 1, the model is always structurally valid. However, the DSL may have semantical constraints, such as a maximum depth of sub-sub topics. Gentleman supports reporting semantical constraints violations through its API.

4 IMPLEMENTATION

Gentleman targets the web as its running platform and is implemented entirely in Javascript. The application runs client-side, enabling offline work. As with any web application, HTML and CSS are used to describe the content and its presentation. The tool can be easily integrated into any web page with the Gentleman script loaded in it. This can be achieved in one of two ways. The developer can decorate an HTML Tag with the attribute `data-gentleman`, such as `<div data-gentleman></div>`. Upon loading, every HTML element on the page found with this attribute will have a Gentleman instance attached to it with the editor rendered inside. Alternatively, the developer can create a Gentleman instance dynamically in Javascript using the instruction `editor = Environment.createEditor()` followed by `editor.render()` to render the editor on the page. This enables a web-based language workbench (like AToMPM or WebGME) to have multiple projectional editors within a single modeling editor. This is useful, for instance, to control and stylize the edition of attributes. Gentleman is an open-source project available on GitHub[2].

5 RELATED WORK

Projectional editors have been implemented in a few language workbenches. Modern solutions include as Jetbrains MPS [5], the Whole Platform [18], and the discontinued projects Intentional Programming [16] and Más [1]. One of the most attractive features of Gentleman is that it is a web solution. This opens the editor to a wider audience as it is not specific to any platform: any device equipped with a web browser can use it. Màs took a similar approach, but none of the current solutions targets the web. They are limited to machines running their targeted operation system for which they build specific packages.

Current language workbenches offer very little integration with other tools and require every artifact to be created inside the tool. For example, MPS can only integrate, via plugin manipulations, with Jetbrains. Similarly, the Whole Platform is built as an Eclipse instance making it possible to create bridges between Eclipse plugins. Gentleman is designed with integration in mind, seamlessly done with web applications, and also offers a bridge to the Eclipse Modeling Framework.

Current projectional editors try to redefine well understood design notions, creating an additional cognitive barrier for their users. MPS, for instance, defines the visuals and their interaction by using what they call *Editor* and *Cell* which can be configured to act as a container, field, link or list; thus redefining the very semantic of what a cell is and how a design expert expects to use a cell. In contrast, Gentleman uses the same language found in design such as layout and field which are further classified for specific use. Furthermore, web designers can rely on their existing knowledge to customize projections in Gentleman.

[2]https://github.com/geodes-sms/gentleman

6 CONCLUSION

We presented the central notions of the projectional editor, Gentleman. The editor offers a rich GUI enabled by standard layouts and fields, and projections that can be applied to any part of the DSL concepts. Ultimately, our goal is to integrate Gentleman in full-fledged language workbenches to offer a more adapted user experience to domain users. Gentleman is an ongoing project with a roadmap filled with more novelties. We would like to integrate a model explorer, investigate proper undo/redo functionalities for projectional editors, and enable collaborative modeling. We are also working on providing a standard API based on the language server protocol to ease the integration with language workbenches.

REFERENCES

[1] [n.d.]. http://mas-wb.appspot.com/.
[2] L. Agner and T. Lethbridge. 2017. A survey of tool use in modeling education. In *Model Driven Engineering Languages and Systems*. IEEE, 303–311.
[3] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund. 2016. Efficiency of projectional editing: A controlled experiment. In *International Symposium on Foundations of Software Engineering*. 763–774.
[4] L. Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.
[5] F. Campagne. 2016. *The MPS Language Workbench Volume I: The Meta Programming System* (3rd ed.). CreateSpace Independent Publishing Platform.
[6] S. Erdweg, T. Van Der Storm, M. Völter, et al. 2013. The state of the art in language workbenches. In *International Conference on Software Language Engineering*. Springer, 197–217.
[7] M. Fowler. 2005. Language workbenches: The killer-app for domain specific languages. https://martinfowler.com/articles/languageWorkbench.html.
[8] V. Gupta. 2008. UI Programming: Handling events and using advanced widgets. *Accelerated GWT: Building Enterprise Google Web Toolkit Applications* (2008), 105–134.
[9] R. Harris and R. Warner. 2004. *The definitive guide to SWT and JFace*. Apress.
[10] D. Hermes. 2015. *Xamarin Mobile Application Development: Cross-Platform C# and Xamarin. Forms Fundamentals*. Apress.
[11] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. 2011. Empirical assessment of MDE in industry. In *International Conference on Software Engineering*. ACM, 471–480.
[12] J. Johnson. 2013. *Designing with the mind in mind: simple guide to understanding user interface design guidelines*. Elsevier.
[13] L. C. Kats and E. Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Object oriented programming systems languages and applications*. 444–463.
[14] Ó. S. Ramón, J. S. Cuadrado, and J. G. Molina. 2014. Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engineering* 21 (2014), 147–186.
[15] C. Sells and I. Griffiths. 2007. *Programming WPF: Building Windows UI with Windows Presentation Foundation*. O'Reilly Media, Inc.
[16] .C Simonyi, M. Christerson, and S. Clifford. 2006. Intentional Software. In *Object-Oriented Programming Systems, Languages, and Applications*. ACM, 451–464.
[17] K. Smolander, K. Lyytinen, V-P. Tahvanainen, and P. Marttiin. 1991. MetaEdit – a flexible graphical environment for methodology modelling. In *International Conference on Advanced Information Systems Engineering*. Springer, 168–193.
[18] R. Solmi. 2005. *Whole Platform*. Ph.D. thesis. Universitá di Bologna e Padova.
[19] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. 2013. AToMPM: A Web-based Modeling Environment. In *MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition*, Vol. 1115. CEUR-WS.org, 21–25.
[20] R. Van Der Straeten, T. Mens, and S. Van Baelen. 2008. Challenges in model-driven software engineering. In *Model Driven Engineering Languages and Systems*. Springer, 35–47.
[21] A. Van Deursen, P. Klint, and J. Visser. 2000. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices* 35, 6 (2000), 26–36.
[22] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. 2014. Towards user-friendly projectional editors. In *International Conference on Software Language Engineering (LNCS, Vol. 8706)*. Springer, 41–61.
[23] J. Whittle, J. Hutchinson, and M. Rouncefield. 2013. The state of practice in model-driven engineering. *IEEE software* 31, 3 (2013), 79–85.
[24] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal. 2013. Industrial adoption of model-driven engineering: Are the tools really the problem?. In *Model Driven Engineering Languages and Systems (LNCS, Vol. 8107)*. Springer, 1–17.