Uniquifying Architecture Visualization through Variable 3D Model Generation

Adrian Hoff
adho@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

Christoph Seidl
chse@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

Michele Lanza
michele.lanza@usi.ch
Software Institute @ USI
Lugano, Switzerland

ABSTRACT

Software visualization facilitates the interactive exploration of large-scale code bases, e.g., to rediscover the architecture of a legacy system. Visualizations of software structure suffer from repetitive patterns that complicate distinguishing different subsystems and recognizing previously visited parts of an architecture.

We leverage variability-modeling techniques to "uniquify" visualizations of subsystems via custom-tailored 3D models of recognizable landmarks: For each subsystem, we derive a descriptor and translate it to a (random but deterministic) configuration of a feature model of variable 3D geometry to support large numbers of different 3D models while capturing the design language of a particular type of landmark. We devised a hybrid variant derivation mechanism using a slots-and-hooks composition system for 3D geometry as well as adjusting visual characteristics, e.g., material. We demonstrate our method by creating various different trophies as landmarks for the visualization of a software system.

CCS CONCEPTS

• Software and its engineering → Software product lines; **• Human-centered computing → Visualization techniques**.

KEYWORDS

Variability Modeling, Software Visualization, 3D Model Generation

ACM Reference Format:
Adrian Hoff, Christoph Seidl, and Michele Lanza. 2023. Uniquifying Architecture Visualization through Variable 3D Model Generation. In *17th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2023), January 25–27, 2023, Odense, Denmark.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3571788.3571798

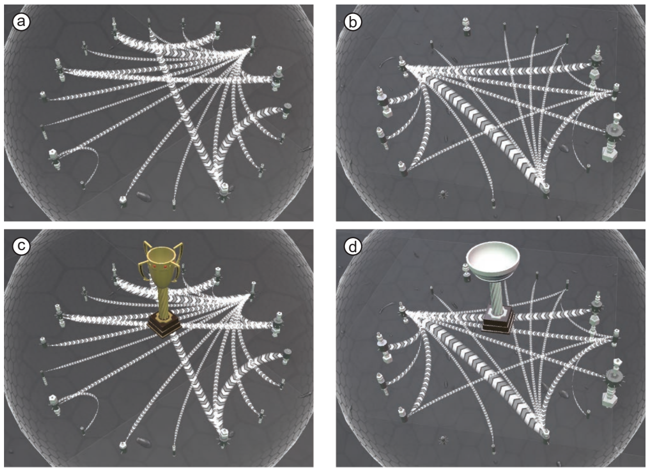

Figure 1: Visualization of two architectural elements suffering from hard-to-distinguish patterns (a, b) vs. the same architectural elements "uniquified" through custom-tailored landmarks (c, d).

1 INTRODUCTION

Software visualization represents the structure, behavior, or evolution of a software system in a visual format to foster comprehension [7]. When exploring large-scale code bases, e.g., to rediscover the design of a legacy system, visualization, esp. of structural system aspects, is essential for gaining an overview of a system's architecture [6, 11]. However, existing software visualization [1, 8, 10, 17, 28] is hampered by a pivotal problem: The structural visualizations of different parts of a software system are hard to differentiate due to repetitive and complex patterns in the visual representation that cannot easily be distinguished (see Figure 1).

The lengthy and iterative exploration of large-scale code bases is further complicated when comparing parts of a system or re-encountering an already visited part of a system: previously gained insights and mental models cannot be associated.

We propose a method for 3D software visualization to "uniquify" different parts of a software system by placing custom 3D models of recognizable landmarks along the structural representation of a particular part of a software system (e.g., packages, components, subsystems). We use trophies (as awarded to winners of sports competitions) as running example for one type of landmark (but one could also imagine statues, towers, bridges etc.).

To aid the structural representation of (potentially very many) parts of a system with custom 3D landmarks, we identified two challenges: First, a large number of sufficiently distinct 3D models is required to ensure that each different part of a system can be represented by a custom landmark. In addition, each type of landmark follows a certain design language [15, 22] that governs potential variations in its appearance, which has to be incorporated in the respective variations of the 3D models.

We leverage techniques from variability engineering to "uniquify" structural visualizations in 3D by generating distinct 3D models for a large number of landmarks following a common design language: We capture the design language of a particular landmark as configuration logic within a feature model where each feature is associated with partial 3D geometry or a visual characteristic of a 3D model, e.g., its material. We analyze the source code associated with the part of a system to be visualized and map its relevant characteristics to a configuration of the feature model. Finally, we create a custom-tailored 3D model representing one landmark via a hybrid variant derivation mechanism combining composition of partial 3D geometry with transformation of visual characteristics.

While this paper describes work in progress, we provide explanations and a prototype implementation[1] of all constituent concepts.

2 BACKGROUND

Our work marries concepts from software visualization with techniques from variability engineering.

2.1 Software Visualization

Software visualization presents a software system's structure, behavior, or evolution in a visual format (using 2D and 3D visual metaphors) to foster the comprehension of structural arrangements and relations [7, 20, 25, 26].

2D metaphors are mostly abstract graphs, trees, and diagrams [2, 12, 23]. 3D metaphors borrow analogies from the physical world to exploit viewers' familiarity with real-world constructs, e.g., cities, islands, or planets [1, 8, 10, 13, 17, 26, 28]. 3D software visualizations can be distinguished by the medium they employ, i.e., computer screens, augmented reality, or virtual reality [16, 18].

A fundamental purpose of software visualization is to provide an overview of a visualized subject system, e.g., to foster top-down exploration [4–6, 9–11, 14]. This includes the visualization of architecture-level constructs such as packages or subsystems. However, software systems may encompass architecture-level constructs with similar (but not identical) characteristics and size so that the generative processes for creating visual representations yield repetitive visual structures (see Figure 1). As a consequence, the exploration process is hampered by problems with orientation (inability to distinguish different architectural elements) and relating previously built memory models over the course of exploration (inability to recognize previously inspected parts of a system).

2.2 Variability Engineering

A *Software Product Line (SPL)* [21, 24] permits structured reuse within a highly-variable software family by exploiting commonalities and managing variabilities of closely related variations of software artifacts. Within an SPL, the *problem space* captures the configuration logic on conceptual level, whereas the *solution space* contains realization artifacts for all possible variants. Configuration logic is represented by a *variability model*, e.g., a *feature model*, which is a hierarchical tree of (de)selectable (optional/mandatory) features potentially structured into alternative groups permitting selection of at least one/exactly one feature (see Figure 3). *Cross-tree constraints* (commonly formulated in propositional logic) may further reduce the configuration space. A *configuration* is a valid selection of features obeying the configuration rules imposed by the feature model from the problem space. A *variant* is the realization of a configuration as realization artifacts from the solution space.

[1] https://gitlab.com/immersive-software-archaeology/variable-3d-landmarks

Variant derivation translates a configuration to a variant comprising the realization of one specific product. There are three principal kinds of variant derivation [24]: *Annotative* methods prune a representation comprising all possible variations (150% model) to only the realization artifacts required for a configuration (100% model). *Compositional* methods build a variant by combining constituent pieces associated with individual features or combinations thereof. *Transformational* methods modify characteristics of an existing variant to retrieve the desired variant. While realization artifacts of the solution space commonly consist of source code, other software artifacts can be made subject of variability within an SPL as well, e.g., in our case, 3D models.

3 VARIABLE 3D MODEL GENERATION

Our method utilizes concepts from variability modeling and 3D model generation to derive "uniquifying" 3D landmarks based on a configurable design language to be placed along otherwise similar visual structures in software architecture visualizations. A prime consideration for our method is to yield 3D landmarks that are distinct, yet following a well-defined design language. In turn, this design language needs to be expressive enough to provide distinct landmarks for all architecture-level constructs of a visualized software system. Figure 2 depicts a conceptual overview of our method.

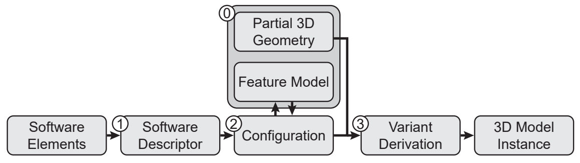

Figure 2: Overview of our method for generating "uniquifying" landmarks from a software descriptor.

We define a configurable design language for 3D landmarks via a feature model and partial 3D geometry ⓪. We automatically synthesize a software descriptor ① that we use to sample a valid configuration of the feature model ②, and forward it to a hybrid variant derivation mechanism ③ yielding a concrete 3D model for a landmark. In the following, we detail each step.

⓪ ***Visual Language Definition*** We perceive–and model–the design system [22, 27] associated with a design language in the sense of an SPL: While the individual visual characteristics of a 3D design language manifest in the implementation as 3D models, the (as of yet implicit) rules and variations permissible within a design language constitute a form of configuration logic. In consequence, the solution space (see Section 2.2) is comprised of individual parts of 3D geometry (partial 3D models). In addition, the problem space consists of a feature model that explicates design rules in the form of permissible configurations described as (de)selectable features. The left part of Figure 3 depicts these two artifacts using the example of configurable trophies. The feature model defines *which design elements* may be combined and the partial 3D geometry defines *how individual parts of a 3D model* may be combined.

To allow the combination of partial 3D geometry according to imposed rules, we have designed a *slots-and-hooks composition system* [3] for 3D models (see Figure 4). On a conceptual level,

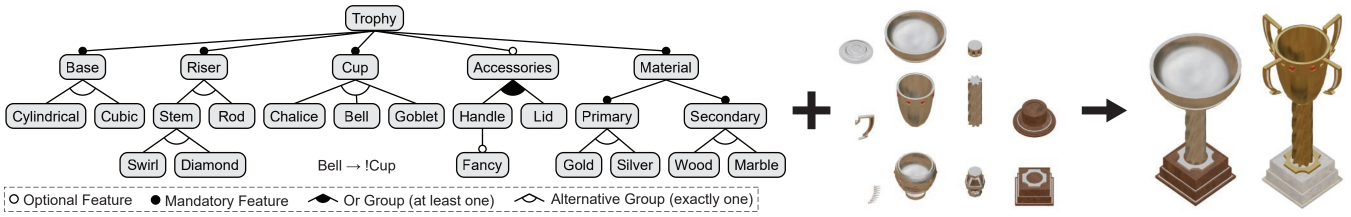

Figure 3: A feature model capturing the configuration rules of a design language (problem space) is aided with partial 3D geometry (solution space) to retrieve custom-tailored 3D models of landmarks via a hybrid variant derivation procedure composing 3D geometry and transforming visual characteristics, e.g., material.

a *slot* serves as an extension point for 3D geometry that defines where and with which rotation/scale an element may be attached. Likewise, a *hook* (defined in another part of 3D geometry) declares the principle option for serving as an extension to a compatible slot. Whether a slot of part of 3D geometry is actually bound (at all) and, if so, with the compatible hook of which specific other part of 3D geometry is determined via a configuration of the feature model during variant derivation (see below). On a practical level, we implement the slots-and-hooks composition system for 3D models by using invisible elements of 3D geometry as markers to define position, rotation, and scale for both slots and hooks. We established a naming convention that identifies these markers as either slot or hook and determine compatibility via matching names.

① ***Software Descriptor Synthesis*** The landmarks we generate have to be created deterministically from (the implementation of) an architectural element and, at the same time, must be stable over miniscule changes to the implementation. To achieve these goals and steer the subsequent creation of a configuration (see below), we borrow the concept of a *descriptor* from computer vision: A visual descriptor is an abstract summary of visual characteristics in an image [29] (e.g., edges) that can be *sensitive* to certain characteristics (e.g., color changes) and *robust* against others (e.g., rotation).

We adapt the concept of a visual descriptor to design a *software descriptor* for architectural elements: The software descriptor should be robust against minor changes (e.g., creating a new attribute or method within a class), yet sensitive toward major modifications (e.g., removing entire classes). In addition, in our use case, the software descriptor steers the sampling of a configuration for a 3D model and the respective landmarks should be stable for each architectural element. Hence, creation of the software descriptor must be deterministic, i.e., the same input of software elements must always result in the same descriptor.

While determining the full scope of an expressive software descriptor is part of our ongoing work, we illustrate the principle use of a software descriptor: For each architectural element, we calculate a software descriptor that is sensitive to the number of contained elements (e.g., classes, structs, etc.) as well as their respective qualified names. We devise a canonical form of the descriptor by combining the number of elements with a hash of all fully qualified (i.e., unambiguously identifiable) names in alphabetical order. As an example, consider a Java package `com.application` consisting of 3 classes `Model`, `View`, and `Controller`. Our illustrative software descriptor for this package starts with a 3 (number of contained elements) followed by a hash of their concatenated fully qualified names resulting in "32117573461" as the software descriptor.

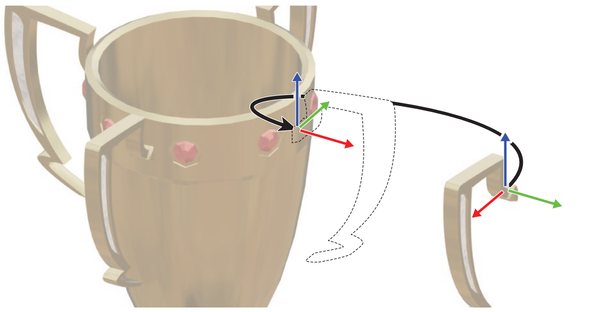

Figure 4: Example of the compositional aspect of our variant derivation. A slots-and-hooks composition mechanism combines partial geometry based on a naming convention of the respective parts.

Through this procedure, we capture relevant high-level aspects of software constructs contained within an architectural element while abstracting from negligible details within a deterministic and compact representation.

② ***Steered Random Configuration Sampling*** We derive a valid configuration from the visual design language based on configuration sampling guided by our software descriptor. Similar to the synthesis of software descriptors, the sampling of configurations must be deterministic, i.e., the same descriptor should always result in the same configuration. At the same time, different software descriptors should result in different configurations. However, which specific configuration is determined for a particular software descriptor is of no concern (as each represents a viable landmark). Hence, we employ random configuration sampling but steer the selection process by using our software descriptor as seed for the random generator to ensure deterministic results. In our implementation[1], we use the random sampling provided by FeatureIDE[2].

③ ***Hybrid Variant Derivation*** To realize a configuration in the form of a variant comprised of a custom-tailored 3D model, we have devised a hybrid variant derivation mechanism that includes aspects of compositional and transformational methods.

The compositional part of variant derivation employs the slots-and-hooks mechanism recursively by stepping through relevant partial 3D geometry and composing parts according to matches in

[2]https://featureide.github.io/

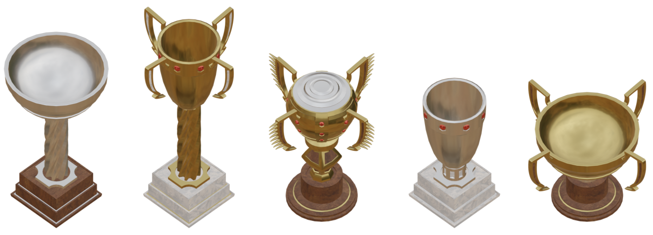
Figure 5: Example of multiple visually distinct trophies created through variant derivation for our running example.

the contained slots and hooks. For instance, for each of the four handles of the trophy variant depicted in Figure 4, the hook (and, therefore, the associated 3D geometry) is positioned, rotated, and scaled according to the respective slot defined in this specific kind of cup. The order for composing slots and hooks may be arbitrary as each will result in an identical 3D model.

The transformational part of variant derivation mutates the appearance of 3D geometry by revisiting all (previously) partial 3D geometry and exchanging materials. For that purpose, the variant derivation mechanism uses a selection of pre-configured materials by matching their names according to a feature naming convention. In our running example, we use two different types of materials that are (primarily) visible with metallic parts of a trophy (riser, cup, handles, lid) or with the base of the trophy, where each type of material has two possible variations (see Figure 1).

The result of variant derivation is a concrete 3D landmark model that developers of 3D software architecture visualizations can utilize to "uniquify" architecture-level constructs similar to the examples depicted in Figure 5.

4 OUTSTANDING CHALLENGES

While we have devised a prototype implementation to demonstrate our concepts, there are outstanding challenges, which we list here to foster academic discourse and invite potential collaborators.

Unique Configuration for each Descriptor Even though our mapping of a software descriptor to a configuration of the feature model is deterministic, there may be cases when two distinct descriptors are mapped to the same configuration and, thus, yield the same 3D model. In part, this is (inadvertently) by design as the number of configurations in the problem space may be lower than the potential number of different architecture descriptors each representing a distinct individual architectural element. To address this challenge, we foresee two promising directions: On conceptual level, we strive to a-priory assess whether the configuration space is sufficiently large to accommodate all possible architectural elements and, if not, to guide expansion of the configuration space (e.g., in the example shown in Figure 3, that adding a new type of lid would add 144 configurations). On practical level, we will explore configuration options via attributes and with continuous values (e.g., for model scale or custom colors) to further expand the configuration space with little burden of creating new elements for 3D models.

Continuous Variation Software visualization commonly incorporates relations between architectural elements in the spatial arrangement of the visualization [26], e.g., hierarchies of nested components or the degree of coupling between different subsystems. In consequence, architectural elements that are "more strongly" related to each other are commonly collocated in the visualization. Currently, our method creates different configurations and, thus, visually distinct 3D models for each architectural element. However, we see great potential in exploiting the relation of architectural elements by having landmarks of closely related architectural elements share certain visual characteristics (e.g., an area where all landmarks have a gold material). For this purpose, we will extend the software descriptor to incorporate hierarchy and relation of elements but will also research how to selectively vary a configuration to achieve continuous variation for the generated 3D models.

Robustness toward Evolution While our tentative implementation of an architecture descriptor is robust against changes in implementation details (e.g., methods and attributes), it is sensitive to changes in the set of classes contained in an architectural element and names of the contained classes. However, over the course of software evolution, even these characteristics may change. Currently, this would yield a different configuration and, thus, an entirely different 3D model for a landmark so that the relation to a previously established mental model will be harmed. To cope with evolutionary changes, we strive to either make the software descriptor robust toward certain architectural changes, so that associated landmarks remain unchanged, or determine a method that allows creation of largely similar configurations to that created landmarks contain only minuscule visual differences to the previous edition.

Landmarks with Semantics In our prototype implementation, we sample 3D models based on a random configuration of a feature model using a software descriptor. While this allows us to deterministically generate a large variety of 3D models that follow a configurable design language, the resulting 3D model variants do not convey characteristics of represented software elements such as their size (e.g., by scaling landmarks), quality (e.g., by including visual effects such as cracks, cf. [19]), or complexity (e.g., by selecting certain features that indicate complexity such as the complex handle in our running example).

5 CONCLUSION & FUTURE WORK

In this paper, we exploited variability engineering techniques to devise a method for generating custom-tailored 3D models of landmarks to “uniquify” otherwise visually similar structures in software architecture visualizations. Our method allows to encode the design language of a family of 3D models within a feature model while also permitting to leverage 3D modeling software to prepare visually appealing (partial) 3D models.

The outstanding challenges of our work are rooted within two fundamental areas: descriptor robustness and variant similarity. *Descriptor robustness:* Our joint goals of providing unique 3D models for landmarks “uniquifying” conceptually different architectural elements and ensuring a visual similarity between closely related architectural elements are, at times, diametrically opposed. Accommodating both goals requires determining additional characteristics to incorporate in the descriptor as well as finding a tradeoff between characteristics the descriptor should be sensitive to/robust against. *Variant similarity:* While there are methods and measures for determining the similarity of different configurations (i.e., selections from the feature model), our goal of achieving visual similarity/continuity (for various use cases) requires the ability to prognose similarity of multiple variants (e.g., visual similarity of 3D models), ideally, without having to rely on analyzing resulting products. Our future work is aimed at researching solutions for descriptor robustness and variant similarity to further improve the benefits of our method for demarcating/recognizing architectural elements in the visualization of even larger and evolving software systems.

ACKNOWLEDGMENTS

This work is supported by the DFF (Independent Research Fund Denmark) within the project ISA – Immersive Software Archaeology (0136-00070B).

REFERENCES

[1] Sazzadul Alam, Sandro Boccuzzo, Richard Wettel, Philippe Dugerdil, Harald Gall, and Michele Lanza. 2009. EvoSpaces-Multi-dimensional navigation spaces for software evolution. In *Human Machine Interaction*. Springer, 167–192.

[2] Mohammad Alnabhan, Awni Hammouri, Mustafa Hammad, Mohammad Atoum, and Omamah Al-Thnebat. 2018. 2D visualization for object-oriented software systems. In *2018 International Conference on Intelligent Systems and Computer Vision (ISCV)*. IEEE, 1–6.

[3] Uwe Aßmann. 2003. *Invasive Software Composition*. Springer.

[4] Sheelagh Carpendale and Yaser Ghanam. 2008. *A survey paper on software architecture visualization*. Technical Report. University of Calgary.

[5] Pierre Caserta and Olivier Zendra. 2010. Visualization of the static aspects of software: A survey. *IEEE transactions on visualization and computer graphics* 17, 7 (2010), 913–933.

[6] Noptanit Chotisarn, Leonel Merino, Xu Zheng, Supaporn Lonapalawong, Tianye Zhang, Mingliang Xu, and Wei Chen. 2020. A systematic literature review of modern software visualization. *Journal of Visualization* 23, 4 (2020), 539–558.

[7] Stephan Diehl. 2007. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media.

[8] Elke Franziska Heidmann, Lynn von Kurnatowski, Annika Meinecke, and Andreas Schreiber. 2020. Visualization of Evolution of Component-Based Software Architectures in Virtual Reality. In *2020 Working Conference on Software Visualization (VISSOFT)*. IEEE, 12–21.

[9] Adrian Hoff, Lea Gerling, and Christoph Seidl. 2022. Utilizing Software Architecture Recovery to Explore Large-Scale Software Systems in Virtual Reality. In *2021 Working Conference on Software Visualization (VISSOFT)*. IEEE.

[10] Adrian Hoff, Michael Nieke, and Christoph Seidl. 2021. Towards immersive software archaeology: regaining legacy systems' design knowledge via interactive exploration in virtual reality. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1455–1458.

[11] Taimur Khan, Henning Barthel, Achim Ebert, and Peter Liggesmeyer. 2012. Visualization and evolution of software architectures. In *Visualization of Large and Unstructured Data Sets: Applications in Geospatial Planning, Modeling and Engineering-Proceedings of IRTG 1131 Workshop 2011*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[12] Mircea Lungu, Michele Lanza, and Oscar Nierstrasz. 2014. Evolutionary and collaborative software architecture recovery with Softwarenaut. *Science of Computer Programming* 79 (2014), 204–223.

[13] Andrian Marcus, Louis Feng, and Jonathan I Maletic. 2003. 3D representations for software visualization. In *Proceedings of the 2003 ACM symposium on Software visualization*. 27–ff.

[14] Anna-Liisa Mattila, Petri Ihantola, Terhi Kilamo, Antti Luoto, Mikko Nurminen, and Heli Väätäjä. 2016. Software visualization today: Systematic literature review. In *Proceedings of the 20th International Academic Mindtrek Conference*. 262–271.

[15] Tim McCreight. 2006. *Design Language, Interpretive Edition*. Brynmorgen Press.

[16] Leonel Merino, Johannes Fuchs, Michael Blumenschein, Craig Anslow, Mohammad Ghafari, Oscar Nierstrasz, Michael Behrisch, and Daniel A Keim. 2017. On the Impact of the Medium in the Effectiveness of 3D Software Visualizations. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 11–21.

[17] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. 2017. CityVR: Gameful software visualization. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 633–637.

[18] David Moreno-Lumbreras, Roberto Minelli, Andrea Villaverde, Jesus M Gonzalez-Barahona, and Michele Lanza. 2023. CodeCity: A comparison of on-screen and virtual reality. *Information and Software Technology* 153 (2023), 107064.

[19] Johann Mortara, Philippe Collet, and Anne-Marie Pinna-Dery. 2022. Customizable visualization of quality metrics for object-oriented variability implementations. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume A*. 43–54.

[20] Richard Müller and Dirk Zeckzer. 2015. Past, present, and future of 3D software Visualization-A systematic literature Analysis. In *International Conference on Information Visualization Theory and Applications*, Vol. 2. SCITEPRESS, 63–74.

[21] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer Berlin/Heidelberg.

[22] Richard Poulin. 2018. *The Language of Graphic Design Revised and Updated: An Illustrated Handbook for Understanding Fundamental Design Principles*. Rockport Publishers Inc.

[23] Blaine A Price, Ian S Small, and Ronald M Baecker. 1992. A taxonomy of software visualization. In *Proc. 25th Hawaii Int. Conf. System Sciences*.

[24] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 477–495.

[25] M-AD Storey, F David Fracchia, and Hausi A Müller. 1999. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software* 44, 3 (1999), 171–185.

[26] Alfredo R Teyseyre and Marcelo R Campo. 2008. An overview of 3D software visualization. *IEEE transactions on visualization and computer graphics* 15, 1 (2008), 87–105.

[27] Sarrah Vesselov and Taurie Davis. 2019. *Building Design Systems: Unify User Experiences through a Shared Design Language*. Apress.

[28] Richard Wettel and Michele Lanza. 2008. Codecity: 3d visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*. 921–922.

[29] Jianxin Wu and Jim M Rehg. 2010. Centrist: A visual descriptor for scene categorization. *IEEE transactions on pattern analysis and machine intelligence* 33, 8 (2010), 1489–1501.