



Contribution to the Engineering of User Interfaces

Arnaud Blouin

► To cite this version:

Arnaud Blouin. Contribution to the Engineering of User Interfaces. Software Engineering [cs.SE]. Université de Rennes 1 [UR1], 2019. tel-02354530

HAL Id: tel-02354530

<https://theses.hal.science/tel-02354530>

Submitted on 7 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HABILITATION À DIRIGER DES RECHERCHES DE

L'INSA RENNES

COMUE UNIVERSITÉ BRETAGNE LOIRE

ECOLE DOCTORALE N° 601

*Mathématiques et Sciences et Technologies
de l'Information et de la Communication*

Spécialité : Informatique

Par

Arnaud Blouin

Contribution to the Engineering of User Interfaces

Habilitation présentée et soutenue à Rennes, le 29.08.2019

Unité de recherche : IRISA – UMR6074

Thèse N° :

Composition du Jury :

Rapporteurs

| | |
|-------------------|--|
| Gaëlle CALVARY | Professor Grenoble INP |
| Philippe PALANQUE | Professor Toulouse 3 University |
| Richard PAIGE | Professor McMaster University, Canada |

Examineurs

| | |
|---------------------|---|
| Yann-Gaël GUÉHÉNEUC | Professor Concordia University, Canada |
| Jean-Marc JÉZÉQUEL | Professor Rennes University |
| Jean VANDERDONCKT | Professor Louvain University, Belgium |

Contents

| | | |
|----------|--|-----------|
| 1 | Contribution to the Engineering of User Interfaces | 5 |
| 1.1 | Context | 5 |
| 1.2 | Challenges and Objectives | 7 |
| 1.3 | Scientific Contributions | 9 |
| 1.3.1 | Software engineering user interfaces: new user interface development abstractions | 9 |
| 1.3.2 | Improving the interactivity and usability of domain-specific languages | 11 |
| 1.4 | Research Methods | 13 |
| 1.4.1 | From insights to empirical evidences: the example of a study on UI listeners | 14 |
| 1.4.2 | Validating approaches empirically: the example of the UI listener refactoring tool | 18 |
| 1.5 | Projects and Supervision | 22 |
| 1.6 | Software Development | 26 |
| 2 | Research Perspectives | 29 |
| 2.1 | DevOps and user interfaces | 31 |
| 2.1.1 | Research Context | 31 |
| 2.1.2 | Scientific Challenges | 31 |
| 2.1.3 | Approach | 33 |
| 2.2 | Engineering domain-specific user interfaces | 37 |
| 2.2.1 | Research Context | 37 |
| 2.2.2 | Scientific Challenges | 37 |
| 2.2.3 | Approach | 39 |
| 2.3 | User interactions as a first-class programming concept | 42 |
| 2.3.1 | Research Context | 42 |
| 2.3.2 | Scientific Challenges | 42 |
| 2.3.3 | Approach | 45 |
| | Selected Publications | 49 |
| | Bibliography | 53 |

Chapter 1

Contribution to the Engineering of User Interfaces

Contents

| | |
|---|-----------|
| 1.1 Context | 5 |
| 1.2 Challenges and Objectives | 7 |
| 1.3 Scientific Contributions | 9 |
| 1.3.1 Software engineering user interfaces: new user interface development abstractions | 9 |
| 1.3.2 Improving the interactivity and usability of domain-specific languages | 11 |
| 1.4 Research Methods | 13 |
| 1.4.1 From insights to empirical evidences: the example of a study on UI listeners | 14 |
| 1.4.2 Validating approaches empirically: the example of the UI listener re- factoring tool | 18 |
| 1.5 Projects and Supervision | 22 |
| 1.6 Software Development | 26 |

1.1 Context

‘Anytime you turn on a computer, you’re dealing with a user interface’ [91]. User interfaces (UI) pervade our daily lives. To do office tasks, to pilot an airliner, to write programs, UIs are the tangible vectors that enable users to interact with software systems. The development of UIs involves multiple roles. Designers and ergonomists are in charge of the design and evaluation of UIs from a strict human factor viewpoint. They use concepts and theories established by the Human-Computer Interaction (HCI) community. Software engineers develop, validate, maintain UIs using software engineering techniques. UI engineering is an interdisciplinary field that cross-cuts these two roles and their underlying domains, HCI and software engineering. In the 80s Draper and Norman motivated the UI engineering field as follows:

The discipline of software engineering can be extended in a natural way to deal with the issues raised in a systematic approach to the design of human-machine interfaces. To a larger extent all that is needed is to take the problem of engineering the user interface as seriously as any other part of software engineering and to apply to it the same kind of techniques, appropriately adapted. [47]

The IFIP Working Group on UI engineering proposes a more technical definition of UI engineering:

UI engineering addresses all aspects related to methods, processes, tools, technologies, and empirical studies involved in the invention, design and construction of interactive systems [...] with a particular focus on principled, methodological engineering approaches. [65]

These definitions agree on one point: UIs are complex pieces of software that require specific development approaches. Indeed, UI engineering is not the crossroads of the HCI and software engineering domains, i.e., a domain that sounds hollow without underlying theories where concepts from the HCI and software engineering domains are simply assembled. UI engineering springs from and is backed with HCI and software engineering. Yet, UI engineering also relies on specific UI engineering theories. We can group the existing UI engineering approaches that develop these theories into two categories.

Approaches #1: Adapting software engineering techniques for engineering UIs

The first group of approaches adapts software engineering techniques for the specific purpose of engineering UIs. These approaches are numerous, such as: UI testing aims at adapting software testing techniques to test UIs [131, 85, 34]; software dynamic adaptation techniques were adapted to UIs [3, 29]; more generally, the UI engineering community widely studies model-driven engineering to overcome UI-related issues [128, 80, 32, 99].

Approaches #2: Injecting HCI concepts into software engineering processes, tools, methods

The second group of approaches integrates HCI concerns within software engineering processes. Several examples of topics are: HCI methods for designing and evaluating APIs [94] (*Application Programming Interface*) and languages [13]; rapid prototyping, that consists of proposing new engineering techniques to quickly develop UI prototypes [38].

The next Section 1.2 explains the scientific UI engineering challenges I focus on. Section 1.3 details the contributions I, with the essential help of the persons I work(ed) with, propose. These contributions are situated in the context of UI engineering, with a strong focus on software engineering. More precisely, the contributions focus on model-driven engineering, software validation and verification, and software variability, for engineering UIs.

1.2 Challenges and Objectives

UI engineering involves the software engineering and HCI communities that should work together, but as explained by Palanque:

Innovation and creativity are the main research drivers of the HCI community, which is currently investing a vast amount of resource in the design and evaluation of 'new' user interfaces and interaction techniques, leaving the correct functioning of these interfaces at the discretion of the helpless developers. [108]

HCI researchers, such as Beaudouin-Lafon, also discuss this problem of interactions between these two communities:

HCI researchers have created a variety of novel [user] interaction techniques and shown their effectiveness in the lab, such 'point designs' are insufficient. Software developers need models, methods, and tools that allow them to transfer these techniques to commercial applications. [15]

A gap thus exists between how the HCI community envisions a UI and how software engineers can implement it. This concerns the two groups of approaches described in the previous section:

Limits and challenges of using software engineering techniques for engineering UIs – Software engineers mostly rely on general-purpose languages (GPL) to develop software systems and UIs. GPLs, such as Java or JavaScript, aim at providing engineers with programming constructs (e.g., object-oriented constructs) to solve a large variety of problems. One issue is that UI engineering involves specific problems and thus specific constructs. Engineers and researchers progressively propose and use constructs specifically designed for engineering UIs. The definition of a UI structure typifies such new constructs. With old UI toolkits, such as Java Swing, engineers had to define the structure of a UI by programming it. To better match the declarative definition of a UI structure engineers and researchers have proposed UIDLs (*User Interface Description Language*) [80, 122, 126]. UIDLs describe the structure of a UI with adapted abstractions, languages and tools. Most of the current mainstream UI toolkits rely on a UIDL, such as XAML for WPF [121], FXML for JavaFX [43], or DOM for Web frameworks [60, 51, 93]. Researchers also worked on the role of UIDLs and how they can overcome other UI challenges such as dynamic adaptive UIs or multi-platform UIs [80, 32].

UI engineering has other major concerns that require dedicated constructs. This is the case of:

- **UI testing** – As any software artefact, UIs need to be tested;
- **UI analysis** – As any software artefact, UIs need to be analysed to find issues or for maintenance or testing operations;
- **UI variability** – Because of the large panel of devices and usages, UIs are no more monolithic but a product line.

Limits and challenges in injecting HCI concepts into software engineering processes, tools, methods – DSLs are interfaces that stand between domain experts and their engineering problems [89, 36]. For domain experts, the visible part of a DSL is its concrete syntax. Concrete syntaxes can take various forms, such as graphical, textual or tabular. These experts handle DSLs through dedicated editors that share similarities with standard IDEs (*Integrated Development Environment*) of GPLs: auto-completion, templating, error checking, or navigation, are standard interactive and usability features that IDEs of both GPLs and DSLs support.

However, the gist of a DSL is to focus on one specific problem. DSLs can thus have major conceptual and technical differences each other. This has impacts on how domain experts interact DSL. Specific interactive features may be adequate for one DSL, but not for other ones. Basic generic support is currently provided to DSL designers, such as auto-completion and error checking for textual DSLs. For the rest, DSL designers have to craft by hand supplementary interactive features for each DSL they develop. One goal of the software-language community is to ease the development of DSLs. To follow this trend, I defend that DSL development processes and tools should provide DSL designers with features to customise, improve the interactivity and usability of DSLs. Two intertwined sub-objectives composes this research line:

- **DSL interactivity** – Supplying DSL editors with time-honoured yet customisable interactive features to help domain experts;
- **DSL development** – Providing language designers with techniques, processes, tools to ease the development of DSLs.

In this habilitation I thus defend the following thesis as a global objective:

UI engineering research has to provide software engineers with theories and techniques at a correct level of abstraction and assessed empirically to help them in coding, testing, documenting usable modern UIs.

1.3 Scientific Contributions

All the contributions I develop in this section could not have been done without the help, the inspiration, the complementarity of the DiverSE research group in which I work since my PhD viva in December 2009 (research group called Triskell at this time).

1.3.1 Software engineering user interfaces: new user interface development abstractions

As for any software artefact, software engineers develop, test and maintain UIs. The development of UIs relies on graphical toolkits while testing UIs relies on UI testing libraries. These toolkits and libraries are built on top of HCI and software engineering concepts. The first scientific contributions I detail in the next paragraphs propose new UI engineering concepts. I detail how I turn them into concrete engineering tools and how I evaluated their impacts *in situ*. These concepts follow the same leitmotiv: engineering UIs is a specific problem that requires specific abstractions. These abstractions thus go beyond the classical object-oriented abstractions provided by the current programming languages to provide abstractions that focus on UI concerns.

UI testing [76, 77] – The first new abstraction is related to UI testing. As any code artefact, developers must validate UI code to provide users with high quality (from a strict software engineering viewpoint) UIs. So, software testers have paid special attention to UI testing in the last decade [11]. They have devised techniques that are effective in finding several kinds of UI errors. However, the introduction of new types of user interactions presents new kinds of errors that are not targeted by current testing techniques. We believe that to advance UI testing, the community needs a comprehensive and high level UI fault model, which incorporates all types of interactions. This contribution proposes a UI fault model designed to identify and classify UI faults. We designed this model empirically by analysing bug reports of real UIs. For each fault proposed in the model, we develop a mutant of a highly interactive software system that introduces an instance of the fault. The goal is to provide UI testers with examples of UI errors to find and to train UI testing tools. The proposed UI fault model aims at guiding software testers in building UI testing tools and writing UI tests based on the characterised UI faults. For example, we used the fault model to develop a tool in an industrial context to find UI errors in UIs of French power plant [77].

UI code analysis [26, 78] – Code analysing techniques extract information from the code or from the execution of the code to, for example, identify bad coding practices or help in generating tests. The second abstraction follows the same idea as for the proposed UI fault model: similarly to software testing that requires fault models, identifying UI bad coding practices requires to reason on UI information extracted from the code. The next contribution I present focuses on the analysis of object-oriented code that controls a UI to extract UI abstractions. UIs intensively rely on event-driven programming: interactive objects send UI events, which capture users' interactions, to dedicated objects called controllers. Controllers use several UI listeners that handle these events to produce UI commands. An empirical study we conducted revealed the presence of a design smell in the code that describes and controls UIs. This new design smell, called *Blob listener*, characterises UI listeners that can produce more than two UI commands. Because of the coupling of the identified design smell and the rest of the code, we proposed a systematic static code analysis procedure that

searches for *Blob listeners*. The technique permits to precisely identify UI commands and the widgets that produce these commands using the code of these UI elements. We then developed a semi-automatically and behaviour-preserving refactoring process to remove *Blob listeners*. We empirically validated the developed techniques by applying them on large open-source software systems. Developers of the studied systems accepted and merged patches we produced. Discussions with these developers assessed the relevance of the *Blob listener*.

UI product line [29, 72] – ‘The traditional focus of software engineering is to develop individual software systems, i.e., one software system at a time. [...] The result obtained is a single software product. In contrast, Software Product Line engineering (SPL) [101, 14, 113] focuses on the development of multiple similar software systems from a common codebase’ [2]. SPL eases the development of similar software systems or different versions of a same software system by promoting software reuse. This eases the static (i.e., at design time) or dynamic (i.e., at run time) adaptations of a software system to a change of configuration. As any software artefact, SPL concerns UIs [112]. In particular for statically or dynamically adapting UIs to a change of context [80, 32, 37]. The next contribution a present focuses both on static or dynamic UI product line.

First, UIs can adapt dynamically to context changes (platform, user, environment). Complex user interfaces and contexts can lead to the combinatorial explosion of the number of possible adaptations. Thus, dynamic adaptations come across the issue of adapting user interfaces in a reasonable time-slot with limited resources. We proposed to combine aspect-oriented modelling (AOM) with property-based reasoning to tame complex and dynamic user interfaces [29]. AOM approaches provide advanced mechanisms for encapsulating cross-cutting features and for composing them to form models [92]. Property-based reasoning consists in tagging objects that compose the system with characterising properties [53]. At run time, a reasoner uses these properties to perform adaptations based on the current context. Reasoning on a limited number of aspects combined with the use of properties overcomes the combinatorial explosion issue.

Second, development practices in virtual reality (VR) hardly follow the software engineering practices. In many cases, engineers develop each VR application from scratch without, for example, any code reuse from related VR applications. VR applications, however, share various specific artefacts such as scenarios, i.e., the ‘story’ of the VR applications [33, 35]. We propose methods to automate the development and evaluation of VR applications with the use of SPL techniques [72, 73]. We implemented these approaches inside tools that have been tried on examples and evaluated by their target users. The results promote the use of these frameworks for producing scenario-based software.

Note that I based most of the contributions of this first part on concepts I promoted during my PhD thesis with the Malai design pattern [22, 23]: Decomposing an interactive system into three blocks – the model, views and controllers [69] (or presenters [114], view models [121], etc.) – is not precise enough regarding the increasing interactivity of modern UIs. These architectures consider other well-established UI concepts, in particular the concepts of user interaction, UI command and undo/redo.

1.3.2 Improving the interactivity and usability of domain-specific languages

As for any user interface, engineering DSLs must make use of HCI concepts at different levels. First, as any user interface, DSL editors must be usable. Engineers must supply DSL editors with adapted interactive features to help their domain experts in using the DSL. Second, developing DSL is a complex engineering job that involves various tasks, such as developing, testing, maintaining editors, documentation, syntaxes [36]. Language designers must be helped during these tasks to propose to domain experts usable DSL editors. The second scientific contributions I detail in the next paragraphs follow these two lines.

Model slicing [25, 24] – I based most of the contributions of this part on the concept of *model slicing* I first present. Model slicing is a model comprehension technique inspired by program slicing [125]. The process of model slicing involves extracting from an input model a subset of model elements that represent a model slice. Model slicing provides a mechanism to isolate and focus on specific parts of a given model using input model elements called *slicing criteria*. For example, when seeking to understand a large class diagram, it may help to extract the sub-part of the diagram that includes only the dependencies of a particular class [27, 28]. Another slicing example is the extraction of the footprint of a model operation (i.e., identifying the model elements handled by the operation) [66] to do type checking [45] or improve model operation performance [129, 124]. In this work, we precisely define the theory of the model slicing operator. Based on this theory, we then designed a DSL, called *Kompren*, to model model slicers for a particular domain (captured in a metamodel). The primary objective of *Kompren* is the selection of classes and properties in an input metamodel. *Kompren* promotes the definition of slicers that slice all necessary elements to make the slice a valid instance of the input metamodel. We used *Kompren* on different use cases, as discussed in the following paragraphs.

Interactivity of DSL editors [27, 28, 19] – The second contribution of this part focuses on improving the interactivity of DSL editors. We worked on DSLs having graphical or tabular syntax with two different contributions. First, graphical syntaxes are widely used by language designers. Representative examples are Ecore [123] and the different UML languages [103] for software engineers, and ScratchJr [52] for kids. One issue with graphical syntaxes is the space on screens a model can take when opened, hindering the comprehension of the studied model. We propose a technique to improve the interactivity, more precisely the navigation through models, of graphical DSL editors [27, 28]. We based this technique on model slicing to slice the current model using filtering data. Model slicing serves as a dynamic filtering feature, where elements not related to the current concern of the user are hidden. We show that the proposed technique eases the navigation through large models and improve the understandability that users have of their models.

Second, Product Comparison Matrices (PCM) are a specific case of tabular DSLs. PCMs form a rich source of data for comparing a set of related and competing products over numerous features. Despite their apparent simplicity, PCMs contain heterogeneous, ambiguous, uncontrolled, and partial information that hinders their efficient exploitations. These DSLs are, for example, intensively used on Wikipedia to compare objects using their characteristics. The contribution we proposed aims at easing the use of PCMs by automatically analysing raw data to produce formalised PCMs with interactive features adapted to their content. The goal of this proposal is to improve the current practice of editing, maintaining and exploiting PCMs.

Easing the development of DSLs [45, 46, 86, 88, 87, 74] – Software engineers are users too [96]. This third contribution focuses on proposing to language designers new techniques to improve the development of DSLs. These contributions are not user interface engineering contributions *per se*; they are software engineering, more precisely language engineering, contributions. However, the goal of these contributions is to ease the work of language designers so that I consider these contributions as improving the usability of software language development tools and methods. I classify these contributions into three parts.

First, we focused on flexibility and reuse while creating DSLs [45, 46]. The theory behinds the current DSL tools faces several flaws that prevent language designers to freely create and maintain DSLs and their models. We worked on two of these flaws. Regarding the first flaw, we conducted an empirical study with UML models publicly available on Github. This study shows up to 93 % of compatibility opportunities to load the models according to different versions of UML. However, the current DSL theory prevents such compatibility by being based on nominal typing: a model is conformed to a unique named language. We proposed a new model typing theory based on structural typing to overcome this flaw [46]. With this theory a language can load any model which structure is compatible with the language. The second flaw concerns language reuse and customizations. When engineering new DSLs, it is likely that efforts spent on the development of prior languages could be leveraged, especially when their domains overlap. The current language workbenches [56], i.e., ‘*a unified environment to assist both language designers and users in, respectively, engineering new DSLs and using them*’ [44], lack an explicit support for language customizations and reuse. We proposed a new language workbench, called *Melange* [45]. Based on the model typing theory we also proposed, *Melange* allows language designers in building DSLs by safely assembling and customising legacy DSLs artefacts.

We also proposed an approach to manage DSL variants easier, i.e., different versions of a DSL adapted to specific purposes but that still share commonalities [86, 87]. When facing DSL variants, the challenge for language designers is to reuse, as much as possible, existing language constructs to narrow implementation from scratch. We thus proposed a reverse-engineering technique that analyses a set of DSL variants to identify language modules and their relations. A language product line can then be derived to customise and build new DSL variants.

Finally, we worked on easing the production and maintenance of DSL documentation [74]. Documenting DSLs and maintaining documentation over changes is a tedious task for software engineers. We propose a slicing-based technique to semi-automatically generate end-user documentation for a given DSL. The process uses as input data, the grammar, several model examples and the metamodel of the DSL. Then, each concept of the language is automatically documented by extracting relevant information from these different input data.

1.4 Research Methods

The research method I followed to develop the contributions summarised in the previous section is deductive and analytical. Deductive in the sense that I started from the thesis of this habilitation to adapt HCI and software engineering theories, develop engineering techniques, and conduct analytical studies to discuss the thesis. Analytical in the sense that my research aims at understanding the current UI and software engineering practices to then help software engineers with new theories and techniques. I thus conducted various empirical studies.

The research method I followed is also based on scientific collaborations. In 2017 I co-funded the GL-IHM French working group that aims at easing the collaborations between HCI researchers (resp. SE researchers) that have an interest in SE (resp. in HCI). CNRS is funding most of the actions of this group. An example of work we are conducting in 2019 in this group is the presentation of early-drafted UI experiments by researchers to get feedback.

Table 1.1: Details of the main empirical studies we conducted

| Articles | Description | Quantitative | Qualitative |
|----------|---|-------------------------------------|--------------------|
| [74] | Exercises with 17 subjects on two tools | Time, correctness | Anonymous feedback |
| [74] | Interviews of tool developers | | Feedback |
| [26] | Empirical study on UI listener code | Fault- change-proneness | |
| [26] | Tool evaluation | Recall, precision, patch acceptance | Interviews |
| [46] | Empirical study on UML models | Model conformance checking | |
| [76] | Empirical study on existing GUI bugs | Bug classification | |
| [28] | Exercises with 32 subjects on two tools | Time, correctness | |
| [19] | Experiments on 75 product matrices from Wikipedia | Precision | |

Over the years, the papers I wrote increasingly detail empirical studies for various purposes:

- evaluate the theories and techniques I proposed;
- get information about the current engineering practices;
- identify limits and then characterise scientific problems.

Table 1.1 summarises the main empirical studies I (co-)conducted. For each paper, I now follow the same schema that I adapt according to the research context:

1. **From personal insights, I try to find empirical evidences that motivate a research problem.** I think that researchers should not propose a scientific solution of a problem they just invented. The goal of such empirical evidences is to overcome this issue by: explaining, illustrating and assessing the existence of a research problem empirically. This threefold process gives credits to a scientific problem and allows me to update my initial vision of a problem.
2. **Evaluate empirically an approach I propose.** Researcher papers (vision papers excluded) have to describe a validation that assesses a proposed approach. Depending on the nature of the contribution, a validation can be a relevant use case or an empirical study. Both should be based on empirical artefacts (e.g., a use case provided by industrial partners or a comparison of tools on representative data). Researchers wrote insightful guides for helping other researchers conducting SE empirical studies [120].

To illustrate this schema, I detail in the next sub-sections a reduced version of the empirical studies I conducted for one of my representative papers: *'User Interface Design Smell: Automatic Detection and Refactoring of Blob Listeners'* [26]. This paper studies UI listeners (aka. UI handlers/callbacks): a method automatically called when the user triggers an event while interacting with a UI. A UI listener usually registers with a single widget to produce one UI command. UI toolkits provide predefined sets of listener interfaces or classes that developers can use to listen specific UI events such as clicks or key pressures.

1.4.1 From insights to empirical evidences: the example of a study on UI listeners

```

1 public void actionPerformed(ActionEvent e) {
2     Object src = e.getSource();
3     if(src==b1){
4         // Command 1
5     }else if(src==b2){
6         // Command 2
7     }else if(src instanceof AbstractButton &&
8         ((AbstractButton)src).getActionCommand().equals(
9             m3.getActionCommand())){
10        // Command 3
11    }}

```

Listing 1.1: Code example of a UI listener that manages three buttons.

I develop UIs using various programming languages and UI toolkits (mainly Angular and JavaFX). I observed in existing code that some developers may use a UI listener to manage several UI commands. Listing 1.1 illustrates this practice where the UI listener `actionPerformed` manages three widgets to produce three UI commands. I consider this coding practice as a bad one.

From this insight, I wanted to objectively state whether the number of UI commands that UI listeners can produce has an effect on the code quality of these listeners.

Research Questions. I thus conducted the following empirical study that relies on three research questions:

- RQ1** To what extent the number of UI commands per UI listeners has an impact on fault-proneness of the UI listener code?
- RQ2** To what extent the number of UI commands per UI listeners has an impact on change-proneness of the UI listener code?
- RQ3** Do developers agree that a threshold value, i.e., a specific number of UI commands per UI listener, that can characterize a UI design smell exist?

To answer these three research questions, we measured the following independent and dependent variables. We implemented the UI code analysing algorithms in *InspectorGidget*, an open-source Eclipse plug-in that analyses Java Swing, SWT and JavaFX software systems. All the material of the experiments is freely available.¹

Independent Variables. The independent variable of the study is the *Number of UI Commands* (CMD). This variable measures the number of UI commands a UI listener can produce. *InspectorGidget* will compute this variable.

Dependent Variables. *Average Commits* (COMMIT). This variable measures the average number of commits of UI listeners. This variable will permit to evaluate the change-proneness of UI listeners. To measure this variable, we automatically count the number of the commits that concern each UI listener.

Average fault Fixes (FIX). This variable measures the average number of fault fixes of UI listeners. This variable will permit to evaluate the fault-proneness of UI listeners. To measure this variable, we manually analyse the log of the commits that concern each UI listener. We manually count the commits which log refers to a fault fix, i.e., logs that point to a bug report of an issue-tracking system (using a bug ID or a URL) or that contain the term ‘fix’ (or a synonymous). We use the following list of terms to identify a first list of commits: *fix, bug, error, problem, work, issue, ticket, close, reopen, exception, crash, NPE, IAE, correct, patch, repair, rip, maintain, warning*. We then manually scrutinised each of these commits.

Both COMMIT and FIX rely on the ability to get the commits that concern a given UI listener. For each software system, we use all the commits of their history as the time-frame of the analysis. We ignore the first commit as it corresponds to the creation of the project.

The size, i.e., the number of lines of code (LoC), of UI listeners may have an impact on the number of commits and fault fixes. So, we need to compare UI listeners that have a similar size by computing the four quartiles of the size distribution of the UI listeners [1]. We kept the fourth quartile (Q_4) as the single quartile that contains enough listeners with different numbers of commands to conduct the study. This quartile Q_4 contains 297 UI listeners that have more than 10 LoCs. For the study the code has been formatted and the blank lines and comments have been removed.

Commits may change the position of UI listeners in the code (by adding or removing LoCs). To get the exact position of a UI listener while studying its change history, we use the Git tool *git-log*.² We then manually check the logs for errors.

¹<https://github.com/diverse-project/InspectorGidget>

²<https://git-scm.com/docs/git-log>

Table 1.2: The four selected software systems and their characteristics

| Software | Version | Toolkit | kLoCs | # commits | # UI listeners | Source repository link |
|---------------------------|---------|---------|-------|-----------|----------------|--|
| platform.ui .workbench | 4.7 | SWT | 143 | 10 049 | 259 | git.eclipse.org/c/gerrit/platform/eclipse. platform.git |
| JabRef | 3.8.0 | Swing | 95 | 8567 | 486 | github.com/JabRef/jabref |
| ArgoUML | 0.35.1 | Swing | 101 | 10 098 | 214 | github.com/rastaman/argouml-maven argouml.tigris.org/source/browse/argouml |
| FreeCol | 0.11.6 | Swing | 118 | 12 330 | 223 | sourceforge.net/p/freecol/git/ci/master/tree |

Objects. The objects of this study are open-source software systems. The dependent variables impose several constraints on the selection of these software systems. The systems must use an issue-tracking system and the Git version control system. We focused on software systems that have more than 5000 commits in their change history to let the analysis of the commits relevant. In this work, we focused on Java Swing and SWT UIs because of the popularity and the large quantity of Java Swing and SWT legacy code available on code repositories such as *Github* and *Sourceforge*. We thus selected four Java Swing and SWT software systems, namely ArgoUML, JabRef, Eclipse (more precisely the *platform.ui.workbench* plug-in) and FreeCol. Table 1.2 lists these software systems, the version used, their UI toolkit, their number of Java line of codes, commits and UI listeners, and the link the their source code. The number of UI listeners excludes empty listeners.

Results. We used the quartile Q_4 to compare listeners with similar sizes. Q_4 has the following distribution. 69.36 % (i.e., 206) of the listeners can produce one command (we will call them one-command listeners). 30.64 % of the listeners can be produce two or more commands: 47 listeners can produce two commands; 16 listeners can produce three commands; 28 listeners can produce at least four commands. To obtain representative data results, we considered in the following analyses three categories of listeners: *one-command listener* (CMD=1 in Table 1.3), *two-command listener* (CMD=2), *three+-command listener* (CMD>=3).

Table 1.3: Means, correlations and Cohen's d of the results

| Dependent variables | Mean CMD=1 | Mean CMD=2 | Mean CMD>=3 | Correlation (significance) | Cohen's d CMD=1 vs CMD=2 (significance) | Cohen's d CMD=2 vs CMD>=3 (significance) | Cohen's d CMD=1 vs CMD>=3 (significance) |
|---------------------|---------------|---------------|----------------|-------------------------------|--|---|---|
| <i>FIX</i> | 1.107 | 1.149 | 2.864 | 0.4281 (***) | 0.0301 (no) | 0.5751 (no) | 0.8148 (***) |
| <i>COMMIT</i> | 5.854 | 6.872 | 10.273 | 0.3491 (***) | 0.1746 (no) | 0.3096 (no) | 0.5323 (no) |

We computed the means of *FIX* and *COMMIT* for each of these three categories. To compare the effect size of the means (i.e., *CMD=1 vs. CMD=2*, *CMD=1 vs CMD=2*, and *CMD=1 vs. CMD>=3*) we used the Cohen's d index [119]. Because we compared multiple means, we used the Bonferroni-Dunn test [119] to adapt the confidence level we initially defined at 95 % (i.e., $\alpha = 0.05$): we divided this α level by the number of comparisons (3), leading to $\alpha = 0.017$. We used the following code scheme to report the significance of the computed p -value: No

significance= $p > 0.017$, $*$ = $p \leq .0017$, $**$ = $p \leq .005$, $***$ = $p \leq .001$. Because *FIX* (resp. *COMMIT*) and *CMD* follow a linear relationship, we used the Pearson's correlation coefficient to assess the correlation between the number of fault fixes (resp. number of changes) and the number of UI commands in UI listeners [119]. The correlation is computed on the quartile Q_4 . The results of the analysis are detailed in Table 1.3.

Figure 1.1 depicts the evolution of *FIX* over *CMD*. We observe a significant increase of the fault fixes when $CMD \geq 3$. According to the Cohen's d test, this increase is large (0.8148). *FIX* increases over *CMD* with a moderate correlation (0.4281, if in $[0.3, 0.7]$, a correlation is considered to be moderate [119]).

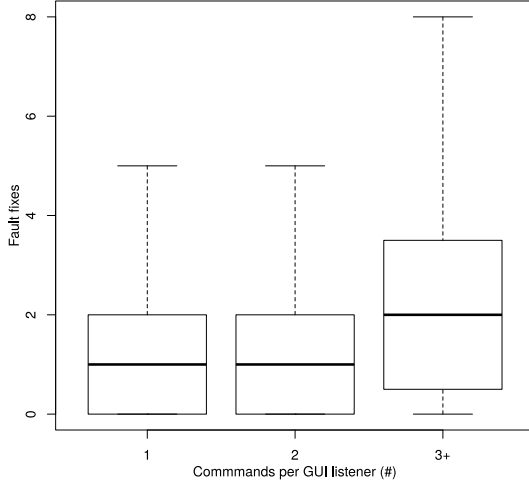


Figure 1.1: Number of fault fixes of UI listeners

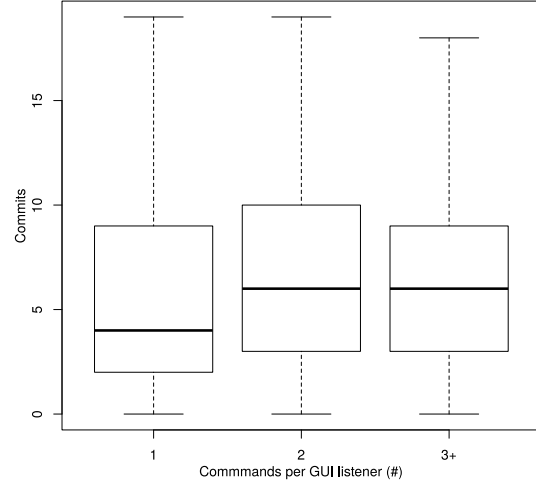


Figure 1.2: Number of commits of UI listeners

Regarding **RQ1**, on the basis of these results we can conclude that managing several UI commands per UI listener has a negative impact on the fault-proneness of the UI listener code: a significant increase appears at three commands per listener, compared to one-command listeners. There is a moderate correlation between the number of commands per UI listener and the fault-proneness.

Figure 1.2 depicts the evolution of *COMMIT* over *CMD*. The mean value of *COMMIT* increases over *CMD* with a weak correlation (0.3491, using the Pearson's correlation coefficient). A medium (Cohen's d of 0.5323) but not significant (p -value of 0.0564) increase of *COMMIT* can be observed between one-command and three+-command listeners. *COMMIT* increases over *CMD* with a moderate correlation (0.3491). Regarding **RQ2**, on the basis of these results we can conclude that managing several UI commands per UI listener has a small but not significant negative impact on the change-proneness of the UI listener code. There is a moderate correlation between the number of commands per UI listener and the change-proneness.

Regarding **RQ3**, we observe a significant increase of the fault fixes for three+-command listeners against one-command listeners. We also observe an increase of the commits for three+-command listeners against one-command listeners. We thus state that a threshold value, i.e., a specific number of UI commands per UI listener, that characterizes a UI design

smell exists. Note that since the COMMIT metrics counts all the commits, bug-fix commits included, the increase of the commits may be correlated to the increase of the fault fixes for three+-command listeners. We contacted developers of the analysed software systems to get feedback about a threshold value. Beyond the *'sounds good'* for three commands per listener, one developer explained that *'strictly speaking I would say, more than one or two are definitely an indicator. However, setting the threshold to low [lower than three commands per listener] could lead to many false positives'*. Another developer said *'more than one [command per listener] could be used as threshold, but generalizing this is probably not possible'*. We agree and define the threshold to three UI commands per UI listener. Of course, this threshold value is an indication and as any design smell it may vary depending on the context.

Conclusion. Based on the results of the empirical study previously detailed, we showed that a significant increase of the fault fixes and changes for two- and three+-command listeners is observed. Considering the feedback from developers of the analysed software systems, we define at three commands per listener the threshold value from which a design smell, we called *Blob listener*, appears. We define the *Blob listener* as follows. A *Blob listener* is a UI listener that can produce several UI commands. *Blob listeners* can produce several commands because of the multiple interactive objects they have to manage. In such a case, *Blob listeners'* methods (such as *actionPerformed*) may be composed of a succession of conditional statements that: 1) identify the interactive object that produced the UI event to treat; 2) execute the corresponding UI command. The **threats to validity** are summarised at the end of the next section.

1.4.2 Validating approaches empirically: the example of the UI listener refactoring tool

Research Questions. We implemented in `InspectorGidget` features for detecting and refactoring *Blob listeners*. To evaluate the efficiency of our detection and refactoring process, we addressed the four following research questions:

- RQ4** To what extent is the detection algorithm able to detect UI commands in UI listeners correctly?
- RQ5** To what extent is the detection algorithm able to detect *Blob listeners* correctly?
- RQ6** To what extent does the refactoring process propose correct refactoring solutions?
- RQ7** To what extent the concept of *Blob listener* and the refactoring solution we propose are relevant?

Objects. The objects we used in this evaluation are the four large open-source software systems we used in the previous study.

Methodology. The accuracy of the static analyses that compose the detection algorithm is measured by the *recall* and *precision* metrics. We ran `InspectorGidget` on each software system to detect UI listeners, commands and *Blob listeners*. We assumed as a precondition that only UI listeners are correctly identified by our tool. Thus, to measure the precision and recall of our automated approach, we manually analysed all the UI listeners detected by `InspectorGidget` to:

Check commands. We manually analysed each UI listeners to state whether the UI commands they contain are correctly identified. The *recall* measures the percentage of correct UI commands that are detected (Equation (1.1)). The *precision* measures the percentage of detected UI commands that are correct (Equation (1.2)). For 39 listeners, we were not able to identify the commands of UI listeners. We removed these listeners from the data set.

$$recall_{cmd}(\%) = \frac{|\{correctCmds\} \cap \{detectedCmds\}|}{|\{correctCmds\}|} \times 100 \quad (1.1)$$

$$precision_{cmd}(\%) = \frac{|\{correctCmds\} \cap \{detectedCmds\}|}{|\{detectedCmds\}|} \times 100 \quad (1.2)$$

The *correctCmds* variable corresponds to all the commands defined in UI listeners, i.e., the commands that should be detected by `InspectorGidget`. The *recall* and *precision* are calculated over the number of false positives (FP) and false negatives (FN). A UI command incorrectly detected by `InspectorGidget` is classified as false positive. A false negative is a UI command not detected by `InspectorGidget`.

Check Blob Listeners. This analysis directly stems from the UI command one since we manually checked whether the detected *Blob listeners* are correct with the threshold value of three commands per UI listener. We used the same metrics used for the UI command detection to measure the accuracy of the *Blob listeners* detection:

$$recall_{blob}(\%) = \frac{|\{correctBlobs\} \cap \{detectedBlobs\}|}{|\{correctBlobs\}|} \times 100 \quad (1.3)$$

$$precision_{blob}(\%) = \frac{|\{correctBlobs\} \cap \{detectedBlobs\}|}{|\{detectedBlobs\}|} \times 100 \quad (1.4)$$

Results. *RQ4: Command Detection Accuracy.* Table 1.4 shows the number of commands successfully detected per software system. `InspectorGidget` detected 1392 of the 1400 UI commands (eight false negatives), leading to a recall of 99.43%. `InspectorGidget` also detected 62 irrelevant commands, leading to a precision of 95.73%. The FP instances are detailed in the original paper [26]. To conclude on RQ4, our approach is efficient for detecting UI commands that compose UI listener, even if improvements still possible.

Table 1.4: Command detection results

| Software System | Successfully Detected Commands (#) | FN (#) | FP (#) | Recall _{cmd} (%) | Precision _{cmd} (%) |
|-----------------|------------------------------------|--------|--------|---------------------------|------------------------------|
| Eclipse | 330 | 0 | 5 | 100 | 98.51 |
| JabRef | 510 | 5 | 7 | 99.03 | 98.65 |
| ArgoUML | 264 | 3 | 3 | 98.88 | 98.88 |
| FreeCol | 288 | 0 | 47 | 100 | 85.93 |
| OVERALL | 1392 | 8 | 62 | 99.43 | 95.73 |

RQ5: Blob Listeners Detection Accuracy. To validate that the refactoring is behaviour-preserving, the refactored software systems have been manually tested by their developers we contacted and ourselves. Test suites of each system have also been used. Table 1.5 gives an overview of the results of the *Blob listeners* detection per software system. 12 false positives and one false negative have been identified against 52 *Blob listeners* correctly detected (true positive – TP). The average recall is 98.11% and the average precision is 81.25%. The average time

(computed on five executions for each software system) spent to analyse the software systems is 5.9 s. It excludes the time that *Spoon* takes to load all the classes, that is an average of 22.4 s per software system. We did not consider the time that *Spoon* takes since it is independent of our approach.

Table 1.5: *Blob listener* detection results

| Software System | TP (#) | FN (#) | FP (#) | Recall blob (%) | Precision blob (%) | Time (ms) |
|-----------------|-----------|----------|-----------|-----------------|--------------------|------------|
| Eclipse | 16 | 0 | 2 | 100 | 88.89 | 4 |
| JabRef | 8 | 0 | 3 | 100 | 72.73 | 5.6 |
| ArgoUML | 13 | 1 | 2 | 92.86 | 86.7 | 8.6 |
| FreeCol | 15 | 0 | 5 | 100 | 75 | 5.5 |
| OVERALL | 52 | 1 | 12 | 98.11 | 81.25 | 5.9 |

Table 1.6: *Blob listener* refactoring results

| Software System | Success (#) | Failure (#) | Precision refact (%) | Time (s) |
|-----------------|-------------|-------------|----------------------|------------|
| Eclipse | 4 | 12 | 25 | 133 |
| JabRef | 4 | 4 | 50 | 236 |
| ArgoUML | 11 | 3 | 78.57 | 116 |
| FreeCol | 7 | 8 | 46.7 | 135 |
| OVERALL | 26 | 27 | 49.06 | 155 |

The FP and FN *Blob listeners* is directly linked to the FP and FN of the commands detection. For example, FP commands increased the number of commands in their listener to two or more so that such a listener is wrongly considered as a *Blob listener*. This is the case for *FreeCol* where 47 FP commands led to 5 FP *Blob listeners*.

To conclude on RQ5, regarding the recall and the precision, our approach is efficient for detecting *Blob listeners*.

RQ6: Blob Listeners Refactoring Accuracy. This research question aims to provide quantitative results regarding the refactoring of *Blob listeners*. The results of *InspectorGadget* on the four software systems are described in Table 1.6. The average refactoring time (i.e., five executions for each software system) is 155 s. The average rate of *Blob listeners* successfully refactored is 55.1%. 27 of the 49 *Blob listeners* have been refactored. Two main reasons explain this result: 1/ There exists in fact two types of *Blob listeners* and our refactoring solution supports one of them; 2/ The second type of *Blob listeners* may not be refactorable because of limitations of the Java GUI toolkits. To conclude on RQ6, the refactoring solution we propose is efficient for one of the two types of *Blob listeners*. Refactoring the second type of *Blob listeners* may not be possible and depends on the targeted GUI toolkit.

RQ7: Relevance of the Blob listener. This last research question aims to provide qualitative results regarding the refactoring of *Blob listeners*. We submitted patches that remove the found and refactorable *Blob listeners* from the analysed software systems. We then asked their developers for feedback regarding the patches and the concept of *Blob listener*. The bug reports that contain the patches, the commits that remove *Blob listeners*, and the discussions are listed in Table 1.7. The patches submitted to *Jabref* and *FreeCol* have accepted and merged. The patches for *Eclipse* are not yet merged but were positively commented. We did not receive any comment regarding the patches for *ArgoUML*. We noticed that *ArgoUML* is no more actively maintained.

We asked developers whether they consider that coding UI listeners that manage several interactive objects is a bad coding practice. The developers that responded globally agree that *Blob listener* is a design smell. ‘It does not strictly violate the MVC pattern. [...] Overall, I like your solution’. ‘Probably yes, it depends, and in examples you’ve patched this was definitely a mess’. An *Eclipse* developer suggest to complete the *Eclipse* UI development documentation to add information related to UI design smells and *Blob listener*.

Table 1.7: Commits and discussions

| Software System | Bug reports, commits and discussions |
|-----------------------|--|
| Eclipse (platform.ui) | bugs.eclipse.org/bugs/show_bug.cgi?id=510745 dev.eclipse.org/mhonarc/lists/platform-ui-dev/msg07651.html |
| JabRef | github.com/JabRef/jabref/pull/2369 github.com/JabRef/jabref/commit/021f094e64a6393a7490ee680d73ef26b3128625 |
| ArgoUML | http://argouml.tigris.org/issues/show_bug.cgi?id=6524 |
| FreeCol | sourceforge.net/p/freecol/mailman/message/35566820 sourceforge.net/p/freecol/git/ci/669cf9c74b208c141cea27ee254292b3422d5718 sourceforge.net/p/freecol/git/ci/2865215d3712a8d4d4bd958c1b397c90460192da sourceforge.net/p/freecol/git/ci/cdc689c7ae4bbac9fcc729477d5cc7e40ac4a90b sourceforge.net/p/freecol/git/ci/0eedd71b269b6cf20ec00f0fc5a7da932ceab4f sourceforge.net/p/freecol/git/ci/973422623b52289481f328b27f12543a4b383f38 sourceforge.net/p/freecol/git/ci/985adc4de11ccd33648e99294e5d91319cb23aa0 sourceforge.net/p/freecol/git/ci/4fe44e747cb30a161d8657750aa75b6c57ea30ab |

Regarding the relevance of the refactoring solution: *‘I like it when the code for defining a UI element and the code for interacting with it are close together. So hauling code out of the action listener routine and into a lambda next to the point a button is defined is an obvious win for me.’* A developer, however, explained that *‘there might be situations where this can not be achieved fully, e.g. due to limiting implementations provided by the framework.’* *‘It depends, if you refactor it by introducing duplicated code, then this is not suitable and even worse as before’.*

To conclude on RQ7, the concept of *Blob listener* and the refactoring solution we propose is accepted by the developers we interviewed. The refactoring has a positive impact on the code quality. The interviews did not permit the identification of how *Blob listener* are introduced in the code.

Threats to validity. External validity threats concern the possibility to generalise our findings. We designed the experiments using multiple Java Swing and SWT open-source software systems to diversify the observations. These unrelated software systems are developed by different persons and cover various user interactions. We provide on the companion web page examples of *Blob listeners* in other Java UI toolkits, namely GWT and JavaFX.

Construct validity threats relate to the perceived overall validity of the experiments. We used `InspectorGidget` to find UI commands in the code. `InspectorGidget` might not have detected all the UI commands. We showed that its precision (95.7) and recall (99.49) limit this threat. Regarding the validation of `InspectorGidget`, the detection of FNs and FPs have required a manual analysis of all the UI listeners of the software systems. To limit errors during this manual analysis, we added a debugging feature in `InspectorGidget` for highlighting UI listeners in the code. We used this feature to browse all the UI listeners and identify their commands to state whether these listeners are *Blob listeners*. We also manually determined whether a listener is a *Blob listener*. To reduce this threat, we carefully inspected each UI command highlighted by our tool.

1.5 Projects and Supervision

We – all the permanent staff of the DiverSE research group and I – consider doctoral studies as a corner-stone of the DiverSE team and of research in general. Thinking about tomorrows research requires thinking about how to train PhD students to be top-level researchers and teachers. This is not an easy task, in particular because of the short duration of the doctoral studies in France (three years) that does not help in publishing in flagship conferences and journals.

Doctoral studies do not always lead to research positions. Software engineer is nowadays a highly prized resource. A consequent part of the PhD students of the team goes to industry as software or research engineers. We think that having PhD in the industry is crucial as doctoral studies develop specific working facets as the ability to step back and see the bigger picture, open-mindedness, pugnacity or resilience.

The work presented in this habilitation and the research lines I envision result from collaborations I have had with PhD students, post-docs and researcher colleagues. Table 1.8 gives the list of PhD I co-supervised. I co-supervised the PhD students with nine different researchers, two of them come from industry. This table provides the amount and type of supervision work I took care of, the co-supervisors, the period, the viva date, the funding and the topic.

Table 1.8: PhD co-supervisions from 2012 to 2019

| Name | Rate (%) | Role | With | Period | Viva | Funding | Topic |
|------------------------|----------|-----------------|----------------------------------|---------|----------|--------------------|-----------------------------|
| Valéria Lelli [75] | 80 | co-supervisor | Baudry | 2012-15 | 15-11-19 | BGLE2 Connexion | UI testing |
| Thomas Degueule [44] | 33 | co-supervisor | Barais Combemale | 2013-16 | 16-12-12 | ITEA2 MERgE | DSL engineering |
| Gwendal Le Moulec [72] | 33 | co-supervisor | Arnaldi Gouranton | 2015-18 | 18-09-26 | MESR grant | Virtual reality engineering |
| Youssou Ndiaye | 25 | co-supervisor | Barais Bouabdallah Aillery | 2016-19 | | CIFRE Orange | Web engineering |
| Romain Lebouc | 33 | main supervisor | Plouzeau Ribault | 2019-22 | | CIFRE KEREVAL | UI testing |

The funding of these PhD studies are heterogeneous. BGLE2 is a industry-driven national call. The BGLE2 *Connexion* project investigated methods and tools to: automatically analyse and compare regulatory requirements evolutions and geographical differences; automatically generate test cases for interactive systems in variable environments. I worked on this last point with different French industrial partners on the test of their GUIs. I assisted Baudry in writing the proposal that concerned Inria Rennes.

MESR is a national research grant. Arnaldi, Gouranton and I decided to work together through this project on the intersection between software engineering and virtual reality.

CIFRE is a French industrial program for conducting a research program with a given company on an innovative topic. We are working with the French company Orange with this program on the topic of Web engineering. I co-wrote the proposal with my co-supervisors.

Through this program we are also working with KEREVAL, a French company that develops software testing services. I wrote most of the proposal. I am the main supervisor of this thesis.

Writing project proposals is an ungrateful, time-consuming yet necessary job to get funding. The projects detailed above are the tip of the iceberg. In the DiverSE team, we monitor project calls and write various proposals that are not accepted and hardly recyclable.

The next sub-sections detail the research work of the PhD students that defended and their current position.

Valéria Lelli – Testing and maintenance of graphical user interfaces

I supervised Lelli at 80 %. We worked on testing and maintaining UIs in the context of the CONNEXION project. Researchers develop software testing techniques to find errors in code. They also assess software quality criteria and measurement techniques to detect error-prone code. In this thesis, we argued that researchers must pay the same attention on the quality and reliability of UIs, from a software engineering point of view.

We specifically made two contributions on this topic. First, UIs can be affected by errors that stem from development mistakes. The first contribution of this thesis is a fault model that identifies and classifies GUI faults. We have shown that GUI faults are diverse and imply different testing techniques to be detected. As for the second contribution, we focus on design smells that can affect UIs specifically. Like any code artefact GUI code developers should use static code analyses to detect implementation defects and design smells. We identify and characterise a new type of design smell, called *Blob listener*. It occurs when a UI listener, that gathers and process UI events, can produce more than one command. We propose a systematic static code analysis procedure that searches for *Blob listener* instances that we implement in a tool called *InspectorGidget*. Experiments we conducted exhibited positive results regarding the ability of *InspectorGidget* in detecting *Blob listeners*. To counteract the use of *Blob listeners*, we propose good coding practices regarding the development of UI listeners.

This PhD is the first work I conducted on UI testing. With Baudry we had the feeling that UIs were under-tested. At this time, UI testing was not trendy as it is the case now. This PhD allowed me to envision how UIs should be tested and what are the current limits. The PhD thesis with Lebouc that started this year stems from this vision.

Lelli holds an associate professor position at the Federal University of Ceará, Brazil.

Thomas Degueule – Composition and interoperability for external domain-specific language engineering

This PhD is the single one not directly related to UI engineering. It focuses on software language engineering (SLE). Development and evolution of DSLs is becoming recurrent in the development of complex software systems. However, despite many advances in the software language engineering domain, DSLs and their tooling still suffer from substantial development costs which hamper their successful adoption in the industry. In this thesis we addressed two main challenges. First, the proliferation of independently developed and constantly evolving DSLs raises the problem of interoperability between similar languages and environments. Second, since DSLs and their environments suffer from high development costs, tools and methods must be provided to assist language designers and mitigate development costs. To address these challenges, we first propose the notion of language interface. Using language interfaces, one can vary or evolve the implementation of a DSL while retaining the compatibility with the services and environments defined on its interface. Then, we present a mechanism, named model polymorphism, for manipulating models through different language interfaces. Finally, we propose a meta-language that enables language designers to reuse legacy DSLs, compose them, extend them, and customize them to meet new requirements. We implement all our contributions in a new language workbench named *Melange* that supports the modular definition of DSLs and the interoperability of their tooling. We evaluate the ability of *Melange* to solve challenging SLE scenarios.

Since I joined the DiverSE group in 2009 (called Triskell at this period), I devoted a consequent part of my research time to SE and SLE. I consider that working on UI engineering requires in certain cases to step back and improve underlying SE principles: it is not possible to adapt limited SE techniques for engineering UIs. This was the case during this PhD. We proposed new fundamental principles (e.g., language interface, model polymorphism) that should improve the usability of SLE development processes. Based on these new principles we can envision new ways of engineering DSL UIs.

In late 2019, Degueule will hold a full time researcher position at CNRS, France. He is currently a postdoctoral researcher at CWI, Netherlands.

Gwendal Le Moulec – Model-driven synthesis of virtual reality applications

This third PhD was co-supervised with local colleagues from the Virtual Reality (VR) domain. VR applications have a specific and complex kind of post-WIMP UI that thus requires specific engineering techniques. Development practices in VR, however, are far to follow SE good practices. This dearth of SE practices may hinder the industrial advent of VR in the next years. For example, each company uses their own development methods and code reuse is not a concern that VR engineers consider [48]. Those lacks of reuse and abstraction are known problems in MDE, which proposes the Software Product Line (SPL) concept to automate the production of software belonging to the same family, by reusing common components. However, this approach is not adapted to software based on a scenario, like in VR.

This PhD thesis focused on the automated development and evaluation of VR software systems with the use of MDE and SPL techniques. We propose two frameworks that respectively address the lacks in MDE and VR: SOSPL (scenario-oriented software product line) and VRSPL (VR SPL). SOSPL is based on a scenario model that handles a software variability model (feature model, FM). Each scenario step matches a configuration of the FM. VRSPL is based on SOSPL. The scenario manages virtual objects manipulation, the objects being generated automatically from a model.

We implemented these frameworks inside tools we tested and evaluated on use cases. The results promote the use of these frameworks for producing scenario-based software.

This PhD was crucial for me: it tackled the engineering of a next generation of UIs that will be widespread in several years with the advent of VR. The benefit of this PhD is twofold. First, we adapted SE techniques to another domain. We can consider this point as an assessment of the adapted techniques. Second, we proposed techniques to engineer modern and complex UIs, which is in line with the UI engineering definition.

Le Moulec holds a software engineer position in a French company.

1.6 Software Development

Following the research methods I follow and detailed in Section 1.4, I (co-)developed several software systems. The development of open-source and freely available engineering tools is a crucial point in my research. Table 1.9 summarises the main tools I (co-)developed, their topics, implementation and website. I then describe the main ones.

Table 1.9: The main tools I (co-)developed).

| Topic | Sub-topic | Implementation | Website |
|-----------------|---|--------------------|---|
| UI testing | Model-based testing | Java | github.com/arnobl/Malai/tree/master/malai-mde |
| UI code quality | Code analysis and refactoring | Java | github.com/diverse-project/InspectorGidget |
| MDE | Model slicing | Java, Scala, XText | github.com/arnobl/kompren |
| SLE | model slicing, visualisation, interactivity | Java | github.com/arnobl/kompren/tree/master/expen |
| SLE | language workbench | Java, Xtext, tools | http://melange.inria.fr |
| SLE | documentation generation | Java | github.com/arnobl/comlanDocywood |

Model slicing with Kompren

Kompren is a DSL that implements the model slicing technique detailed in Section 1.3. Kompren has an Eclipse editor that provides auto-completion, error detection and an on-the-fly compilation of the model slicer into a Java library. I developed all the Java and Scala code of Kompren (around 22 kLoCs). We used Kompren in international collaborations on different topics [28, 124] and is now used within the Melange tool.

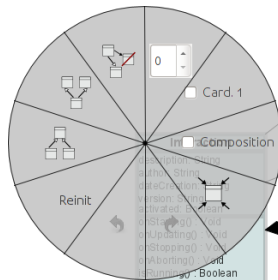


Figure 1.3: The buttons of this pie menu call the slicer with different parameters for filtering-out metamodel elements.

I illustrate *Kompren* with the use case of metamodel visualisation [28]. When seeking to understand a large model, model slicing can help in extracting the sub-part of interest from the model. For instance, language designers can use model slicers to build interactive visualisation features for large metamodels. Such features are dynamic filters that show on demand the inheritance or the composition tree of a targeted class by hiding the other elements. Figure 1.3 depicts such features where the buttons of the pie menu call this slicer with different parameters. Listing 1.2 specifies the model slicer used in Figure 1.3. The principle of the metamodel viewer of Figure 1.3 is that a user can click on a class of the visualised metamodel to then select a filter. The slicer of Listing 1.2 defines three different filters: a filter for slicing the super inheritance tree of a class; another one for the lower inheritance tree of a class; a last one for pruning the metamodel to show only the classes and relations linked to the targeted class. A Java library is generated from the model slicer. Language designers can then integrate this library into a tool, such as a metamodel viewer.

```

1 slicer MetamodelVisualizationSlicer {
2   domain: "platform:/plugin/org.eclipse.emf.ecore/model/Ecore.genmodel"
3   input:.ecore.EClass
4   radius:.ecore.EClass
5   slicedClass:.ecore.ENamedElement
6   slicedClass:.ecore.EStructuralFeature feat
7     constraint: card1 [[ feat.lowerBound>0 ]]
8   slicedProperty:.ecore.EClass.eSuperTypes option
9   slicedProperty:.ecore.EClass.eSuperTypes option opposite(lowerTypes)
10  slicedProperty:.ecore.EReference.containment
11  slicedProperty:.ecore.ETypedElement.lowerBound option
12  slicedProperty:.ecore.ETypedElement.upperBound option
13 }
```

Listing 1.2: A *Kompren* model slicer used for defining interactive visualization features

Assembling new DSLs with Melange

Melange is a language workbench that provides a modular approach for customising, assembling and integrating multiple DSL specifications and implementations. Melange helps to manage variability within language specifications (syntactic and semantic variation points), and reuse pieces of syntax and semantics from one DSL to another. The language workbench embeds a model-oriented type system that provides model polymorphism and language substitutability, i.e., the possibility to: manipulate a model through different interfaces; define generic transformations that can be invoked on models written using different DSLs. Melange also provides a dedicated meta-language where models are first-class citizens and languages are used to instantiate and manipulate them. By analogy with the class-based, object-oriented paradigm, we can classify Melange as a language-based, model-oriented language.

Melange is tightly integrated within the *Eclipse Modeling Framework* ecosystem. I participated in the development and design of Melange that is mainly developed by Degueule.

Listing 1.3 is a Melange model. It imports the abstract syntax (the metamodel) of two DSLs, defined as languages *L1* and *L2* (Lines 1 and 4). This model then uses these two languages to build abstract syntaxes of new DSLs. Line 7 defines a new language *L3* by merging *L1* and *L2*. Line 10 defines a new language *L4* by slicing *L1* using the attribute *A.a*.

```

1 language L1 {
2   syntax "MM1.ecore"
3 }
4 language L2 {
5   syntax "MM2.ecore"
6 }
7 language L3 inherits L1 {
8   merge L2
9 }
10 language L4 {
11   slice+ L1 using [MM1.A.a]
12 }

```

Listing 1.3: A simple Melange model.

Analysing UI code with InspectorGidget

InspectorGidget is an open-source tool composed of 12 kLoCs (Java) I developed with Lelli to analyse UI code. InspectorGidget can analyse UI Java code (JavaFX, SWT, Swing, Android). It first extracts UI information from code. This step is complex as UI code is usually intertwined with the rest of the application code. InspectorGidget extracts information related to the widgets, UI listeners, UI commands to build a high-level model. We can view this step as reverse engineering.

Then, from the extracted data we developed a UI code analysis and refactoring tool that spots and fixes bad smells [26]. We aim at developing new tools based on InspectorGidget, such a UI code coverage tool or a UI test generator. Figure 1.4 is an example of UI code refactored with InspectorGidget. The code on the left is affected by the *Blob listener* code smell as discussed in Sections 1.3 and 1.4. InspectorGidget refactored this code to produce the new code on the right.

| | | |
|--|--|--|
| <pre> class B implements ActionListener { JButton but1; JButton but2; B() { but1 = new JButton(text: "button1"); but1.setActionCommand("BUTTON1_ACTION_CMD"); but1.addActionListener(this); but2 = new JButton(text: "button2"); but2.setActionCommand("BUTTON2_ACTION_CMD"); but2.addActionListener(this); } @Override public void actionPerformed(final ActionEvent e) { if(e.getActionCommand().equals(anObject: "BUTTON1_ACTION_CMD")) { System.out.println(x: "Command 1"); return; } if(e.getActionCommand().equals(anObject: "BUTTON2_ACTION_CMD")) { System.out.println(x: "Command 2"); return; } } } </pre> | <pre> 7 7 8 8 9 9 10 10 11 11 12 12 13 13 14 14 15 15 16 16 17 17 18 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 </pre> | <pre> class B { JButton but1; JButton but2; B() { but1 = new JButton(text: "button1"); but1.addActionListener(e → System.out.println(x: "Command 1")); but2 = new JButton(text: "button2"); but2.addActionListener(e → System.out.println(x: "Command 2 ")); } } </pre> |
|--|--|--|

Figure 1.4: An example of automated UI code refactoring with InspectorGidget.

Chapter 2

Research Perspectives

In this habilitation I detailed how I envision UI engineering: a research domain that has to provide software engineers with theories and techniques at a correct level of abstraction to help them in producing usable modern UIs. Researchers must assess these theories and techniques *empirically*. I then detailed the contributions I, with the essential help of the persons I work(ed) with, made to this domain between 2010 and 2019.

I classified the detailed contributions into two branches. First, I detailed new dedicated abstractions for engineering UIs. Second, I detailed new techniques to ease the development and use of domain-specific languages, that are a specific form of user interfaces.

Research roadmap – Numerous UI engineering challenges still exist. For example: Vanderdonckt detailed in 2008 various challenges, that for most of them still exist, of using MDE for engineering UIs [128]; Paige motivated challenges of building interactive DSLs [106]. The next sections focus and detail three future UI engineering challenges I envision and that stem from the contributions detailed in this habilitation. As an illustration of the large scope of the UI engineering domain, these three challenges focus on very different topics.

The first perspective focuses on UI testing. UI validation is a long-standing domain [63] that has to follow and integrate the technological and usage changes that affect UIs. Such changes are for example the current advent of post-WIMP UIs in the industry such a virtual reality systems. Such systems go beyond the traditional WIMP concepts (window, icon, menu, pointer) with specific user interactions and interfaces [40]. Testing such UIs is a complex task since multiple UI testing concepts do not match post-WIMP UIs: what does test coverage mean for such UIs? Which UI oracles? Etc. UI validation is also board topic that includes human-factor evaluation, such as usability evaluation, and UI testing that focuses on automatic validation. The perspective I develop on UI testing in this section focuses on a concept increasingly used by major companies: DevOps, i.e., the ability to reduce the delay between a change in a software system and the patching of this change in production [49, 115]. One goal of DevOps is not to separate anymore the development process (the *Dev*) from the delivery process (the *Ops*) by promoting continuous delivery of a software system. Software testing is a corner-stone of DevOps as it assures the quality of the code changes [41]. DevOps from a UI perspective requires new UI testing techniques that I detail in Section 2.1.

The second perspective focuses on DSLs. In a near future engineers may create DSLs on a daily basis, as it is the case with web frameworks. The language engineering community works at reducing the development costs of DSLs. Engineers release a DSL with a set of associated tools, such as editors, compilers, simulators or linters. These tools have to be usable, even if they are developed and maintained on a daily basis. This means that UI engineering has to focus on DSL engineering: visualisation techniques or relevant user interactions are topics that researchers should tightly integrate within DSL development processes. I detail in Section 2.2 two research axes that go in this direction: easing the use of visualisation and interaction techniques for DSLs; analysing DSL usages.

The third perspective focuses on programming modern UIs. A major concept of programming UIs is the UI event processing: by interacting with a user interface, users trigger UI events that the software then gathers and processes. The current UI toolkits still use a technique proposed with SmallTalk and the *Model-View-Controller* (MVC) pattern in the 80s [69]: the UI event processing model, currently implemented using callback methods or reactive programming [10] libraries. This model considers low-level UI events as the first-class concept developers use for coding more and more complex user interactions. This model suffers from several major flaws that prevent the free development of modern user interactions. I detail in Section 2.3 a roadmap for designing a new *user interaction* processing model.

Educational challenges – As an Associate Professor at INSA Rennes, a high-level post-graduate French public school of engineering, I spend a **considerable** part of my working time teaching future engineers and interacting with local companies. I consider that conducting research activities goes beyond publishing articles, writing project proposals and interacting with other researchers. Having an impact on the education of future software engineers and thus on the industry is also a challenge I work on. I aim at providing them with **up-to-date** software knowledge that would help them in the uptake of the next software engineering evolution. Related to the perspectives I detail in this chapter, methinks that such changes may be for example:

- Software engineers cannot restrict their skills to the strict software engineering domain anymore. Producing software systems now – or in fact since a couple of years now – spans over multiple other computer science domains such as distributed systems, security, HCI and *UI engineering*.
- Software engineers may create computer languages on a daily basis in a near future, as it is currently the case with web frameworks. Software engineers must learn to learn languages and how to create *usable* languages, instead of simply learning programming languages.
- Software development processes are changing. First, engineers aim at reducing the time for releasing software systems. This requires engineers to have a strong background on software validation techniques, including UI validation. Second, software systems are less and less monolithic. The numerous devices and platforms on which software systems have to run make their development more and more complex. The term *software product line* is now used to characterise such new kinds of software systems;

- Software engineering is not an activity reserved to the audience of computer science engineers anymore. For example, electronics engineers have a substantial part of software engineers in their jobs. They have to: validate the code they embed on small devices; develop web applications; etc.

2.1 DevOps and user interfaces

2.1.1 Research Context

Modern software systems do not stop for maintenance or evolution any more. Facebook deploys a new version of their software systems every ten seconds without any perceptible stop. This run towards continuous delivery is possible thanks to the use of project management, release engineering, and software engineering tools and techniques: version control systems, dependency tools, building tools, testing frameworks, logging tools, deployment tools, virtualisation, team management techniques, *etc.* This trend, sometimes called DevOps [49, 115], aims at reducing the gap between changes in a software system and the deployment of these changes in production. To do so, the jobs of developer (the *Dev*) and tester, release engineer (the *Ops*) are no more separated. Highly automated processes are also deployed to ease the development and evolution of a software system. One of the benefits of DevOps is a significant reduction in the software release costs.

In this race for automation, the need to produce and maintain tests is one of the cornerstones to ensure the quality of continuously deployed software systems. This also concerns user interface tests that automatically ensure quality criteria of UIs. A recent study has shown that the success of mobile applications, which make significant use of front-ends, depends heavily on the quality of their UIs [81]. However, integrating UIs in DevOps processes is hampered by various technical and scientific obstacles. A UI is a specific software artefact. Testing UIs requires focusing on different concerns than for object-oriented code, as previously explained in this habilitation. Automating UI testing is also a challenge. We observed by discussing with local companies that they do not always script UI tests for automation. Having a DevOps approach for UIs thus requires new UI testing techniques to reduce human intervention during the development, maintenance and result analysis of UI tests.

2.1.2 Scientific Challenges

The global scientific challenge of this perspective is the improvement of UI testing techniques to be used in a devOps context. Achieving this global challenge requires addressing several scientific issues that concern the state of the art of UI testing practices. In this perspective I detail the following scientific issues:

- UI coverage is not developed as standard code coverage;
- non-deterministic, aka. flaky, UI tests prevent the automatic process of UI testing in a DevOps context;
- missing UI code analysing tools to assess the code quality of UIs.

No UI coverage. Code coverage is used to find code elements that are not or only partly covered by tests. As one of the pillars of the test process, the coverage calculation is used in DevOps to find weaknesses in the test suite. A UI test is an ordered sequence of user interactions, that can be viewed as a navigation path within the UI. A set of a UI tests correspond to a set of paths, called event-flow graph (EFG) [84]. The current coverage techniques focus on object-oriented constructs. However, measuring the coverage of a UI test suite requires to compute the EFG of a test suite. The goal is to find the relevant navigation paths not yet covered. This is a complex task as a UI notably consists of interactive elements, user interactions, UI commands. Measuring the UI test coverage is a complex task. It requires identifying in the code user interactions and UI commands to compute a fine-grained EFG. Figure 2.1 is an example of such a classical EFG. Events are considered as interactive objects (the four menus). The graph describes the transitions between these interactive objects. Such an EFG ignores major UI concerns, by: mixing interactive objects with the user interactions they provide; considering simple user interactions only; ignoring the underlying code that triggers the transition. Note that for this last point, research works started to make use of static analyses on UI code [7].

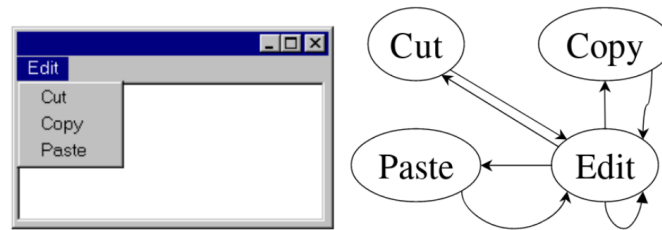


Figure 2.1: Example of a basic event-flow graph, from [134]. The UI on the left, composed of menus. The EFG on the right that describes the possible paths between these menus.

Not considering all these UI details in the computation of the UI coverage is a severe limit of considering UI testing in DevOps: developers have no detail about the quality of their UI test suite, while software quality is a corner-stone of DevOps. This also prevents test amplification [42], i.e., the automatic production of new tests from existing ones.

Flaky UI tests. The execution of UIs is based on multi-threading. One thread runs the UI in parallel of the application main thread. This raises various concurrency issues. For example, UI tests affected by bad coding practices that, for example, make use of `sleep` routines to deal with concurrency. A UI test, which runs in the main process, may have to wait until the UI task ends (e.g., the test simulates a user that interacts with the UI) to then executing the oracles. The following code example shows a UI test that makes use of `sleep` routines to wait for the end of a UI task to run the UI oracles.¹ A recent study shows that fixing code smells, such as the use of `sleep` routines in a test, fix up to 50% of the spotted flaky tests [110]. We think that code smells specific to UI testing exist and should be studied in this purpose, as we did with the *Blob Listener* UI code smell [26, 78].

¹<https://github.com/latexdraw/latexdraw/commit/c3bcb5d19a77a2481b0e5ca521b36abceaea3199#diff-2d13f6456312079bf6b0416d9b0f334aR104>

```
1 @Test
2 public void testIncludes() {
3     prefs.includesProperty().setValue("");
4     clickOn(setter.latexIncludes).write("foo").sleep(1000L);
5     assertEquals("foo", prefs.includesProperty().get());
6 }
```

Listing 2.1: A flaky UI test, gathered from an open-source project, that makes use of `sleep` routines

Other examples are the UI animations that may alter the execution of a UI test, and graphical drivers of the running platform that may change the results of a UI test oracle. Such tests are called flaky test [82, 20]: several executions of the same test intermittently produce different test results. In a DevOps context, it is necessary to have techniques to automatically find and correct flaky UI tests.

Missing UI code analysing tools. DevOps requires software validation and maintenance steps. In the previous points I discuss software testing techniques that permit testers to produce test scripts to be run against the system under test to find bugs. DevOps also makes use of code analysis tools that permit the detection of bugs or design smells, i.e., code statements that embody poor design and error-prone coding choices. One of the mainstream open-source Java code analysis tools is *FindBugs*. *FindBugs* can detect more than 300 object-oriented programming mistakes and dubious coding idioms. It has been downloaded more than a half million times and is used by many major companies [8, 9]. For example, Google has incorporated FindBugs into their standard testing and code review process, and has fixed more than 1 000 issues in their internal code base identified by FindBugs [8]. It has been demonstrated that object-oriented design smells can have a negative impact on maintainability [132], understandability [1] and change- or fault-proneness [68]. Identifying design smells and developing techniques to automatically detect them is thus crucial to assess the quality of software systems, UIs included. Yet, there is very limited support to ensure the code quality of UIs. Sparse research work have been conducted for developing tools that detect UI design smells for both WIMP and post-WIMP UIs. Finding bugs and design smells in UI code is not a trivial task since the existing object-oriented validation and maintenance techniques and tools (such as FindBugs) can hardly be seamlessly reused.

2.1.3 Approach

UI coverage. Improving the computation of UI coverage by UI tests requires a detailed analysis of UI code that is partly coded in XML for describing the UI (i.e., UIDLs), and in object-oriented. This is mandatory to extract UI information such as UI commands, user interactions, or data bindings. Most of the UI testing approaches focus on monkey testing: a robot interacts on the UI using enabled interactive objects to grab information about the UI behaviour to then produce tests [84]. These approaches are time-consuming as they explore at run time the EVG of a UI. We think that static analyses of the UI code can be used to initiate the computation of a fine-grained EVG to be used to compute the UI coverage. Figure 2.2 is an example of such a fine-grained EFG. A user interaction can be viewed as a sub-EFG; a

drag-and-drop interaction, for example, is a suite of *press*, *drag* and *move* events. When a user interaction is triggered, a UI command that acts on the system is produced. Then, a user can perform other user interactions that will form a complex EFG. Extracting such detailed EFGs from the code can be then used to compute UI coverage and test amplification.

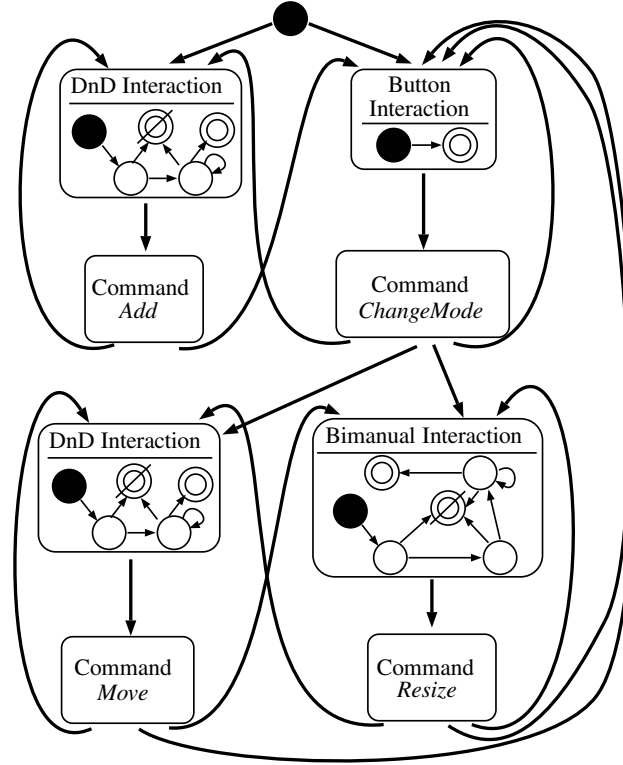


Figure 2.2: An example of a fine-grained EFG, adapted from [77]

UI test amplification. Supporting UI coverage has the following benefits. It helps testers in writing new UI tests to improve the quality of the UI. As DevOps aims at automating as far as possible the different steps of software delivery, recent research works focus on automatic test amplification [41, 42]. In a DevOps context, improving UI test coverage requires the production and amplification, as automatic as possible of UI tests based on:

- The use of the UI coverage to find UI zones to cover. Then use existing UI tests can be used to derive new UI tests that focus on these uncovered zones. Such a test amplification technique requires static analysing techniques of UI tests to extract the different steps and oracles that compose them. Then, new UI tests can be composed by assembling existing steps. For example, Listing 2.2 is a UI test composed of four UI steps: *addRec1*, *addRec2*, *clickOnRec1* and *ctrlClickOnRec2*. Then, the oracle checks the UI state. By analysing such UI tests, we should be able to infer new relevant sequences of steps and associated oracles.

```

@Test
public void testCtrlClickOnShapeAddsSelection() {
    new CompositeGUIVoidCommand(addRec1, addRec2, clickOnRec1,
                                ctrlClickOnRec2).execute();
    assertEquals(2, canvas.getDrawing().getSelection().size());
    assertNotSame(canvas.getSelection().getShapeAt(0).orElseThrow(),
                  canvas.getSelection().getShapeAt(1).orElseThrow());
}

```

Listing 2.2: A UI test

An important point is that extracting paths from an EFG is infinite (it is a graph). In a DevOps context, it is not relevant to produce UI tests as much as possible: test executions take time and may hamper the continuous delivery of software systems. The technique should also consider the *hot spots* of the UI: the navigation paths widely used to users. The UI coverage should then focus on these hot spots first during the UI test amplification. Identifying hot spots in a UI can be done by logging anonymous information while users are interacting with UIs [5].

- The use of UI mutation operators [76, 102]. An example of UI mutation operation is the removal of view templates. For example, Listing 2.3 is an Angular view template [60]. This template contained algorithmic instructions (*ngFor* for a *for* loop) and event processing instructions (*mousedown* that calls the method *cellClicked* on a mouse-down event). Mutations can remove or alter these instructions to check whether tests cover them. Mutants that survived can be used as the initial starting point to produce new tests.

```

<ng-container *ngFor="let y of [0,1,2,3,4,5,6]">
  <div #cells *ngFor="let x of [0,1,2,3,4,5,6]" class='cell'
    (mousedown)="cellClicked($event)"
    [attr.data-x]=x [attr.data-y]=y>
  </div>
</ng-container>

```

Listing 2.3: An Angular view template extracted from an HTML document

Flaky tests and UI smells detection. Detecting and fixing UI smells, including those that affect flaky UI tests, requires:

- Identifying and characterizing UI design smells. We have to conduct empirical studies on representative software systems to both identify UI design smells and evaluate their potential negative impact on the code. This requires a set of representative software systems to be the ground truth for large empirical studies. Studying design smells is empirically-driven since they appear as software engineers code software systems, UIs included. This also requires the analysis of historical information to evaluate the change- and error-proneness of UI code. In the design smell literature, the change- and error-proneness are considered as a negative impact of a design smell on the code [68, 109]. Computing the change- and error-proneness can be done by analysing historical information of software systems, such as Git or SVN commits. We started applying this principle on UI code with as results the identification of a UI smells we called *Blob Listener* [26, 78].

- Static code analysis to locate UI code and compute UI metrics. Identifying UI code among all the code of a software system is not an easy task since UI code may be intertwined with the rest of the code. The current code analysis techniques and tools focus on object-oriented concerns. They thus cannot be directly used to locate UI code. So, dedicated static code analyses are required to precisely locate UI code. The developed static code analyses will notably locate UI controllers, UI listener implementations, widget definitions and configurations, data bindings between data and widgets, UI commands and their possible undo/redo features.
- Assembling UI metrics and OO metrics to form detection heuristics. This concerns both UI code smells and UI test smells that may lead to flaky tests. Indeed, a recent study showed that 50 % of the fragile tests analysed were affected by bad practices [110]. The static UI code analyses will permit the computation of UI metrics and identify UI elements. The design of a detection heuristics, i.e., the assembly of specific metrics with specific values, are then necessary to automatically detect the identified UI design smells in the code. UI metrics will be mainly used, supplemented with object-oriented metrics to possibly precise the detection. This will be done empirical by both selecting and adjusting the metrics manually and using the change- and error-proneness results. The heuristics will be validated on the ground truth of the project.

2.2 Engineering domain-specific user interfaces

2.2.1 Research Context

I view modelling as the art of being at the correct level of abstraction when working on a specific problem. With this definition, modelling is not in opposition with programming, and this separation is far from being Manichean.

There is a tendency in many papers that we read to put a brick wall between modelling and programming — to treat them as conceptually different things that can only be bridged via transformations (created by these magical wizards, or transformation engineers). [107]

For example, when I code an API I both program using a GPL and design a model. I aim at providing the users of this API with a set of types and services, in fact a sub language, to help them in addressing a very specific problem. Under certain conditions APIs are even considered as a specific case of DSLs called internal DSLs [55]. Another example with the R programming language dedicated to data processing and analysis. When I write an R script to do statistics, I feel like I am programming, i.e., writing a receipt than can be understood and executed by a computer. Yet, R is a domain-specific language, a textual language specifically designed to do statistics on data. As explained by Rumpe and France: ‘modelling languages that support the building of executable models can be viewed as approximate forms of very high-level programming languages’ [116].

The concept of abstraction is not limited to software modelling but is also a corner-stone of software engineering and programming. The time-honoured *separation of concerns* [111] is a perfect illustration of this: each concern, i.e., problem, that composes a software system should be confined from the rest of the concerns. Software maintenance and evolution, or the fact that different concerns may involve different specific stakeholders (that may not be software engineers) are reasons of such a separation. Regarding this last point, when working on a specific concern a stakeholder may want specific tools, namely DSLs:

It is common to develop DSLs for narrow, well-understood domains. In contrast to perceived wisdom – that significant effort should be employed in developing models that cover broad domains and capture knowledge in that domain – practical application of domain modelling is ‘quick and dirty’, where DSLs (and accompanying generators) can be developed sometimes in as little as two weeks. [130]

2.2.2 Scientific Challenges

Modelling and DSLs are not limited to software engineering [36], and DSLs were not invented for software engineers. Figure 2.3 is an example of modelling and DSLs in the textile industry. First, a stylist focuses on the global look, the style, the materials of a cloth. Second, a sewing pattern maker transforms the stylist’s model into another model that focuses on the dimensions of the cloth. Such a model is called a sewing pattern. Finally, a dressmaker takes this sewing pattern as input with materials and ‘compiles’ them as a real cloth. This chain typifies a model-driven process that uses different DSLs to match the problem of the different involved stakeholders.

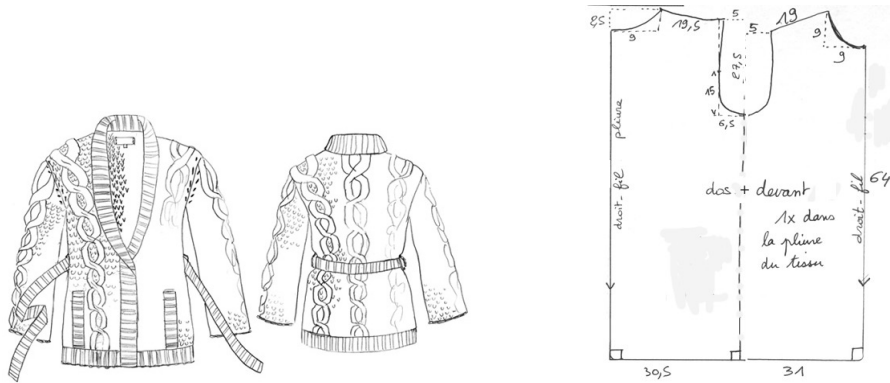


Figure 2.3: Model-driven engineering and DSLs in the textile industry. On the left, a model of a cloth sketched by a stylist. On the right, a model (a sewing pattern) of the same cloth from the viewpoint of a sewing pattern maker.

To face this diversity of potential stakeholders, DSLs can take many forms, such as graphical, tabular, or textual. The most widespread [64] yet criticised [130, 64] graphical DSL in software engineering is certainly UML (that is in fact a set of DSLs) [103]. The current perceived gap between programming and modelling may be partly due to UML: many people think that modelling means using UML, drawing boxes and arrows and generating code [107]. DSLs are in fact used for various purposes [116, 130, 64] (by order of importance and not limited to): understanding a problem at an abstract level; team communication; capture and document designs; code generation; simulation and execution.

To achieve these goals, DSLs rely on a large panel of tools: editors, linters, simulators, compilers, etc. Because DSLs can be created on a daily basis, the development of these tools is a critical point. These tools must be quickly developed, maintained and tested, while being usable. To do so, the current language engineering community mainly regroups people from the model-driven engineering and programming language communities. The language engineering community should strongly rely on the UI engineering community as well: the development of DSLs implies the development of IDEs and the lack of usability of such IDEs hinders the adoption of DSLs [57, 95]. Approaches have been proposed to evaluate the usability of a DSL [4, 12]. Other approaches focus on improving the interactivity or the understandability of specific DSLs [58, 79]. Yet, language engineering requires techniques to ease the development of usable tools, in particular IDEs, for any DSL. This means that interactivity, usability, visualisation, must be considered during the development process of any DSL. The current practices consist in producing IDEs to then customise them by hand by using the potency and facing the limits of the underlying toolkits. I identify two main challenges.

Visualisation and interactivity of DSLs. In our earlier work, we worked at improving the navigability within large UML class diagrams [27, 28]. The development of the proposed interactive features does not scale the fact that DSLs can be built on a daily basis: our development was specifically done for the UML class diagram and using the developed features for another graphical DSL requires the entire re-development of them. The DSL development process should help language designers in selecting and customising interactive and navigation features that can help in using their DSLs.

Analysing DSL usages. In our earlier work, we investigate the automatic production of documentation for DSLs in a usability purpose [74]. The proposed approach analyses DSL artefacts (metamodel, models, syntaxes) to extract information and build documentation and illustrating examples. We think that other kinds of analyses should be developed to study the usage of DSLs.

First, ‘*there is little doubt that examples are generally useful for teaching and learning and understanding. For instance, well-chosen program samples could help those learning programming (languages)*’ [71]. Lämmel uses the term *chrestomathy* to refer as a collection of programs/models that can be used as illustrative examples [71]. In our previous work, the examples produced from existing models suffer from several limits: they are code examples of textual DSLs only; these examples lack of context, which hinders their understandability.

Second, in the early stages of the DSL development process *domain analysis* can help in identifying the concepts of the DSL and in producing some general documentation [127]. In complement, DSL must be evaluated *empirically* based on their real use. This would permit engineers to improve DSL syntaxes and editors.

2.2.3 Approach

Visualisation and interactivity of DSLs. HCI researchers develop interactive features and visualisation techniques on a daily basis. DSL development toolkits should integrate such techniques during the design of the DSL editors. A typical example for graphical editors is the semantic zooming: a semantic zoom shows different details depending on the zoom level, contrary or in complement of the physical zoom that scales the displayed information. Listing 2.4 shows an illustrative example of a possible model that describes the interactive feature of an editor for the Ecore DSL [123]. An Ecore model is usually represented using a class diagram graphical syntax. Such syntax can be defined using classical DSL tools such as Sirius [50]. In complement of the graphical syntax, a model can supplement the development process by describing the interactive features that the editor should have. In this example, we defined a semantic zoom. Depending on the zoom level (here: 75 % or 50 %), some details are masked. For example with the example, at a zoom level of 75 %, class attributes and operations are hidden. At a zoom level of 50 %, reference cardinalities and labels are hidden. In the same vein, the default layout, can be defined in such models.

```

editor ClassDiagram {
    metamodel: './Ecore.ecore'
    semanticZoom: {
        75: [EClass.attributes, EClass.operations]
        50: [EReference.lowerBound, EReference.upperBound, EReference.name]
    }
    layout: hierarchical
}

```

Listing 2.4: A model that describes interactive features of a DSL editor.

This principle of designing the interactive features of an editor is the same for other kinds of syntaxes. The main difference is that the possible interactive features differ. For example, Listing 2.5 shows another example of such a editor model but for a textual syntax. Xtext is certainly the widespread textual DSL toolkit in the MDE community [21]. The model of Listing 2.5 complements an Xtext model by easing the declaration of interactive or visualisation

features. For example, the principle of code bubbles is an alternative of standards panels in IDEs [31]. The model also selects the kind of debuggers, for executable DSLs, to employ. Here we use an omniscient debugger [30], where its tracing process is located in the package `org.kompren.xdsl`. The execution traces are visualised using a call-stack visualisation technique (*Gralka*) [61]. Zooms are also important for textual DSLs.

```

editor KomprenEditor {
  metamodel: './Kompren.ecore'
  editor: bubbles
  debugger: ominiscient[org.kompren.xdsl] {
    callstack: Gralka
  }
}

```

Listing 2.5: A model that describes interactive features of a DSL editor.

Analysing DSL usages. Mining software repositories, such as *Github*, permits the gathering of models. In [46], we conducted an empirical study on UML models. After a cleaning process, we obtained 1651 valid UML2 models. We then used these models, developed by people from the academy or the industry, to conduct experiments. Researchers can use models of a DSL gathered from such software repositories for various purposes. One can analyse the use of a DSL: concepts used in models are compared to the DSL metamodel to identify the hotspots of the language, or the underused parts. This can be also a source for: identifying and characterising DSL smells; identifying recurrent patterns; producing model examples; building search engines. Regarding this last point, we developed in an unpublished work a search engine for Ecore models. This search engine used the Ecore models stored on Github. Figure 2.4 depicts a prototype of this search engine. Users provide keywords (here *class*, *package* and *operation*) and the search engine then displays a list of Ecore models. For each result, the search engine extracts a sub-model that makes use of the provided keywords using a model slicing technique.


However, by essence a DSL targets a narrow audience. This can be a major limit of such an approach that must focus on widespread DSLs (ThingML, Maven POM, OCL, etc.).

Q class package operation

We have found 83 result(s)

fUML.ecore

+



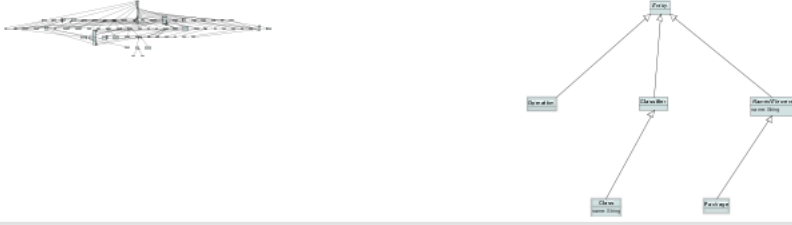
package class operation

NsURI: <http://www.modelexecution.org/fuml>

Referred metamodels: <http://www.eclipse.org/emf/2002/Ecore>

SysML.ecore


+



package class operation

uml.ecore

+




package class operation

NsURI: <http://www.eclipse.org/uml2/3.0.0/UML>

Referred metamodels: <http://www.eclipse.org/emf/2002/Ecore>, [ecore.ecore](http://www.eclipse.org/emf/2002/Ecore.ecore)

CMOF.ecore

+



package class operation

NsURI: <http://schema.omg.org/spec/MOF/2.0/cmof.xml>

Referred metamodels: <http://www.eclipse.org/emf/2002/Ecore>

< 1 2 3 4 5 6 7 8 9 >

Figure 2.4: An example of a search engine for Ecore models.

2.3 User interactions as a first-class programming concept

2.3.1 Research Context

The user interactions provided by a UI form a dialect between a system and its users [62]: a user interaction can be viewed as a sentence composed of predefined words that correspond to low-level UI events. For example, the execution of a drag-and-drop interaction can be viewed as a sentence emitted by a user to the system. This sentence is usually composed of the words *pressure*, *move* and *release*, that are UI events assembled in a specific order. A core step while programming UIs is the processing of such sentences. This requires to first assemble UI events to build a user interaction, to then turn a user interaction execution into a command that will act on the software. To do so, while the human-computer interaction community designs novel and complex UIs for highly interactive user interfaces, software developers still use a technique proposed with SmallTalk and the *Model-View-Controller* (MVC) pattern in the 80s [69]: the UI event processing model, currently implemented using callback methods or reactive programming [10] libraries. This model considers low-level UI events as the first-class concept developers use for coding more and more complex user interactions. The reason is that interacting with classical widgets (e.g., buttons, lists, menus) is usually one-shot: the standard event processing libraries treats a single UI event, such as a mouse pressure on a button or menu. For more complex user interactions, an abstraction gap exists leading to several critical flaws that hinder code reuse and affect separation of concerns and code complexity:

- the concept of user interaction does not exist, so developers have to re-develop user interactions for each UI using UI events;
- developers have to manually code and maintain the glue code that processes UI events to produce output commands;
- the code that processes UI events intertwines several concerns, in particular the behavior of user interactions and the glue code that produce commands;
- the use of event callbacks to process UI events can lead to ‘spaghetti’ code [98, 105] and relies on the *Observer* pattern that has several major drawbacks [83, 118, 117, 54], can be affected by design smells [26].

One may note that this problem was already at the heart of my PhD thesis in 2009 [22]. Ten years after, software engineering still faces this issue more than ever: software systems are more and more interactive, yet industrial graphical toolkits never evolved on this problem.

2.3.2 Scientific Challenges

We illustrate the current limitations of processing UI events using the following example. In this example, a user can move a rectangular node displayed by the user interface using a drag-lock interaction. A drag-lock is a kind of drag-and-drop (DnD) interaction. Using a standard DnD, the user performs a pressure on a node, drags it, to finally release it. A drag-lock starts by double-clicking on the node to drag. The user can then move the locked node until she double-clicks again at the dropping location. The drag-lock interaction is an

interesting motivating example as it is a standard UI but not provided off-the-shelf by the current UI toolkits. Figure 2.5 (on the left) depicts using a finite-state machine (FSM) the assembly of UI events to build a drag-lock interaction. A transition refers to a UI event or another UI. A UI execution ends when its FSM reaches a terminal state.

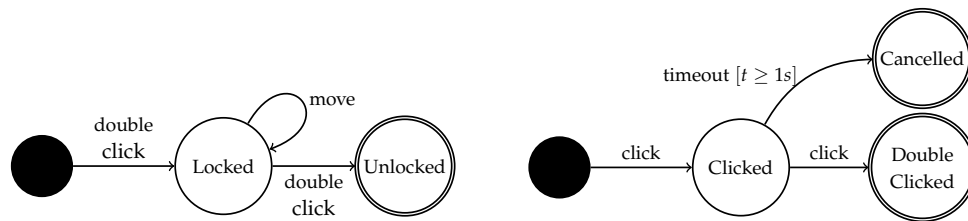


Figure 2.5: FSMs of the drag-lock (top) and double-click (bottom) user interactions used in Listing 2.6. The double-click transition of the drag-lock refers to the double-click interaction on the right.

```

let isDragLocked = false;
const mm_listener = function(evt) {
  draggable.attr({ x: evt.x, y: evt.y });
};
const mu_listener = function(evt) {
  removeEventListener("mousemove", mm_listener);
  removeEventListener("mouseup", mu_listener);
};
draggable.mousedown(function(evt) {
  if(evt.button == 0) {
    draggable.attr({ x: evt.x, y: evt.y });
    addEventListener("mousemove", mm_listener);
    addEventListener("mouseup", mu_listener);
  }
});
draggable.dblclick(function(evt) {
  if(evt.button == 0) {
    if(isDragLocked) {
      draggable.style.cursor = '';
      removeEventListener("mousemove", mm_listener);
    } else {
      draggable.style.cursor = 'hand';
      addEventListener("mousemove", mm_listener);
    }
    isDragLocked = !isDragLocked;
  }
});

```

Listing 2.6: A JavaScript code snippet that moves a node using a drag-lock, adapted from [105]

During the drag-lock of this example, the UI uses a ‘hand’ cursor as user feedback. Listing 2.6 is a JavaScript implementation of this example. Classical event callbacks, namely mouse pressure (Line 9), mouse move (Line 2), mouse up (Line 2) and double-click (Line 16), are used to implement the drag-lock. The main UI concept used to code the example is the UI event.²

This drag-lock example of Listing 2.6 suffers from the following flaws:

Separation of concerns. Separation of concerns is a core concept in software engineering [111]. Listing 2.6 illustrates how relying on UI events breaks this concept by intertwining in the same code:

- The behaviour of the user interaction (the drag-lock). Current UI toolkits and approaches consider UI events as a first-class concept for coding user interfaces. UI events, however, are *low-level implementation details* that developers need to manually assemble to build user interactions, such as the drag-lock.
- The transformation of user interactions into UI commands. In the same code that assembles UI callbacks to build a user interaction, developers have to define how to produce output UI commands. Line 3 in Listing 2.6 is the instruction that moves the dragged node. Note that this instruction is not strictly-speaking a command as it is not encapsulated in a specific command object [59].
- Conditions that constraint the execution of the UI. Lines 10 and 17 check whether the user used the drag-lock using the primary button of the mouse.
- Statements that provide feedback to the user. In the example the cursor is changed during the drag-lock (Lines 19 and 22) to provide the user with feedback about what she is doing.

Software reuse. Software reuse is also a core and long-standing software engineering concept [70, 67]. Software reuse takes various forms:

- As explained by [70], ‘*all approaches to software reuse use some form of abstraction for software artefacts. Abstraction is the essential feature in any reuse technique*’. By still considering UI events as first-class concerns, current UI event processing approaches suffer from this exact problem of abstraction that hinders reuse.
- Libraries and frameworks enable software reuse by providing developers with pre-defined and reusable artefacts [70, 67]. The HCI community designs various user interactions that engineers need to develop and embed in libraries to ease their (re)use.
- User interactions can be classified in different categories. For example, drag-lock is a kind of DnD interaction. A developer should easily replace a DnD with a drag-lock as their underlying data are the same: start and end position data. Figure 2.6 depicts another example with alternative behaviours of the drag-lock and double-click interactions of Figure 2.5. The double-click is now cancelled on a move between the two clicks. The timeout has changed to 0.5 s. The drag-lock now requires at least one move

²One may note that some UI events are not atomic: if the double-click is a user interaction based on several events (*pressure* and *release*), it is sometimes considered as a UI event since it is one-shot. This is also the case of the mouse click and the key typing interactions. Figure 2.5 (on the right) depicts using an FSM the assembly of UI events to build a standard double-click.

between the two double-clicks, otherwise it is cancelled. A pressure on the key 'ESC' cancels the UI. A developer should easily replace the standard DnD by this variant. This can be hardly achieved with the current UI event processing approaches as a developer has to modify the assembly of UI events. By considering user interactions as objects, object polymorphism would ease this form of reuse.

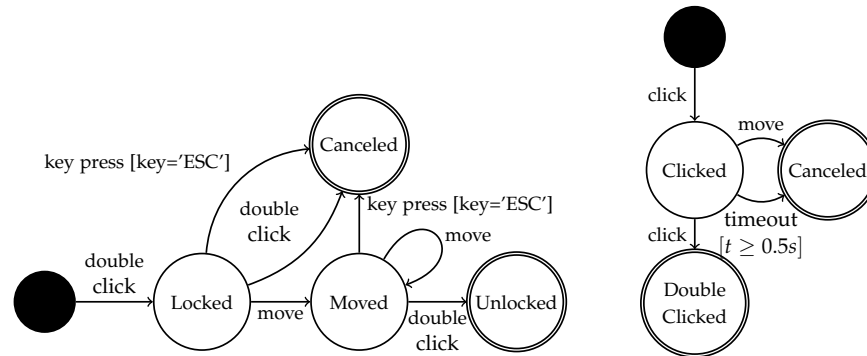


Figure 2.6: Alternative versions of the drag-lock (left) and the double-click (right) user interactions

- Undo/redo. The code of Listing 2.6 modifies the data directly in the event handlers (Line 3). So, the changes cannot be stored to be then undone and redone. This would require glue code manually crafted by developers in the code of Listing 2.6 to support such a feature. UI processing approaches should provide mechanisms to ease the support and the reuse of undoable commands. This requires to reify commands as first-class concepts following the *Command* design pattern [59].

Complexity. Intertwining in the same code the assembly of UI events to build user interactions, the transformation of UI events into commands makes the code more complex. The processing of low-level UI events may lead to a code smell named 'spaghetti' code [98, 105] that makes the code more complex.

Researchers proposed different programming models to overcome flaws of the current UI event processing model [98, 105, 97, 6, 90, 104, 18]. None of them overcome all the above-mentioned flaws.

2.3.3 Approach

The flaws of the current event processing model require a deep change in the way developers handle user input events. The new event processing model to develop must provide developers with constructs at the adequate level of abstraction. This new model must also permit to reduce the gap between HCI designers and software engineers by employing the same core concept: *user interaction*, instead of focusing on atomic input events. This should ease the transfer of new user interactions, designed by the HCI community, into software toolkits.

To do so, the new UI processing model to propose³ should take the form of a fluent API, aka. an internal DSL. The goal is to provide developers with an API in programming languages they know but supplemented with supplied abstractions.

Consider the example of Listing 2.6: a drag-lock interaction (Figure 2.6 depicts its behaviour) moves an object. The following Java code depicts a possible example of the new model to develop. This code example configures and builds a binding between a *DragLock* interaction and a *Translate* command:

```

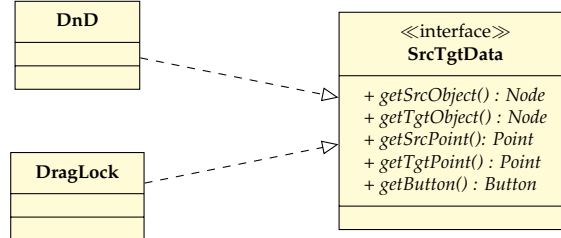
1 nodeBinder(new DragLock(), i -> new Translate(i.getSrc().getUserData()).
2   on(node). // node is an interactive object of the user interface
3   first((i, c) -> i.getSrcObject().setEffect(new DropShadow())) .
4   then((i, c) -> c.setCoord(
5     c.getShape().getX() + i.getTgtPt().getX() - i.getSrcPt().getX(),
6     c.getShape().getY() + i.getTgtPt().getY() - i.getSrcPt().getY())) .
7   when(i -> i.getButton() == MouseButton.PRIMARY) .
8   exec() .
9   endOrCancel((i, c) -> i.getSrcObject().setEffect(null)) .
10  bind();

```

Listing 2.7: A binding that transforms a drag-lock interaction into a user command

This example relies on five key concepts:

- *User interactions are reusable objects* that graphical libraries should provide to developers instead of low-level UI events. Developers, moreover, rarely develop user interactions: the HCI community defines state-of-the-art user interactions that graphical libraries should implement and provide to developers off-the-shelf.

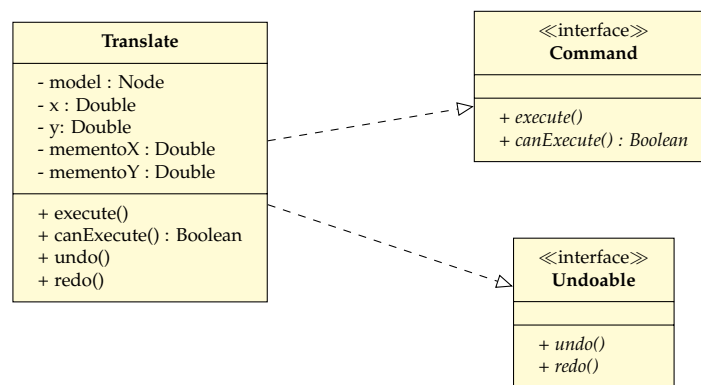


- *A user interaction should be stateful and expose data* that a binding can use. The *DragLock* class used in Listing 2.7 provides data that binding routines can use (e.g., in *map*, *first*, *then*, *when* and *endOrCancel*). The above class diagram depicts the example of the drag-lock and DnD interactions. The underlying data of the drag-lock interaction are defined in the *SrcTgtData* interface. These data are composed of: the source position (*getSrcPoint*); the source picked object (*getSrcObject*); the target position (*getTgtPoint*); the target picked object (*getTgtObject*); the mouse button (*getButton*). The interaction data are updated during the execution of the user interaction. The binding routines, such as *map* in Listing 2.7, do not use the running interaction directly. Instead they can access the interaction data, i.e., the type of *i* is *SrcTgtData*. Because the drag-lock and the DnD interactions are of the same kind they have the same type of data. So, in the

³User interactions form a core concept of this model instead of events. So, we call this model a user interaction processing model instead of an event processing model.

code of Listing 2.7 a developer can replace the drag-lock interaction with a DnD with no other change on the binding. The model to develop should make no assumption on how user interactions are implemented. This can be, for example, using FSMs [23, 6], Petri nets [99], or reactive programming [39]. Our examples make use of FSMs.

- *UI commands should be reusable and undoable objects* defined separately from bindings. UI commands rely on the *Command* pattern [59]. The UI command *Translate* in Listing 2.7 has the following class diagram. The class *Translate* defines an *Undoable Command*. *Translate* has attributes that correspond to the data required to execute the command. A binding produces *Command* objects that can be *Undoable*. Undoable commands can be undone and redone. This may require specific *Memento* [59] data to save the initial state of the objects modified by the command. The attributes *mementoX* and *mementoY* form the memento of *Translate*.



- *The new model should focus on transforming input user interactions into output commands.* When a user interacts with a user interface, her final goal is to give orders, i.e., commands, to the system. A binding relies on this goal. One UI command can be created, updated, executed, or cancelled, along one execution of a user interaction. Different binding routines are called during one execution of the user interaction to create and configure an output command. The execution of the current command and its registration if undoable are automatically managed by the binding.
- *The new model should provide developers with advanced mechanisms* for debugging UIs and analysing their usages. For example, the new model can embed logging mechanisms to ease the analysis of user interactions to improve the usability of UIs [5] or to build UI recommendation systems [133].

To have industrial impact, the proposed model must match the engineering reality of software engineers and the expressiveness that HCI designers should expect. Researchers must evaluate the implementations of the proposed new model with a representative panel of software engineers and using realistic scenarios. Moreover, the proposed model should be able to express various kinds of modern user interactions designed and prototyped on a daily basis by the HCI community. Beyond a new model, this perspective requires software engineering syllabuses to provide students with a strong background on HCI and on UI engineering.

Selected Publications

I co-authored with 43 different researchers. I worked with five of them through international collaborations: Montréal University, Canada; UQAM, Canada; Colorado State University, USA.

I co-authored:

- seven international journal papers;
- one national journal papers;
- 18 international conference papers;
- eight national conference papers;
- eight international demonstration or workshop papers;
- one workshop proceedings.

Selected Journal Papers

A. Blouin, V. Lelli, B. Baudry and F. Coulon, 'User Interface Design Smell: Automatic Detection and Refactoring of Blob Listeners', *Information and Software Technology*, vol. 102, pp. 49–64, 2018. <https://hal.inria.fr/hal-01499106> (cit. on pp. 9, 13, 14, 19, 28, 32, 35, 42).

G. Le Moulec, A. Blouin, V. Gouranton and B. Arnaldi, 'Automatic Production of End User Documentation for DSLs', *Computer Languages, Systems and Structures*, vol. 54, pp. 337–357, 2018. <https://hal.inria.fr/hal-01549042> (cit. on pp. 12, 13, 39).

D. A. Méndez-Acuña, J. A. Galindo, B. Combemale, A. Blouin and B. Baudry, 'Reverse Engineering Language Product Lines from Existing DSL Variants', *Journal of Systems and Software*, vol. 133, pp. 145–158, 2017. <https://hal.inria.fr/hal-01524632> (cit. on p. 12).

T. Degueule, B. Combemale, A. Blouin, O. Barais and J.-M. Jézéquel, 'Safe Model Polymorphism for Flexible Modeling', *Computer Languages, Systems and Structures*, vol. 49, p. 30, 2016. <https://hal.inria.fr/hal-01367305> (cit. on pp. 12, 13, 40).

A. Blouin, B. Combemale, B. Baudry and O. Beaudoux, 'Kompren: Modeling and Generating Model Slicers', *Software and Systems Modeling*, vol. 14, no. 1, pp. 321–337, 2015. <https://hal.inria.fr/hal-00746566> (cit. on p. 11).

A. Blouin, N. Moha, B. Baudry, H. Sahraoui and J.-M. Jézéquel, ‘Assessing the Use of Slicing-based Visualizing Techniques on the Understanding of Large Metamodels’, *Information and Software Technology*, vol. 62, pp. 124–142, 2015. <https://hal.inria.fr/hal-01120558> (cit. on pp. 11, 13, 26, 27, 38).

Selected Conference Papers

G. Le Moulec, F. Argelaguet, V. Gouranton, A. Blouin and B. Arnaldi, ‘AGENT: Automatic Generation of Experimental Protocol Runtime’, in *ACM Symposium on Virtual Reality Software and Technology*, ser. VRST’17, 2017, pp. 1–10. <https://hal.archives-ouvertes.fr/hal-01613873> (cit. on p. 10).

T. Degueule, B. Combemale, A. Blouin, O. Barais and J.-M. Jézéquel, ‘Melange: A Meta-language for Modular and Reusable Development of DSLs’, in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE’15, 2015, pp. 25–36. <https://hal.inria.fr/hal-01197038> (cit. on pp. 11, 12).

V. Lelli, A. Blouin and B. Baudry, ‘Classifying and Qualifying GUI Defects’, in *8th IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST’15, 2015, pp. 1–10. <https://hal.inria.fr/hal-01114724> (cit. on pp. 9, 13, 35).

G. Bécan, N. Sannier, M. Acher, O. Barais, A. Blouin and B. Baudry, ‘Automating the Formalization of Product Comparison Matrices’, in *29th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE’14, 2014. <https://hal.inria.fr/hal-01058440> (cit. on pp. 11, 13).

O. Beaudoux, M. Clavreul, A. Blouin, M. Yang, O. Barais and J.-M. Jézéquel, ‘Specifying and Running Rich Graphical Components with Loa’, in *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, ser. EICS’12, 2012, pp. 169–178. <https://hal.inria.fr/hal-00684881> (cit. on p. 45).

O. Beaudoux, A. Blouin, O. Barais and J.-M. Jézéquel, ‘Specifying and implementing UI Data Bindings with Active Operations’, in *ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS’11, 2011, pp. 127–136. <https://hal.inria.fr/inria-00590896>.

A. Blouin, B. Combemale, B. Baudry and O. Beaudoux, ‘Modeling Model Slicers’, in *ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS’11, 2011, pp. 62–76. <https://hal.inria.fr/inria-00609072> (cit. on p. 11).

A. Blouin, B. Morin, O. Beaudoux, G. Nain, P. Albers and J.-M. Jézéquel, ‘Combining Aspect-Oriented Modeling with Property-Based Reasoning to Improve User Interface Adaptation’, in *ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS’11, 2011, pp. 85–94. <https://hal.inria.fr/inria-00590891> (cit. on pp. 6, 10).

O. Beaudoux, A. Blouin, O. Barais and J.-M. Jézéquel, ‘Active Operations on Collections’, in *ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS’10, 2010, pp. 91–105. <https://hal.inria.fr/inria-00542763>.

A. Blouin and O. Beaudoux, ‘Improving modularity and usability of interactive systems with Malai’, in *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, ser. EICS’10, 2010, pp. 115–124. <https://hal.inria.fr/inria-00477627> (cit. on pp. 10, 47).

PhD Manuscripts

G. Le Moulec, ‘Synthèse d’applications de réalité virtuelle à partir de modèles’, PhD thesis, INSA de Rennes, 2018. <https://tel.archives-ouvertes.fr/tel-01959918> (cit. on pp. 10, 22).

T. Degueule, ‘Composition and interoperability for external domain-specific language engineering’, PhD thesis, Université Rennes 1, 2016. <https://tel.archives-ouvertes.fr/tel-01488300> (cit. on pp. 12, 22).

V. L. Leitao, ‘Testing and maintenance of graphical user interfaces’, PhD thesis, INSA de Rennes, 2015. <https://tel.archives-ouvertes.fr/tel-01232388> (cit. on p. 22).

A. Blouin, ‘Un modèle pour l’ingénierie des systèmes interactifs dédiés à la manipulation de données’, PhD thesis, Université d’Angers, Nov. 2009. <https://tel.archives-ouvertes.fr/tel-00446314> (cit. on pp. 10, 42).

Bibliography

- [1] M. Abbes, F. Khomh, Y. G. Guéhéneuc and G. Antoniol, ‘An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension’, in *Proceedings of the European Conference on Software Maintenance and Reengineering*, ser. CSMR’11, 2011, pp. 181–190. doi: [10.1109/CSMR.2011.24](https://doi.org/10.1109/CSMR.2011.24) (cit. on pp. 15, 33).
- [2] M. Acher, ‘Managing, multiple feature models: Foundations, languages and applications’, PhD thesis, Nice, 2011. <http://www.mathieuacher.com/PhDAcher2011-revised.pdf> (cit. on p. 10).
- [3] P. A. Akiki, A. K. Bandara and Y. Yu, ‘Adaptive model-driven user interface development systems’, *ACM Comput. Surv.*, vol. 47, no. 1, 9:1–9:33, May 2014. doi: [10.1145/2597999](https://doi.org/10.1145/2597999) (cit. on p. 6).
- [4] D. Albuquerque, B. Cafeo, A. Garcia, S. Barbosa, S. Abrahão and A. Ribeiro, ‘Quantifying usability of domain-specific languages: An empirical study on software maintenance’, *Journal of Systems and Software*, vol. 101, pp. 245–259, 2015. doi: [10.1016/j.jss.2014.11.051](https://doi.org/10.1016/j.jss.2014.11.051) (cit. on p. 38).
- [5] A. Apaolaza and M. Vigo, ‘WevQuery: Testing Hypotheses About Web Interaction Patterns’, *Proc. ACM Hum.-Comput. Interact.*, vol. 1, no. EICS, 4:1–4:17, 2017. doi: [10.1145/3095806](https://doi.org/10.1145/3095806) (cit. on pp. 35, 47).
- [6] C. Appert and M. Beaudouin-Lafon, ‘SwingStates: Adding state machines to Java and the Swing toolkit’, *Software: Practice and Experience*, vol. 38, no. 11, pp. 1149–1182, 2008. doi: [10.1002/spe.v38:11](https://doi.org/10.1002/spe.v38:11) (cit. on pp. 45, 47).
- [7] S. Arlt, A. Podelski, C. Bertolini, M. Schäfer, I. Banerjee and A. M. Memon, ‘Lightweight static analysis for GUI testing’, in *IEEE 23rd International Symposium on Software Reliability Engineering*, ser. ISSRE’12, IEEE, 2012, pp. 301–310. doi: [10.1109/ISSRE.2012.25](https://doi.org/10.1109/ISSRE.2012.25) (cit. on p. 32).
- [8] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix and W. Pugh, ‘Using Static Analysis to Find Bugs’, *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008. doi: [10.1109/MS.2008.130](https://doi.org/10.1109/MS.2008.130) (cit. on p. 33).
- [9] N. Ayewah and W. Pugh, ‘The google findbugs fixit’, in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA ’10, 2010, pp. 241–252. doi: [10.1145/1831708.1831738](https://doi.org/10.1145/1831708.1831738) (cit. on p. 33).
- [10] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx and W. d. Meuter, ‘A survey on reactive programming’, *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, p. 52, 2013. doi: [10.1145/2501654.2501666](https://doi.org/10.1145/2501654.2501666) (cit. on pp. 30, 42).

- [11] I. Banerjee, B. Nguyen, V. Garousi and A. Memon, ‘Graphical user interface (GUI) testing: Systematic mapping and repository’, *Information and Software Technology*, vol. 55, no. 10, pp. 1679–1694, 2013. doi: [10.1016/j.infsof.2013.03.004](https://doi.org/10.1016/j.infsof.2013.03.004) (cit. on p. 9).
- [12] A. Barišić, V. Amaral and M. Goulão, ‘Usability driven DSL development with USE-ME’, *Computer Languages, Systems & Structures*, vol. 51, pp. 118–157, 2018. doi: [10.1016/j.cl.2017.06.005](https://doi.org/10.1016/j.cl.2017.06.005) (cit. on p. 38).
- [13] A. Barišić, V. Amaral and M. Goulão, ‘Usability evaluation of domain-specific languages’, in *2012 Eighth International Conference on the Quality of Information and Communications Technology*, ser. QUATIC’12, IEEE, 2012, pp. 342–347. doi: [10.1109/QUATIC.2012.63](https://doi.org/10.1109/QUATIC.2012.63) (cit. on p. 6).
- [14] L. Bass, P. Clements and R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2003 (cit. on p. 10).
- [15] M. Beaudouin-Lafon, ‘Designing interaction, not interfaces’, in *Proceedings of the working conference on Advanced visual interfaces*, ser. AVI ’04, ACM, 2004, pp. 15–22. doi: [10.1145/989863.989865](https://doi.org/10.1145/989863.989865) (cit. on p. 7).
- [16] O. Beaudoux, A. Blouin, O. Barais and J.-M. Jézéquel, ‘Active Operations on Collections’, in *ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS’10, 2010, pp. 91–105. <https://hal.inria.fr/inria-00542763>.
- [17] O. Beaudoux, A. Blouin, O. Barais and J.-M. Jézéquel, ‘Specifying and implementing UI Data Bindings with Active Operations’, in *ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS’11, 2011, pp. 127–136. <https://hal.inria.fr/inria-00590896>.
- [18] O. Beaudoux, M. Clavreul, A. Blouin, M. Yang, O. Barais and J.-M. Jézéquel, ‘Specifying and Running Rich Graphical Components with Loa’, in *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, ser. EICS’12, 2012, pp. 169–178. <https://hal.inria.fr/hal-00684881> (cit. on p. 45).
- [19] G. Bécane, N. Sannier, M. Acher, O. Barais, A. Blouin and B. Baudry, ‘Automating the Formalization of Product Comparison Matrices’, in *29th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE’14, 2014. <https://hal.inria.fr/hal-01058440> (cit. on pp. 11, 13).
- [20] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung and D. Marinov, ‘DeFlaker : Automatically Detecting Flaky Tests’, in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE ’18, 2018, pp. 433–444. doi: [10.1145/3180155.318016](https://doi.org/10.1145/3180155.318016) (cit. on p. 33).
- [21] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013 (cit. on p. 39).
- [22] A. Blouin, ‘Un modèle pour l’ingénierie des systèmes interactifs dédiés à la manipulation de données’, PhD thesis, Université d’Angers, Nov. 2009. <https://tel.archives-ouvertes.fr/tel-00446314> (cit. on pp. 10, 42).
- [23] A. Blouin and O. Beaudoux, ‘Improving modularity and usability of interactive systems with Malai’, in *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, ser. EICS’10, 2010, pp. 115–124. <https://hal.inria.fr/inria-00477627> (cit. on pp. 10, 47).

- [24] A. Blouin, B. Combemale, B. Baudry and O. Beaudoux, 'Kompren: Modeling and Generating Model Slicers', *Software and Systems Modeling*, vol. 14, no. 1, pp. 321–337, 2015. <https://hal.inria.fr/hal-00746566> (cit. on p. 11).
- [25] A. Blouin, B. Combemale, B. Baudry and O. Beaudoux, 'Modeling Model Slicers', in *ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS'11, 2011, pp. 62–76. <https://hal.inria.fr/inria-00609072> (cit. on p. 11).
- [26] A. Blouin, V. Lelli, B. Baudry and F. Coulon, 'User Interface Design Smell: Automatic Detection and Refactoring of Blob Listeners', *Information and Software Technology*, vol. 102, pp. 49–64, 2018. <https://hal.inria.fr/hal-01499106> (cit. on pp. 9, 13, 14, 19, 28, 32, 35, 42).
- [27] A. Blouin, N. Moha, B. Baudry and H. Sahraoui, 'Slicing-based Techniques for Visualizing Large Metamodels', in *IEEE Working Conference on Software Visualization*, ser. VIS-SOFT 2014, 2014. <https://hal.inria.fr/hal-01056217> (cit. on pp. 11, 38).
- [28] A. Blouin, N. Moha, B. Baudry, H. Sahraoui and J.-M. Jézéquel, 'Assessing the Use of Slicing-based Visualizing Techniques on the Understanding of Large Metamodels', *Information and Software Technology*, vol. 62, pp. 124–142, 2015. <https://hal.inria.fr/hal-01120558> (cit. on pp. 11, 13, 26, 27, 38).
- [29] A. Blouin, B. Morin, O. Beaudoux, G. Nain, P. Albers and J.-M. Jézéquel, 'Combining Aspect-Oriented Modeling with Property-Based Reasoning to Improve User Interface Adaptation', in *ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS'11, 2011, pp. 85–94. <https://hal.inria.fr/inria-00590891> (cit. on pp. 6, 10).
- [30] E. Bousse, J. Corley, B. Combemale, J. Gray and B. Baudry, 'Supporting efficient and advanced omniscient debugging for xDSMLs', in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2015, 2015, pp. 137–148. doi: 10.1145/2814251.2814262 (cit. on p. 40).
- [31] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeptura and J. J. LaViola Jr., 'Code bubbles: A working set-based interface for code understanding and maintenance', in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10, 2010, pp. 2503–2512. doi: 10.1145/1753326.1753706 (cit. on p. 40).
- [32] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon and J. Vanderdonckt, 'A unifying reference framework for multi-target user interfaces', *Interacting with computers*, vol. 15, no. 3, pp. 289–308, 2003. doi: 10.1016/S0953-5438(03)00010-9 (cit. on pp. 6, 7, 10).
- [33] M. Cavazza, F. Charles and S. J. Mead, 'Character-based interactive storytelling', *IEEE Intelligent systems*, vol. 17, no. 4, pp. 17–24, 2002. <http://tees.openrepository.com/tees/handle/10149/58294> (cit. on p. 10).
- [34] W. Choi, K. Sen, G. Necula and W. Wang, 'DetReduce: Minimizing Android GUI Test Suites for Regression Testing', in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 445–455. doi: 10.1145/3180155.3180173 (cit. on p. 6).

- [35] G. Claude, V. Gouranton and B. Arnaldi, ‘Versatile Scenario Guidance for Collaborative Virtual Environments’, in *Proceedings of 10th International Conference on Computer Graphics Theory and Applications*, ser. GRAPP’15, 2015. <https://hal-univ-rennes1.archives-ouvertes.fr/hal-01147733> (cit. on p. 10).
- [36] B. Combemale, R. France, J.-M. Jézéquel, B. Rumpe, J. Steel and D. Vojtisek, *Engineering modeling languages: Turning domain knowledge into tools*. Chapman and Hall/CRC, 2016 (cit. on pp. 8, 11, 37).
- [37] J. Coutaz, J. L. Crowley, S. Dobson and D. Garlan, ‘Context is key’, *Communications of the ACM*, vol. 48, no. 3, p. 49, Mar. 2005. doi: 10.1145/1047671.1047703 (cit. on p. 10).
- [38] F. Cuenca, K. Coninx, D. Vanacken and K. Luyten, ‘Graphical toolkits for rapid prototyping of multimodal systems: A survey’, *Interacting with Computers*, vol. 27, no. 4, pp. 470–488, 2014. doi: 10.1093/iwc/iwu003 (cit. on p. 6).
- [39] E. Czaplicki and S. Chong, ‘Asynchronous Functional Reactive Programming for GUIs’, in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’13, ACM, 2013, pp. 411–422. doi: 10.1145/2491956.2462161 (cit. on p. 47).
- [40] A. van Dam, ‘Post-WIMP user interfaces’, *Commun. ACM*, vol. 40, no. 2, pp. 63–67, 1997. doi: 10.1145/253671.253708 (cit. on p. 29).
- [41] B. Danglot, O. L. Vera-Pérez, B. Baudry and M. Monperrus, ‘Automatic Test Improvement with DSpot: a Study with Ten Mature Open-Source Projects’, *Empirical Software Engineering*, 2018. <https://arxiv.org/pdf/1811.08330> (cit. on pp. 29, 34).
- [42] B. Danglot, O. Vera-Perez, Z. Yu, M. Monperrus and B. Baudry, ‘A snowballing literature study on test amplification’, *Journal of Systems and Software*, 2019. <https://arxiv.org/abs/1705.10692> (cit. on pp. 32, 34).
- [43] C. Dea, M. Heckler, G. Grunwald, J. Pereda and S. Phillips, *JavaFX 8: Introduction by Example*. Apress, 2014 (cit. on p. 7).
- [44] T. Degueule, ‘Composition and interoperability for external domain-specific language engineering’, PhD thesis, Université Rennes 1, 2016. <https://tel.archives-ouvertes.fr/tel-01488300> (cit. on pp. 12, 22).
- [45] T. Degueule, B. Combemale, A. Blouin, O. Barais and J.-M. Jézéquel, ‘Mélange: A Meta-language for Modular and Reusable Development of DSLs’, in *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE’15, 2015, pp. 25–36. <https://hal.inria.fr/hal-01197038> (cit. on pp. 11, 12).
- [46] T. Degueule, B. Combemale, A. Blouin, O. Barais and J.-M. Jézéquel, ‘Safe Model Polymorphism for Flexible Modeling’, *Computer Languages, Systems and Structures*, vol. 49, p. 30, 2016. <https://hal.inria.fr/hal-01367305> (cit. on pp. 12, 13, 40).
- [47] S. W. Draper and D. A. Norman, ‘Software engineering for user interfaces’, *IEEE Transactions on Software Engineering*, vol. SE-11, no. 3, pp. 252–258, Mar. 1985. doi: 10.1109/TSE.1985.232208 (cit. on pp. 5, 6).
- [48] T. Duval, A. Blouin and J.-M. Jézéquel, ‘When Model Driven Engineering meets Virtual Reality: Feedback from Application to the Collaviz Framework’, in *Software Engineering and Architectures for Realtime Interactive Systems Working Group*, 2014. <https://hal.inria.fr/hal-00969072> (cit. on p. 24).

- [49] C. Ebert, G. Gallardo, J. Hernantes and N. Serrano, 'DevOps', *IEEE Software*, vol. 33, no. 03, pp. 94–100, May 2016. doi: [10.1109/MS.2016.68](https://doi.org/10.1109/MS.2016.68) (cit. on pp. 29, 31).
- [50] Eclipse, *Eclipse Sirius*. <https://www.eclipse.org/sirius/> (cit. on p. 39).
- [51] Facebook, *React*. <https://reactjs.org/> (cit. on p. 7).
- [52] L. P. Flannery, B. Silverman, E. R. Kazakoff, M. U. Bers, P. Bontá and M. Resnick, 'Designing scratchjr: Support for early childhood learning through computer programming', in *Proceedings of the 12th International Conference on Interaction Design and Children*, ser. IDC '13, ACM, 2013, pp. 1–10. doi: [10.1145/2485760.2485785](https://doi.org/10.1145/2485760.2485785) (cit. on p. 11).
- [53] F. Fleurey and A. Solberg, 'A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems', in *International Conference on Model Driven Engineering Languages and Systems*, ser. MoDELS'09, Springer, 2009, pp. 606–621. doi: [10.1007/978-3-642-04425-0_47](https://doi.org/10.1007/978-3-642-04425-0_47) (cit. on p. 10).
- [54] G. Foust, J. Järvi and S. Parent, 'Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems', in *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2015, ACM, 2015, pp. 121–130. doi: [10.1145/2814204.2814207](https://doi.org/10.1145/2814204.2814207) (cit. on p. 42).
- [55] M. Fowler, *Domain-specific languages*. Pearson Education, 2010 (cit. on p. 37).
- [56] M. Fowler, 'Language workbenches: The killer-app for domain specific languages', 2005, <https://www.martinfowler.com/articles/languageWorkbench.html> (cit. on p. 12).
- [57] R. France, B. Rumpe and M. Schindler, 'Why it is so hard to use models in software development: Observations', *Software & Systems Modeling*, vol. 12, no. 4, pp. 665–668, Oct. 2013. doi: [10.1007/s10270-013-0383-z](https://doi.org/10.1007/s10270-013-0383-z) (cit. on p. 38).
- [58] M. Frisch and R. Dachsel, 'Off-screen visualization techniques for class diagrams', in *Proceedings of the 5th International Symposium on Software Visualization*, ser. SOFTVIS'10, 2010, pp. 163–172. doi: [10.1145/1879211.1879236](https://doi.org/10.1145/1879211.1879236) (cit. on p. 38).
- [59] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995 (cit. on pp. 44, 45, 47).
- [60] Google, *Angular*. <https://angular.io/> (cit. on pp. 7, 35).
- [61] P. Gralka, C. Schulz, G. Reina, D. Weiskopf and T. Ertl, 'Visual exploration of memory traces and call stacks', in *2017 IEEE Working Conference on Software Visualization (VIS-SOFT)*, 2017, pp. 54–63. doi: [10.1109/VISSOFT.2017.15](https://doi.org/10.1109/VISSOFT.2017.15) (cit. on p. 40).
- [62] M. Green, 'A survey of three dialogue models', *ACM Trans. Graph.*, vol. 5, no. 3, pp. 244–275, 1986. doi: [10.1145/24054.24057](https://doi.org/10.1145/24054.24057) (cit. on p. 42).
- [63] M. L. Hammontree, J. J. Hendrickson and B. W. Hensley, 'Integrated data capture and analysis tools for research and testing on graphical user interfaces', in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '92, 1992, pp. 431–432. doi: [10.1145/142750.142886](https://doi.org/10.1145/142750.142886) (cit. on p. 29).

- [64] J. Hutchinson, J. Whittle and M. Rouncefield, ‘Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure’, *Science of Computer Programming*, vol. 89, pp. 144–161, 2014, Special issue on Success Stories in Model Driven Engineering. doi: [10.1016/j.scico.2013.03.017](https://doi.org/10.1016/j.scico.2013.03.017) (cit. on p. 38).
- [65] IFIP Working Group 2.7/13.4, <http://ui-engineering.org/mission/>, 2018 (cit. on p. 6).
- [66] C. Jeanneret, M. Glinz and B. Baudry, ‘Estimating Footprints of Model Operations’, in *International Conference on Software Engineering*, ser. ICSE’11, 2011, pp. 601–610. doi: [10.1145/1985793.1985875](https://doi.org/10.1145/1985793.1985875) (cit. on p. 11).
- [67] R. E. Johnson, ‘Frameworks = (components + patterns)’, *Commun. ACM*, vol. 40, no. 10, pp. 39–42, 1997. doi: [10.1145/262793.262799](https://doi.org/10.1145/262793.262799) (cit. on p. 44).
- [68] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc and G. Antoniol, ‘An exploratory study of the impact of antipatterns on class change- and fault-proneness’, *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012. doi: [10.1007/s10664-011-9171-y](https://doi.org/10.1007/s10664-011-9171-y) (cit. on pp. 33, 35).
- [69] G. E. Krasner, S. T. Pope *et al.*, ‘A description of the model-view-controller user interface paradigm in the smalltalk-80 system’, *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988 (cit. on pp. 10, 30, 42).
- [70] C. W. Krueger, ‘Software reuse’, *ACM Comput. Surv.*, vol. 24, no. 2, pp. 131–183, 1992. doi: [10.1145/130844.130856](https://doi.org/10.1145/130844.130856) (cit. on p. 44).
- [71] R. Lämmel, ‘Software chrestomathies’, *Science of Computer Programming*, vol. 97, pp. 98–104, 2015. doi: <https://doi.org/10.1016/j.scico.2013.11.014> (cit. on p. 39).
- [72] G. Le Moulec, ‘Synthèse d’applications de réalité virtuelle à partir de modèles’, PhD thesis, INSA de Rennes, 2018. <https://tel.archives-ouvertes.fr/tel-01959918> (cit. on pp. 10, 22).
- [73] G. Le Moulec, F. Argelaguet, V. Gouranton, A. Blouin and B. Arnaldi, ‘AGENT: Automatic Generation of Experimental Protocol Runtime’, in *ACM Symposium on Virtual Reality Software and Technology*, ser. VRST’17, 2017, pp. 1–10. <https://hal.archives-ouvertes.fr/hal-01613873> (cit. on p. 10).
- [74] G. Le Moulec, A. Blouin, V. Gouranton and B. Arnaldi, ‘Automatic Production of End User Documentation for DSLs’, *Computer Languages, Systems and Structures*, vol. 54, pp. 337–357, 2018. <https://hal.inria.fr/hal-01549042> (cit. on pp. 12, 13, 39).
- [75] V. L. Leitao, ‘Testing and maintenance of graphical user interfaces’, PhD thesis, INSA de Rennes, 2015. <https://tel.archives-ouvertes.fr/tel-01232388> (cit. on p. 22).
- [76] V. Lelli, A. Blouin and B. Baudry, ‘Classifying and Qualifying GUI Defects’, in *8th IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST’15, 2015, pp. 1–10. <https://hal.inria.fr/hal-01114724> (cit. on pp. 9, 13, 35).

- [77] V. Lelli, A. Blouin, B. Baudry and F. Coulon, 'On Model-Based Testing Advanced GUIs', in *11th Workshop on Advances in Model Based Testing (A-MOST 2015)*, ser. Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on, 2015, pp. 1–10. <https://hal.inria.fr/hal-01123647> (cit. on pp. 9, 34).
- [78] V. Lelli, A. Blouin, B. Baudry, F. Coulon and O. Beaudoux, 'Automatic Detection of GUI Design Smells: The Case of Blob Listener', in *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS'16, 2016, pp. 263–274. doi: [10.1145/2933242.2933260](https://doi.org/10.1145/2933242.2933260). <https://hal.inria.fr/hal-01308625> (cit. on pp. 9, 32, 35).
- [79] K. Lemon, E. B. Allen, J. C. Carver and G. L. Bradshaw, 'An empirical study of the effects of gestalt principles on diagram understandability', in *First International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM'07, 2007, pp. 156–165. doi: [10.1109/ESEM.2007.37](https://doi.org/10.1109/ESEM.2007.37) (cit. on p. 38).
- [80] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon and V. López-Jaquero, 'USIXML: a language supporting multi-path development of user interfaces', in *International Workshop on Design, Specification, and Verification of Interactive Systems*, Springer, 2004, pp. 200–220. doi: [10.1007/11431879_12](https://doi.org/10.1007/11431879_12) (cit. on pp. 6, 7, 10).
- [81] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto and D. Poshyanyk, 'API change and fault proneness: a threat to the success of Android apps', in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE'13, 2013, p. 477. doi: [10.1145/2491411.2491428](https://doi.org/10.1145/2491411.2491428) (cit. on p. 31).
- [82] Q. Luo, F. Hariri, L. Eloussi and D. Marinov, 'An empirical analysis of flaky tests', *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pp. 643–653, 2014. doi: [10.1145/2635868.2635920](https://doi.org/10.1145/2635868.2635920) (cit. on p. 33).
- [83] I. Maier and M. Odersky, 'Deprecating the Observer Pattern with Scala.React', Tech. Rep., 2012. <https://infoscience.epfl.ch/record/176887> (cit. on p. 42).
- [84] A. M. Memon, 'An Event-flow Model of GUI-Based Applications for Testing', *Software Testing, Verification & Reliability*, vol. 17, no. 3, pp. 137–157, 2007. doi: <https://doi.org/10.1002/stvr.364> (cit. on pp. 32, 33).
- [85] A. M. Memon, 'GUI testing: Pitfalls and process', *Computer*, no. 8, pp. 87–88, 2002. <https://www.computer.org/csdl/mags/co/2002/08/r8087.pdf> (cit. on p. 6).
- [86] D. A. Méndez-Acuña, J. A. Galindo, B. Combemale, A. Blouin and B. Baudry, 'Reverse Engineering Language Product Lines from Existing DSL Variants', *Journal of Systems and Software*, vol. 133, pp. 145–158, 2017. <https://hal.inria.fr/hal-01524632> (cit. on p. 12).
- [87] D. Méndez-Acuña, J. A. Galindo Duarte, B. Combemale, A. Blouin and B. Baudry, 'Puzzle: A tool for analyzing and extracting specification clones in DSLs', in *the 15th International Conference on Software Reuse*, ser. ICSR'16, 2016. <https://hal.archives-ouvertes.fr/hal-01284822> (cit. on p. 12).

- [88] D. Méndez-Acuña, J. A. Galindo Duarte, B. Combemale, A. Blouin, B. Baudry and G. Le Guernic, ‘Reverse-engineering reusable language modules from legacy domain-specific languages’, in *the 15th International Conference on Software Reuse*, ser. ICSR’16, 2016. <https://hal.archives-ouvertes.fr/hal-01284816> (cit. on p. 12).
- [89] M. Mernik, J. Heering and A. M. Sloane, ‘When and How to Develop Domain-Specific Languages’, *ACM Computing Surveys*, vol. 37, pp. 316–344, 2005. doi: [10.1145/1118890.1118892](https://doi.org/10.1145/1118890.1118892) (cit. on p. 8).
- [90] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield and S. Krishnamurthi, ‘Flapjax: A Programming Language for Ajax Applications’, in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA ’09, ACM, 2009, pp. 1–20. doi: [10.1145/1640089.1640091](https://doi.org/10.1145/1640089.1640091) (cit. on p. 45).
- [91] Z. Mijailovic and D. Milicev, ‘A Retrospective on User Interface Development Technology’, *Software, IEEE*, vol. 30, pp. 76–83, 2013. doi: [10.1109/MS.2013.45](https://doi.org/10.1109/MS.2013.45) (cit. on p. 5).
- [92] B. Morin, O. Barais, G. Nain and J.-M. Jezequel, ‘Taming dynamically adaptive systems using models and aspects’, in *Proceedings of the 31st International Conference on Software Engineering*, IEEE Computer Society, 2009, pp. 122–132. <https://hal.archives-ouvertes.fr/inria-00468516/> (cit. on p. 10).
- [93] Mozilla, *MDN Web Docs*. <https://developer.mozilla.org> (cit. on p. 7).
- [94] L. Murphy, M. B. Kery, O. Alliyu, A. Macvean and B. A. Myers, ‘API Designers in the Field: Design Practices and Challenges for Creating Usable APIs’, in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing*, ser. VL/HCC’18, 2018, pp. 249–258. doi: [10.1109/VLHCC.2018.8506523](https://doi.org/10.1109/VLHCC.2018.8506523) (cit. on p. 6).
- [95] G. Mussbacher, D. Amyot, R. Breu, J.-M. Bruel, B. H. Cheng, P. Collet, B. Combemale, R. B. France, R. Heldal, J. Hill *et al.*, ‘The relevance of model-driven engineering thirty years from now’, in *International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS’14, 2014, pp. 183–200. doi: [10.1007/978-3-319-11653-2_12](https://doi.org/10.1007/978-3-319-11653-2_12) (cit. on p. 38).
- [96] B. A. Myers, A. J. Ko, T. D. LaToza and Y. Yoon, ‘Programmers are users too: Human-centered methods for improving programming tools’, *Computer*, vol. 49, no. 7, pp. 44–52, Jul. 2016. doi: [10.1109/MC.2016.200](https://doi.org/10.1109/MC.2016.200) (cit. on p. 12).
- [97] B. A. Myers, ‘A new model for handling input’, *ACM Trans. Inf. Syst.*, vol. 8, no. 3, pp. 289–320, Jul. 1990. doi: [10.1145/98188.98204](https://doi.org/10.1145/98188.98204) (cit. on p. 45).
- [98] B. A. Myers, ‘Separating application code from toolkits: Eliminating the spaghetti of call-backs’, in *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST ’91, ACM, 1991, pp. 211–220. doi: [10.1145/120782.120805](https://doi.org/10.1145/120782.120805) (cit. on pp. 42, 45).
- [99] D. Navarre, P. Palanque, J.-F. Ladry and E. Barboni, ‘ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability’, *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 16, no. 4, p. 18, 2009. doi: [10.1145/1614390.1614393](https://doi.org/10.1145/1614390.1614393) (cit. on pp. 6, 47).

- [100] Y. Ndiaye, O. Barais, A. Blouin, A. Bouabdallah and N. Aillery, ‘Requirements for preventing logic flaws in the authentication procedure of web applications’, in *The 34th ACM/SIGAPP Symposium On Applied Computing*, ser. SAC’19, 2019. <https://hal.inria.fr/hal-02087663>.
- [101] L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002 (cit. on p. 10).
- [102] R. A. P. Oliveira, E. Alégroth, Z. Gao and A. Memon, ‘Definition and evaluation of mutation operators for GUI-level mutation analysis’, in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW’15, 2015, pp. 1–10. doi: [10.1109/ICSTW.2015.7107457](https://doi.org/10.1109/ICSTW.2015.7107457) (cit. on p. 35).
- [103] OMG, *UML Specification*, 2007. <https://www.omg.org/spec/UML/> (cit. on pp. 11, 38).
- [104] S. Oney, B. Myers and J. Brandt, ‘ConstraintJS: programming interactive behaviors for the web by integrating constraints and states’, in *Proceedings of the 25th annual ACM symposium on User interface software and technology*, ser. UIST ’12, ACM, 2012, pp. 229–238. doi: [10.1145/2380116.2380146](https://doi.org/10.1145/2380116.2380146) (cit. on p. 45).
- [105] S. Oney, B. Myers and J. Brandt, ‘Interstate: Interaction-oriented language primitives for expressing GUI behavior’, in *Proceedings of the 27th annual ACM symposium on User interface software and technology*, ser. UIST ’14, ACM, 2014, pp. 10–1145. doi: [10.1145/2642918.2647358](https://doi.org/10.1145/2642918.2647358) (cit. on pp. 42, 43, 45).
- [106] R. F. Paige, ‘Language engineering: Challenges, opportunities and potential disasters for interactive systems’, in *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS ’16, ACM, 2016, pp. 3–3. doi: [10.1145/2933242.2948132](https://doi.org/10.1145/2933242.2948132) (cit. on p. 29).
- [107] R. F. Paige and L. M. Rose, ‘Lies, Damned Lies and UML2Java’, *Journal of Object Technology*, vol. 12, no. 1, 2013. doi: [10.5381/jot.2013.12.1.c1](https://doi.org/10.5381/jot.2013.12.1.c1) (cit. on pp. 37, 38).
- [108] P. Palanque, *Engineering interactive critical systems – ACM lectures*, <https://speakers.acm.org/lectures/6824>, 2018 (cit. on p. 7).
- [109] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk and A. De Lucia, ‘Mining version histories for detecting code smells’, *IEEE Transactions on Software Engineering*, 2014. doi: [10.1109/TSE.2014.2372760](https://doi.org/10.1109/TSE.2014.2372760) (cit. on p. 35).
- [110] F. Palomba and A. Zaidman, ‘Does refactoring of test smells induce fixing flaky tests?’, in *2017 IEEE International Conference on Software Maintenance and Evolution*, ser. IC-SME’17, 2017, pp. 1–12. doi: [10.1109/ICSME.2017.12](https://doi.org/10.1109/ICSME.2017.12) (cit. on pp. 32, 36).
- [111] D. L. Parnas, ‘On the criteria to be used in decomposing systems into modules’, *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972. doi: [10.1145/361598.361623](https://doi.org/10.1145/361598.361623) (cit. on pp. 37, 44).
- [112] A. Pleuss, S. Wollny and G. Botterweck, ‘Model-driven development and evolution of customized user interfaces’, in *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS ’13, ACM, 2013, pp. 13–22. doi: [10.1145/2494603.2480298](https://doi.org/10.1145/2494603.2480298) (cit. on p. 10).

- [113] K. Pohl, G. Böckle and F. J. van Der Linden, *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005 (cit. on p. 10).
- [114] M. Potel, 'MVP: Model-View-Presenter the Taligent programming model for C++ and Java', *Taligent Inc*, 1996 (cit. on p. 10).
- [115] J. Roche, 'Adopting DevOps practices in quality assurance', *Commun. ACM*, vol. 56, no. 11, pp. 38–43, 2013. doi: [10.1145/2538031.2540984](https://doi.org/10.1145/2538031.2540984) (cit. on pp. 29, 31).
- [116] B. Rumpe and R. France, 'On the relationship between modeling and programming languages', *Software and Systems Modeling*, vol. 11, no. 1, pp. 1–2, 2012. <http://www.springerlink.com/index/y1126331504h4612.pdf> (cit. on pp. 37, 38).
- [117] G. Salvaneschi, S. Amann, S. Proksch and M. Mezini, 'An empirical study on program comprehension with reactive programming', in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, ACM, 2014, pp. 564–575. doi: [10.1145/2635868.2635895](https://doi.org/10.1145/2635868.2635895) (cit. on p. 42).
- [118] G. Salvaneschi and M. Mezini, 'Towards reactive programming for object-oriented applications', *Transactions on Aspect-Oriented Software Development XI*, vol. 8400, pp. 227–261, 2014. doi: [10.1007/978-3-642-55099-7_7](https://doi.org/10.1007/978-3-642-55099-7_7) (cit. on p. 42).
- [119] D. J. Sheskin, *Handbook Of Parametric And Nonparametric Statistical Procedures, Fourth Edition*. Chapman & Hall/CRC, Jan. 2007 (cit. on pp. 16, 17).
- [120] F. Shull, J. Singer and D. I. Sjøberg, *Guide to advanced empirical software engineering*. Springer, 2007 (cit. on p. 14).
- [121] J. Smith, 'WPF Apps With The Model-View-ViewModel Design Pattern', *MSDN Magazine*, Feb. 2009. <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx> (cit. on pp. 7, 10).
- [122] N. Souchon and J. Vanderdonckt, 'A Review of XML-compliant User Interface Description Languages', in *Interactive Systems. Design, Specification, and Verification*, ser. DSV-IS'03, Springer Berlin Heidelberg, 2003, pp. 377–391. doi: [10.1007/978-3-540-39929-2_26](https://doi.org/10.1007/978-3-540-39929-2_26) (cit. on p. 7).
- [123] D. Steinberg, F. Budinsky, E. Merks and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008 (cit. on pp. 11, 39).
- [124] W. Sun, B. Combemale, R. B. France, A. Blouin, B. Baudry and I. Ray, 'Using Slicing to Improve the Performance of Model Invariant Checking', *The Journal of Object Technology*, p. 28, 2015. <https://hal.inria.fr/hal-01179369> (cit. on pp. 11, 26).
- [125] F. Tip, 'A survey of program slicing techniques', *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995. <https://www.franktip.org/pubs/jpl1995.pdf> (cit. on p. 11).
- [126] UsiXML-Consortium, 'UsiXML, USeR Interface eXtensible Markup Language', UsiXML Consortium, Tech. Rep., 2007. <http://www.usixml.org> (cit. on p. 7).
- [127] A. Van Deursen and P. Klint, 'Domain-specific language design requires feature descriptions', *CIT. Journal of computing and information technology*, vol. 10, no. 1, pp. 1–17, 2002. doi: [10.2498/cit.2002.01.01](https://doi.org/10.2498/cit.2002.01.01) (cit. on p. 39).

- [128] J. Vanderdonckt, 'Model-Driven Engineering of User Interfaces: Promises, Successes, and Failures', in *Proceedings of 5th Annual Romanian Conference on Human-Computer Interaction*, ser. ROCHI'08, 2008. https://dial.uclouvain.be/downloader/downloader.php?pid=boreal:118090&datastream=PDF_01 (cit. on pp. 6, 29).
- [129] R. Wei, D. S. Kolovos, A. Garcia-Dominguez, K. Barmpis and R. F. Paige, 'Partial Loading of XMI Models', in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS'16, ACM, 2016, pp. 329–339. doi: [10.1145/2976767.2976787](https://doi.org/10.1145/2976767.2976787) (cit. on p. 11).
- [130] J. Whittle, J. Hutchinson and M. Rouncefield, 'The state of practice in model-driven engineering', *IEEE Software*, vol. 31, no. 3, pp. 79–85, May 2014. doi: [10.1109/MS.2013.65](https://doi.org/10.1109/MS.2013.65) (cit. on pp. 37, 38).
- [131] N. Winston, 'Catching bugs earlier: the unexpected benefits of automating GUI testing', in *Fifth International Software Quality Week, San Fransisco, USA*, 1992 (cit. on p. 6).
- [132] A. Yamashita and L. Moonen, 'Do code smells reflect important maintainability aspects?', in *IEEE International Conference on Software Maintenance*, ser. ICSM'12, 2012, pp. 306–315. doi: [10.1109/ICSM.2012.6405287](https://doi.org/10.1109/ICSM.2012.6405287) (cit. on p. 33).
- [133] W. Yuan, H. H. Nguyen, L. Jiang, Y. Chen, J. Zhao and H. Yu, 'API recommendation for event-driven Android application development', *Information and Software Technology*, vol. 107, pp. 30–47, 2019. doi: [10.1016/j.infsof.2018.10.010](https://doi.org/10.1016/j.infsof.2018.10.010) (cit. on p. 47).
- [134] X. Yuan, M. B. Cohen and A. M. Memon, 'GUI Interaction Testing: Incorporating Event Context', *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 559–574, Jul. 2011. doi: [10.1109/TSE.2010.50](https://doi.org/10.1109/TSE.2010.50) (cit. on p. 32).