Addressing Fine-Grained Variability in User-Centered Software Product Lines: A Case Study on Dashboards

Andrea Vázquez-Ingelmo[1](✉), Francisco J. García-Peñalvo[1], and Roberto Therón[1,2]

[1] GRIAL Research Group, Computer Sciences Department, Research Institute for Educational Sciences, University of Salamanca, Salamanca, Spain
{andreavazquez,Fgarcia,theron}@usal.es

[2] VisUSAL Research Group, University of Salamanca, Salamanca, Spain

Abstract. Software product lines provide a theoretical framework to generate and customize products by studying the target domain and by capturing the commonalities among the potential products of the family. This domain knowledge is subsequently used to implement a series of configurable core assets that will be systematically reused to obtain products with different features to match particular user requirements. Some kind of interactive systems, like dashboards, require special attention as their features are very fine-grained. Having the capacity of configuring a dashboard product to match particular user requirements can improve the utility of these products by providing the support to users to reach useful insights, in addition to a decrease in the development time and an increase in maintainability. Several techniques for implementing features and variability points in the context of SPLs are available, and it is important to choose the right one to exploit the SPL paradigm benefits to the maximum. This work addresses the materialization of fine-grained variability in SPL through code templates and macros, framed in the particular domain of dashboards.

Keywords: Software product lines · SPL · Granularity · User interfaces · Dashboards · Customization

1 Introduction

Software product lines (SPLs) address the systematic development of software assets for building families of products that share a specific domain [1, 2]. By reutilizing, configuring and composing these software assets, the time-to-market of new derived products decreases, in addition to an increase in requirements traceability, customization levels, flexibility, maintainability and of course, productivity.

However, implementing and introducing an SPL is not a straightforward job. The domain in which the SPL will be framed must be thoroughly studied to extract significant features and capture the commonalities among the potential products that could be developed through this paradigm. Planning the development of highly configurable software components allows the delay of design decisions, enhancing flexibility regarding the materialization of dynamic or even new requirements. These delayed design decisions are the so-called variability points [3].

The study of the target domain is the first step regarding an SPL design process, but the implementation of the identified variability points within the core assets of the product family remains a crucial and a critical challenge for this paradigm to succeed.

There are several techniques to materialize variability points, and the desired granularity of the SPL features is a relevant factor to choose the right method referring to the ability to modify the products behavior or their underlying functionality. In addition to the desired granularity level, the target domain of the SPL is also a key factor regarding the choice of the implementation technique.

For instance, user-centered tools require high levels of customization, both at functional and at visual design level. Developing these type of tools need further efforts on the requirements elicitation processes, in order to fully understand the final users' necessities and to provide them with helpful interfaces. Customizing user interfaces within the SPL paradigm context, however, is still a complex task, yet requiring semi-automatic or completely manual processes [4]. A large number of possible user profiles (and their associated particular requirements) could make the automatic derivation of interactive systems chaotic regarding its possible features, hampering the evolution and maintainability of the product line. The main issue regarding these interaction-intensive systems is the fine-grained nature of their features: a slight modification on interaction patterns, interface layout, color palette, etc. could be crucial regarding the final perceived usability of a generated product.

A particular case of these interactive systems is dashboards. These tools aim at helping users to reach useful insights about datasets, facilitating the discovery of unusual patterns or significant data points. The potential of dashboards resides in their ability to present information at-a-glance, supporting complex procedures like decision-making processes, communication, and learning, etc. [5]. A lot of profiles could be involved in these procedures though, being difficult to provide a common and general dashboard useful for each of them. That is why the SPL paradigm can ease the development of customized dashboards by reutilizing its different components (i.e., visualizations, controls, filters, interaction patterns, etc.), instead of implementing a single dashboard for each data domain or user involved. However, dashboards need fine-grained variability to provide powerful customizations and to support particular configurations for different user profiles, helping them to reach their own goals regarding data exploration and data explanation.

The remainder of this paper is structured as follows. Section 2 is an overview of a set of available methods for implementing variability within SPLs. Section 3 analyzes the particular case of the dashboards domain regarding the granularity of its features, presenting the case study in which an experimental framework for generating dashboards has been developed in Sect. 4. Finally, Sect. 5 discusses the achieved results regarding granularity, and Sect. 6 presents the conclusions of this work.

2 Variability Mechanisms

There exist different techniques to implement variability points in SPLs. It is important to choose wisely given the requirements of the product line itself (i.e., the complexity of the software to develop, its number of features, their granularity requirements, etc.). Generally, at the code level, the variability points that correspond to a specific feature will be spread across different source files [6]. That is why separating concerns at the implementation level is essential to avoid the variability points to be scattered, as this feature dispersion would decrease code understandability and maintainability. Implementing each feature in individual code modules can help with this separation of concerns [6], but it is difficult to achieve fine-grained variability through this approach. A balance between code understandability and granularity should be devised to choose both a maintainable and highly customizable SPL.

This section will briefly describe different methods that are potentially suitable to the dashboards' domain given their particular features, although there are more approaches to implement variability in SPLs that can be consulted in [6].

2.1 Conditional Compilation

Conditional compilation uses preprocessor directives to inject or remove code fragments from the final product source code. This method allows the achievement of any level of feature granularity due to the possibility of inserting these directives at any point of the code, even at expression or function signature level [7]. Also, although pretended to the C language, preprocessor directives can be used for any language and arbitrary transformations [8]. The main drawback of this approach is the decrease of code readability and understandability as interweaving and nesting these preprocessor directives makes the code maintainability a tedious task [9].

2.2 Frames

Frame technology is based on entities (frames) that are assembled to compose final source code files. Frames use preprocessor-like directives to insert or replace code and to set parameters [6]. An example of a variability implementation method based on frame technology is the XML-based Variant Configuration Language (XVCL) [10]. Through this approach, only the necessary code is introduced in concrete components by specifying frames that contain the code and directives associated with different features and variants. XVCL is independent of the programming language and can handle variability at any granularity level [11].

2.3 Template Engines

Template engines allow the parameterization and inclusion/exclusion of code fragments through different directives. If the template engine allows the definition of macros, features can be refactored into different code fragments encapsulated through these elements, improving the code organization and enabling variability at any level of granularity. Templating engines can also be language-independent, providing a powerful tool for generating any type of source file [12] by using programming directives such as loops and conditions.

2.4 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) allows the implementation of crosscutting concerns through the definition of aspects, centralizing features that need to be present in different source files through unique entities (aspects) thus improving code understandability and maintainability by avoiding scattered features and "tangled" code [13].

AOP is a popular method to materialize variability points in SPLs due to the possibility of modifying the system behavior at certain points, namely join points [14–16]. However, AOP could lack fine-grained variability (i.e., variability at sentence, expression or signature level) and particular frameworks or language extensions are necessary to implement aspects in certain programming languages.

3 The Dashboards' Domain

Regarding the present work target domain (i.e., dashboards), the chosen implementation technique was to use a template engine. The decision was made due to the fine granularity that can be achieved through this method, which is necessary to materialize even the slightest variability on the visualization components. Another factor for choosing this technique lies in the straightforward way of implementing variability regarding the products' features and its language-independent nature.

Framing technology could also be a potential solution within the dashboards' domain, but the decision of designing a DSL to wrap the features at a higher level made the use of code templates a more suitable solution, providing complete freedom to define the syntax of the DSL (specification x-frames are based on a fixed syntax [11], which could result in lack of flexibility for this work's approach) as the directives within the templates can be fully parameterized.

The selected template engine was Jinja2 (http://jinja.pocoo.org/docs/2.10/) given its rich API and powerful features such as the possibility of defining macros, importing them, defining custom filters and tags in addition to its available basic directives (loops, conditions, etc.).

4 Results of the Case Study

As it has been aforementioned, a DSL has been designed along with the SPL to abstract and ease the application engineering process. This DSL binds the feature model with the implementation method at code-level [17], enabling the specification of features through XML technology. Designing a DSL not only eases the configuration of variants but also improves the traceability of features through the different SPL paradigm phases (and opens up the possibility of combining the SPL paradigm with model-driven development [16]).

For this case study, it is necessary to provide a configurable SPL that enable automatic generation of dashboards with different features. These features involve a variety of potential requirements: from the modification of the dashboard layout (i.e., including or removing whole visualization components) to the modification of a particular interaction pattern to manage to zoom on visualizations, for example. To achieve the desired levels of granularity and to support the DSL for automating the application engineering phase, a template engine (Jinja2) was selected, as indicated in Sect. 3.

Templating resembles conditional compilation, as their underlying behavior based in programming directives is very similar. The main benefit of templates is that they support these directives and macros in a more sophisticated manner.

As presented in Sect. 2, the main drawback of conditional compilation is the scatter of concerns and features, decreasing code maintainability and readability. One of the benefits of using a powerful template engine like Jinja2 is the possibility of clustering the necessary code fragments that compose a certain feature in sets of macros. This approach improves maintainability, as the code fragments in charge of the features will be contained and organized in associated files.

The practical approach followed to apply this implementation method is exemplified in the remainder of this section.

Figure 1 shows a high-level view of the feature model for the dashboard product line developed for the Spanish Observatory for University Employability and Employment (OEEU, https://oeeu.org) [18, 19] to allow users to explore and reach insights about the data collected by this organization [20–25]. The generated dashboard can have different pages, each one composed of different visualizations and data filters. At this high-level view, features are coarse-grained; whole components can be included or removed from the final generated dashboard.

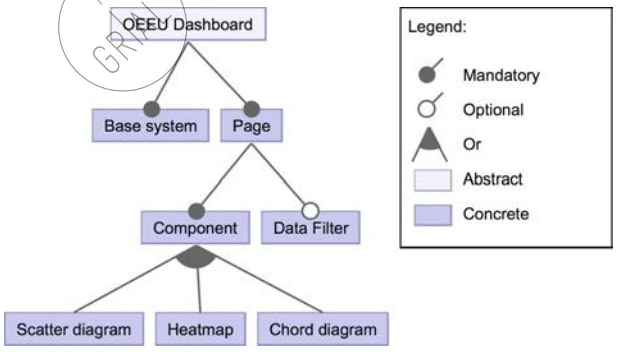


Fig. 1. High-level view of the dashboard SPL's feature diagram.

Low-level features (i.e., leaf nodes of the feature diagram) require fine-grained granularity within the dashboard domain, as these features concern minor visual, functional or interaction characteristics. Figure 2 shows low-level features for a scatter diagram component about the possible functionalities related to its data and behavior.

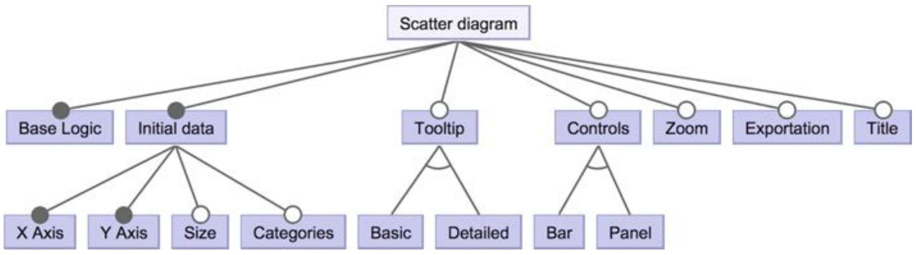
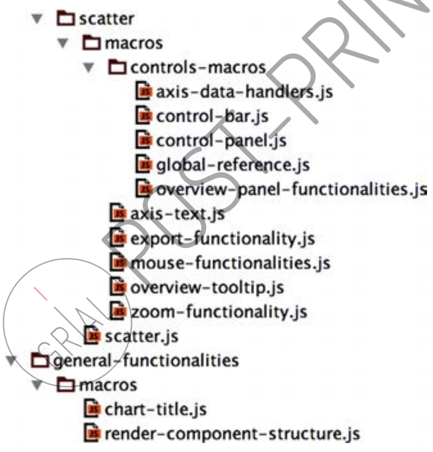


Fig. 2. A snippet of the feature diagram showing lower-level features regarding a scatter diagram component. Some of these features (e.g., the "controls" feature) have their own subsequent features to provide higher customization levels regarding the visualization's functional and information requirements.

To materialize these features at code-level, each feature is arranged in its own file and each file is composed with a set of macros Fig. 3. This set of macros contains the required code fragments associated with an SPL feature.

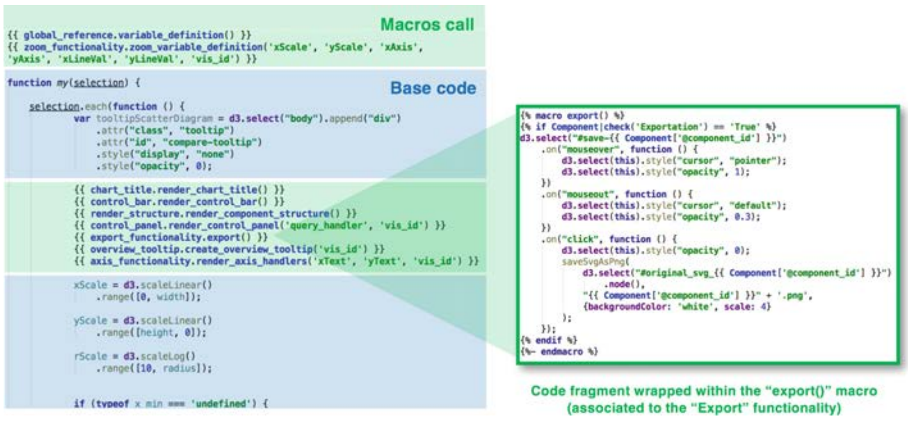


```
▼ scatter
  ▼ macros
    ▼ controls-macros
        axis-data-handlers.js
        control-bar.js
        control-panel.js
        global-reference.js
        overview-panel-functionalities.js
      axis-text.js
      export-functionality.js
      mouse-functionalities.js
      overview-tooltip.js
      zoom-functionality.js
    scatter.js
▼ general-functionalities
  ▼ macros
      chart-title.js
      render-component-structure.js
```

Fig. 3. Example of the code templates organization.

The macros calls are executed within the base logic of the component (in this case the "scatter.js" file contains the basic logic for the scatter diagram, which is mandatory and common for all possible product derivations, as specified in the feature diagram).

The macros themselves are affected by the conditional directives in charge of adapting the code giving particular configurations. This means that the base code will only contain the macro calls, delegating the condition check to the macros and making the code cleaner. By using this approach at the implementation level, concerns are not continuously scattered through the code as it could happen with pure preprocessor directives Fig. 4.

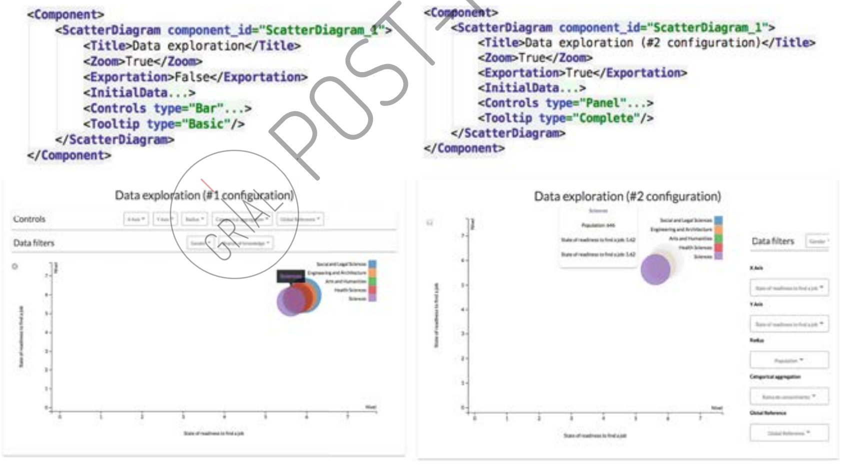

Macros call
Base code
Code fragment wrapped within the "export()" macro
(associated to the "Export" functionality)

```
{{ global_reference.variable_definition() }}
{{ zoom_functionality.zoom_variable_definition('xScale', 'yScale', 'xAxis',
'yAxis', 'xLineVal', 'yLineVal', 'vis_id') }}

function my(selection) {

    selection.each(function () {
            var tooltipScatterDiagram = d3.select("body").append("div")
                .attr("class", "tooltip")
                .attr("id", "compare-tooltip")
                .style("display", "none")
                .style("opacity", 0);

            {{ chart_title.render_chart_title() }}
            {{ control_bar.render_control_bar() }}
            {{ render_structure.render_component_structure() }}
            {{ control_panel.render_control_panel('query_handler', 'vis_id') }}
            {{ export_functionality.export() }}
            {{ overview_tooltip.create_overview_tooltip('vis_id') }}
            {{ axis_functionality.render_axis_handlers('xText', 'yText', 'vis_id') }}

            xScale = d3.scaleLinear()
                .range([0, width]);

            yScale = d3.scaleLinear()
                .range([height, 0]);

            rScale = d3.scaleLog()
                .range([10, radius]);


            if (typeof x_min === 'undefined') {
```

```
{% macro export() %}
{% if Component|check('Exportation') == 'True' %}
d3.select("#save-{{ Component['@component_id'] }}")
    .on("mouseover", function () {
        d3.select(this).style("cursor", "pointer");
        d3.select(this).style("opacity", 1);
    })
    .on("mouseout", function () {
        d3.select(this).style("cursor", "default");
        d3.select(this).style("opacity", 0.3);
    })
    .on("click", function () {
        d3.select(this).style("opacity", 0);
        saveSvgAsPng(
            d3.select("#original_svg_{{ Component['@component_id'] }}")
                .node(),
            "{{ Component['@component_id'] }}" + '.png',
            {backgroundColor: 'white', scale: 4}
        );
    });
{% endif %}
{%- endmacro %}
```

Fig. 4. A snippet of the scatter diagram's JavaScript code. The base code (highlighted in blue) contains macro calls (highlighted in green). If the condition wrapped within the macro is matched, the associated code is injected (i.e., the associated feature will be supported).

Through the DSL and the code templates, a custom code generator can build personalized dashboards that meet the specified requirements automatically Fig. 5.



```
<Component>
    <ScatterDiagram component_id="ScatterDiagram_1">
        <Title>Data exploration</Title>
        <Zoom>True</Zoom>
        <Exportation>False</Exportation>
        <InitialData...>
        <Controls type="Bar"...>
        <Tooltip type="Basic"/>
    </ScatterDiagram>
</Component>
```

```
<Component>
    <ScatterDiagram component_id="ScatterDiagram_1">
        <Title>Data exploration (#2 configuration)</Title>
        <Zoom>True</Zoom>
        <Exportation>True</Exportation>
        <InitialData...>
        <Controls type="Panel"...>
        <Tooltip type="Complete"/>
    </ScatterDiagram>
</Component>
```

Data exploration (#1 configuration)
Controls
Data filters
Data exploration (#2 configuration)
Data filters

Fig. 5. Two different scatter diagram configurations achieved through the DSL (on top). As it can be seen, the tooltip type, for example, provides different behaviors when interacting with the visualization elements. Also, the layout of the whole visualization can be modified

5 Discussion

SPLs have proved to be a powerful paradigm for managing particular sets of requirements in an efficient and maintainable way. However, these requirements could need different granularity levels, as some important features could be coarse-grained while others could be fine-grained. Choosing the right implementation technique is a complex task because several factors must be taken into account: the levels of granularity, the understandability, and maintainability of the code, the viability of the technique, etc. This work addresses fine-grained granularity in a SPL of dashboards. Dashboards are key tools for reaching of insights regarding particular datasets and to support decision-making processes. Having the power of customizing their features at fine-grained level could be highly valuable, as dashboards usually ask to be user-tailored to provide useful support for particular and individual goals.

In the presented case study, a DSL has been designed for abstracting the configuration process. The use of this DSL to feed a code generator has been one of the determining factors to choose a template engine as the implementation method of the SPL's variability points. Although this approach still lacks powerful maintainability levels, it maintains a proper requirements' traceability by arranging features in a variety of macro definitions. Using XVCL [10] could have been another solution to manage these fine-grained features, but the decision of wrapping the SPL specification through a DSL asked for a more flexible and customizable method such as a template engine. What is more, a combination of the AOP paradigm with the templating method could be highly beneficial providing both customizations regarding directives and a better technique to manage crosscutting concerns (an issue that a template engine could not solve straightforwardly). Also, the approach asks for a method to address data heterogeneity in order to visualize data from any kind of source. However, although presenting these caveats, the results are promising and prove that a robust template engine could be a beneficial method to materialize fine-grained variability within the SPL paradigm context.

Regarding the application on the dashboard domain, having a dashboard SPL could address several problems related to individual personalization, meeting particular requirements. This approach could provide tailored dashboards efficiently after an in-depth elicitation of requirements without consuming many resources, avoiding overwhelming configuration processes delegated to end-users themselves [26].

6 Conclusions

Dashboards are sophisticated tools that require fine-grained features to offer valuable user experiences to their target users. A template-based approach to implement variability points at code level has been applied to an SPL of dashboards.

Creating an SPL of dashboards is not a straightforward task, as different variability dimensions are involved (variability regarding visual design, functionality, layout, data sources, etc.). Using a template engine to implement the core assets of the SPL can address the mentioned fine-grained variability and increase the traceability of features.

This SPL paradigm application to the dashboards' domain opens up different research paths, such as experimenting with different fine-grained configurations to find the best configuration for a particular user profile (A/B testing [23, 27]) or applying machine learning or knowledge bases [28] to provide potentially suitable configurations automatically given certain contexts or user characteristics. Also, developing an automatic link between the feature diagram and the DSL, as well between the DSL and the code templates' directives could further improve maintainability and traceability.

Acknowledgements. This research work has been supported by the Spanish *Ministry of Education and Vocational Training* under an FPU fellowship (FPU17/03276). This work has been partially funded by the Spanish Government Ministry of Economy and Competitiveness throughout the DEFINES project (Ref. TIN2016-80172-R) and the PROVIDEDH project, funded within the CHIST-ERA Programme under the national grant agreement: PCIN-2017-064 (MINECO, Spain).

References

1. Clements, P., Northrop, L.: Software Product Lines. Addison-Wesley, Boston (2002)
2. Pohl, K., Böckle, G., Linden, van Der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, New York (2005)
3. Van Gurp, J., Bosch, J., Svahnberg, M.: On the notion of variability in software product lines. In: 2001. Proceedings. Working IEEE/IFIP Conference on Software Architecture, pp. 45–54. IEEE (2001)
4. Pleuss, A., Hauptmann, B., Keunecke, M., Botterweck, G.: A case study on variability in user interfaces. In: Proceedings of the 16th International Software Product Line Conference, vol. 1, pp. 6–10. ACM (2012)
5. Sarikaya, A., Correll, M., Bartram, L., Tory, M., Fisher, D.: What do we talk about when we talk about dashboards? IEEE Trans. Visual. Comput. Graph (2018)
6. Gacek, C., Anastasopoules, M.: Implementing product line variabilities. In: ACM SIGSOFT Software Engineering Notes, pp. 109–117. ACM (2001)
7. Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M.: An analysis of the variability in forty preprocessor-based software product lines. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, pp. 105–114. ACM (2010)
8. Favre, J.-M.: Preprocessors from an abstract point of view. In: Proceedings of the Third Working Conference on Reverse Engineering 1996, pp. 287–296. IEEE (1996)
9. Spencer, H., Collyer, G.: #ifdef considered harmful, or portability experience with C News (1992)
10. Jarzabek, S., Bassett, P., Zhang, H., Zhang, W.: XVCL: XML-based variant configuration language. In: Proceedings of the 25th International Conference on Software Engineering, pp. 810–811. IEEE Computer Society (2003)
11. Zhang, H., Jarzabek, S., Swe, S.M.: XVCL approach to separating concerns in product family assets. In: International Symposium on Generative and Component-Based Software Engineering, pp. 36–47. Springer, Heidelberg (2001)
12. Cisco Blogs. https://blogs.cisco.com/developer/network-configuration-template
13. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: European Conference on Object-Oriented Programming, pp. 220–242. Springer, Heidelberg (1997)

14. Waku, G.M., Rubira, C.M., Tizzei, L.P.: A case study using AOP and components to build software product lines in android platform. In: 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 418–421. IEEE (2015)
15. Heo, S.-h., Choi, E.M.: Representation of variability in software product line using aspect-oriented programming. In: Fourth International Conference on Software Engineering Research, Management and Applications, 2006, pp. 66–73. IEEE (2006)
16. Voelter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. In: 11th International Software Product Line Conference, SPLC 2007, pp. 233–242. IEEE (2007)
17. Voelter, M., Visser, E.: Product line engineering using domain-specific languages. In: 15th International Software Product Line Conference (SPLC), pp. 70–79. IEEE (2011)
18. Michavila, F., Martínez, J.M., Martín-González, M., García-Peñalvo, F.J., Cruz-Benito, J., Vázquez-Ingelmo, A.: Barómetro de empleabilidad y empleo universitarios. Edición Máster 2017. Observatorio de Empleabilidad y Empleo Universitarios, Madrid, España (2018)
19. Michavila, F., Martínez, J.M., Martín-González, M., García-Peñalvo, F.J., Cruz-Benito, J.: Barómetro de empleabilidad y empleo de los universitarios en España, 2015 (Primer informe de resultados). Observatorio de Empleabilidad y Empleo Universitarios, Madrid (2016)
20. Vázquez-Ingelmo, A., Cruz-Benito, J., García-Peñalvo, F.J., Martín-González, M.: Scaffolding the OEEU's data-driven ecosystem to analyze the employability of spanish graduates. In: García-Peñalvo, F.J. (ed.) Global Implications of Emerging Technology Trends, pp. 236–255. IGI Global, Hershey (2018)
21. García-Peñalvo, F.J., Cruz-Benito, J., Martín-González, M., Vázquez-Ingelmo, A., Sánchez-Prieto, J.C., Therón, R.: Proposing a machine learning approach to analyze and predict employment and its factors. Int. J. Interact. Multimedia Artif. Intell. **5**(2), 39–45 (2018)
22. Vázquez-Ingelmo, A., García-Peñalvo, F.J., Therón, R.: Generation of customized dashboards through software product line paradigms to analyse university employment and employability data. Learning Analytics Summer Institute Spain 2018 – LASI-SPAIN 2018, León, Spain (2018)
23. Cruz-Benito, J., Vázquez-Ingelmo, A., Sánchez-Prieto, J.C., Therón, R., García-Peñalvo, F. J., Martín-González, M.: Enabling adaptability in web forms based on user characteristics detection through A/B testing and machine learning. IEEE Access **6**, 2251–2265 (2018)
24. Vázquez-Ingelmo, A., García-Peñalvo, F.J., Therón, R.: Domain engineering for generating dashboards to analyze employment and employability in the academic context. In: 6th International Conference on Technological Ecosystems for Enhancing Multiculturality, Salamanca, Spain (2018)
25. Vázquez-Ingelmo, A., García-Peñalvo, F.J., Therón, R.: Application of domain engineering to generate customized information dashboards. In: International Conference on Learning and Collaboration Technologies, pp. 518–529. Springer, Switzerland (2018)
26. Elias, M., Bezerianos, A.: Exploration views: understanding dashboard creation and customization for visualization novices. In: IFIP Conference on Human-Computer Interaction, pp. 274–291. Springer, Heidelberg (2011)
27. Kakas, A.C.: A/B Testing (2017)
28. Moritz, D., Wang, C., Nelson, G.L., Lin, H., Smith, A.M., Howe, B., Heer, J.: Formalizing visualization design knowledge as constraints: actionable and extensible models in Draco. IEEE Trans. Visual. Comput. Graph. **25**, 438–448 (2019)