A Catalog of Unintended Software Dependencies in Multi-Lingual Systems at ASML

Tom Groot
Eindhoven University of Technology & ASML
The Netherlands
t.groot@tue.nl

Lina Ochoa Venegas
Eindhoven University of Technology
The Netherlands
l.m.ochoa.venegas@tue.nl

Bogdan Lazăr
ASML
The Netherlands
bogdan.lazar@asml.com

Jacob Krüger
Eindhoven University of Technology
The Netherlands
j.kruger@tue.nl

ABSTRACT

Multi-lingual software systems build on interconnected components that are implemented in different programming languages. The multi-lingual nature of such systems causes additional complexity, for instance, when developers aim to identify what components of a system use the same data. Organizations and developers typically aim to adhere to a specified system architecture to avoid certain dependencies between multi-lingual components. However, such dependencies may still be introduced and only resolved later on. Thus, we refer to them as *unintended* dependencies: dependencies that may exist, but are not wanted by the developers or organization. There has been little research on multi-lingual systems so far, and dependencies within such systems have not been studied explicitly. With this paper, we tackle this issue by contributing a catalog of unintended software dependencies in multi-lingual systems. We elicited it by interviewing 17 practitioners at ASML. We report eight types of unintended dependencies, their causes, the resulting problems, and how they can be resolved. Further, we connect our findings to research on software smells and dependencies in monolingual systems. Our contributions serve as recommendations for practitioners on how to deal with unintended dependencies, as supportive evidence for existing research, and as basis for new techniques for managing dependencies in (multi-lingual) systems.

CCS CONCEPTS

• **Software and its engineering → Layered systems**; **Software design engineering**; **Maintaining software**.

KEYWORDS

Dependencies, Software Architecture, Multi-Lingual Systems, Software Quality, Software Maintenance

ACM Reference Format:
Tom Groot, Lina Ochoa Venegas, Bogdan Lazăr, and Jacob Krüger. 2024. A Catalog of Unintended Software Dependencies in Multi-Lingual Systems at ASML. In *46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3639477.3639725

ICSE-SEIP '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0501-4/24/04.
https://doi.org/10.1145/3639477.3639725

1 INTRODUCTION

In practice, more and more software systems involve components implemented via different programming languages and their respective technologies, which we refer to as *multi-lingual* (a.k.a., multi-language or cross-language) systems. Consequently, practitioners can benefit from the pros of combining different languages, such as reducing costs through reusing libraries implemented in a different language, optimizing performance by using specific languages, or integrating legacy code that has not been re-engineered, yet [3]. To manage the complexity of multi-lingual systems, developers divide them into *components* (or modules, subsystems, libraries, services, etc.) that are ideally easier to maintain and adapt to new requirements compared to monoliths [1]. In turn, to leverage the functionalities offered by different components, dependencies among these components must be established. Since components written in different programming languages cannot simply access each others' code elements like variables or functions (we refer to *symbols*), even more complexity is added by a separate interface that exposes elements within such a dependency.

Unfortunately, in practice, dependencies between components may not adhere to rules defined by the organization or developer community that implements the multi-lingual system. That is, such a system may involve *unintended dependencies* that reduce its modularity and are typically unexpected by developers. Conversely, this may complicate program comprehension and cause misbehavior—threatening the benefits of using components implemented in different languages [1, 9, 28, 31, 44, 45].

Identifying, managing, and improving dependencies in monolingual systems has been researched extensively in the context of software architecture, architecture smells, and package managers [8, 18, 24, 34]. However, adapting such techniques for multi-lingual systems is challenging, for instance, because the different coding styles make it more challenging to identify whether the same data is used by multiple components [21, 35]. To the best of our knowledge, there is no mature tooling to solve the problem of identifying and resolving unintended dependencies in multi-lingual systems. In fact, research on the architecture and design of multi-lingual systems has only recently gained more traction [2–4, 33]. Still, the existing studies (cf. Section 3) have not investigated the notion of unintended dependencies between the different components in a multi-lingual system. Real-world insights into unintended dependencies based on practitioners' experiences are a valuable contribution to provide a foundation for future research and help other developers.

In this paper, we contribute such insights by reporting the results of interviews with 17 practitioners working on large multi-lingual software systems at ASML. During these interviews, we asked

the practitioners about their experiences with dependencies in the multi-lingual systems they worked on, and which of these dependencies they consider unintended for what reasons. We transcribed the interviews and conducted a thematic analysis [7] to construct a catalog of eight unintended dependencies, including their causes, consequent problems, and resolution strategies. More detailed, we contribute the following within this paper:

- We define a catalog of eight unintended dependencies between components in multi-lingual systems that the ASML developers have experienced.
- We describe the causes, consequences, and resolution strategies associated to these unintended dependencies based on our interviewees' expertise.
- We discuss the implications of our catalog for research and practice, focusing on the commonalities and differences of mono-lingual and multi-lingual systems.

We argue that these contributions are helpful to shed light into the specifics of multi-lingual systems and unintended dependencies. Our insights can help practitioners manage their systems and guide researchers in designing novel techniques for identifying, managing, and resolving dependencies in multi-lingual systems.

2 THE ASML CASE

ASML Holding B.V. is a Dutch multinational company with more than 40,000 employees worldwide. The company holds more than 16,000 patents, generated a revenue of more than 21 € billion in 2022, and is considered the most highly valued tech company in Europe.[1] As of 2022, the company is the largest supplier of lithography machines needed for producing computer chips worldwide. It is also the only supplier of extreme ultraviolet lithography photolithography machines needed for producing the most advanced computer chips.

Software is a key part in ASML's business, since software monitors and manages the different components within chip-production machines. To save development time and improve customer satisfaction, meeting non-functional software requirements, such as performance, scalability, reliability, and maintainability, is important for ASML. However, with the rising complexity introduced by a growing number of components written in different programming languages (i.e., resulting in a multi-lingual system), meeting such requirements becomes progressively challenging. In particular, the introduction of unintended dependencies between components can hinder software quality and maintainability of the whole multi-lingual system. So, ASML requires an understanding of what its unintended multi-lingual software dependencies are, as well as how to detect and resolve these dependencies.

ASML's systems build on a layered architecture that includes, but is not limited to, a presentation and an application layer. A system is further decomposed into subsystems that integrate different components, with each subsystem being the responsibility of a team. The layering and decomposition are orthogonal structures, meaning that a component of one subsystem can exist in the same layer of another subsystem. Additionally, the components are written in different programming languages, which can be either Python, C, C++, Java, Matlab, or Julia. A typical system at ASML involves millions

[1] https://www.bbc.com/news/business-64514573

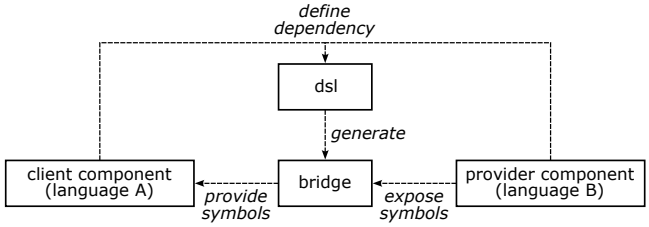


Figure 1: Concept of ASML's DSL generating a bridge between two components that are implemented in different programming languages.

of Lines of Code (LOCs), and each subsystem comprises around hundreds of thousands of LOCs. For instance, one concrete subsystem of ASML's system has 22 components that involve 867 C++ (234,814 LOCs), 337 C (119,426 LOCs), and 46 Python (11,189 LOCs) files. Thus, most individual subsystems at ASML already represent quite large multi-lingual software.

Within ASML's multi-lingual systems, dependencies among components are defined at two places. First, dependencies are defined in a build scope definition file, which is used during the build process to identify all dependencies of the selected components and integrate those into the system. Second, the dependencies are defined via a Domain-Specific Language (DSL) tailored for this purpose. This DSL allows to specify an interface that describes what code symbols (e.g., variables, functions, fields) are used by different components—thereby establishing the actual dependency. Afterwards, the DSL is used to automatically generate a "bridge" that exposes the functionality of a provider component written in a specific programming language to a client component written in another language. We sketch a conceptional overview of this process in Figure 1. Conceptually, we could say that the DSL makes a dependency work functionally by creating the bridge, whereas all dependencies are defined in the build scope definition file to ensure the right components are selected. Using this design, ASML can benefit from using different programming languages, allowing its developers, for instance, to build on Python for data-science focused components and C or C++ for performance-critical components. Regarding the depicted ASML subsystem, the DSL code involves 470 additional files that include 41,787 LOCs.

To exemplify how the DSL and bridge work, let us assume a client component written in Python and a provider component written in C. Within the provider component, a developer may have implemented a function `f` that they want to use in the client component. For this purpose, they specify the input parameters and return type of `f` as well as the programming languages of the provider component (C) as an interface in the DSL. Now, the DSL generates a new C file as the actual implementation of the interface that allows to access the relevant symbols (i.e., `f`). The developer defines the target languages (i.e., Python) for the bridge in the makefile of the DSL. When running the build process, the DSL now generates the actual bridge by creating a Python file that involves a method with the same name as `f`, and which can be called by the client component to access the C code in the bridge that provides `f`. To ensure the mapping across the different programming languages,

the generated files are named identically according to the scheme `providerComponent_interface_function`.

3 RELATED WORK AND RESEARCH GOAL

Next, we summarize the related work on software dependencies, based on which we motivate the novel contributions of our work.

Anti-Patterns and Smells in Multi-Lingual Systems. Neitsch et al. [33] have analyzed the build systems of five multi-lingual open-source systems, identifying several patterns and anti-patterns. For instance, the anti-pattern "Filename Collision" can cause runtime errors because file systems may use case-sensitive or case-insensitive file names. Similarly, Abidi et al. [3] elicited six anti-patterns of multi-lingual systems from papers and gray literature (e.g., developers' blogs). The anti-pattern related most closely to our research goal is "Excessive Inter-Language Communication," which refers to (unintended) situations in which components in different languages are excessively calling each other. Often, such anti-patterns or otherwise "bad" pieces of software are also described using the metaphors of code [27], design [5], or architecture [32] smells [39]. Particularly architecture smells focus on dependencies in software systems that are perceived as problematic, and thus are often unintended. Architecture smells have been studied extensively in the past, but primarily in the context of a single programming language for which smells like "Cyclic Dependencies" have been defined [6, 18, 38]. For multi-lingual systems, Abidi et al. [2, 4] have further identified 12 code and 15 design smells. However, we are not aware of papers focusing on architecture smells and dependencies in multi-lingual systems. In summary, while there have been many studies on design and dependency issues in software, multi-lingual systems only recently gained attention and an in-depth understanding of dependency issues in such systems is missing. Also, the existing works in this area focus on open-source systems and researchers' own analyses of these, whereas we report on interviews with practitioners working on large-scale industrial systems.

Program Analysis for Multi-Lingual Systems. Recently, several researchers have proposed techniques that adapt static and dynamic program analyses for multi-lingual systems, but integrating multiple languages in such techniques is challenging [21, 35]. For instance, different coding styles, varying naming conventions, and ways to identify what symbols are used in the individual components pose challenges. Most prominently, Pereira dos Reis et al. [35] found that 83.1 % of the techniques for code-smell detection cover only one programming language. A technique for inspecting multi-lingual systems has been developed by GitHub, using Abstract-Syntax Trees (ASTs) to match the symbols of different programming languages [11]. However, the tool does not support the detection of smells or unintended dependencies. Again, there has been more extensive research for systems implemented in a single language. For instance, Chuang et al. [10] have introduced OPAL, a tool for inspecting and removing (unintended) dependencies in Java programs. Other researchers have proposed techniques for identifying, visualizing or refactoring dependencies or smells within a software system [15, 16, 19, 37, 46]. However, we are not aware of such research focusing on dependencies in multi-lingual systems or being conducted in real-world industry settings.

Research Goal. ASML has defined a layered architecture and design rules regarding the dependencies between components to ensure its systems' quality. For instance, the components of a layer A may be allowed to call components of another layer B, but not vice versa. So, any call from a component in layer B to a component from layer A would be considered an unintended dependency, since it would not break the system but does not match ASML's design rules. When inspecting the related work, we neither identified an existing technique that supports detecting and analyzing dependencies in ASML's multi-lingual systems; nor a study that provides an understanding of (unintended) dependencies or architecture smells relevant to multi-lingual systems. Consequently, we decided to study unintended dependencies at ASML to provide a catalog and empirical data on such dependencies. To guide our study, we defined three research questions (RQs):

RQ$_1$ *What are types of unintended dependencies for the multi-lingual systems of ASML?*
So far, unintended dependencies have primarily been addressed as architecture smells in mono-lingual systems. While we assumed and found that the unintended dependencies in multi-lingual systems are highly similar to smells or design rules for mono-lingual systems, corresponding evidence and details on the specifics of multi-lingual systems are missing. We tackle this gap by eliciting a catalog of eight unintended dependencies from interviews with practitioners at ASML.

RQ$_2$ *What are causes and consequences of the unintended dependencies in multi-lingual systems?*
We further investigated why dependencies may still be introduced despite being unintended. Moreover, we elicited whether there are specific challenges or consequences for which a particular type of unintended dependency should be avoided. Thereby, we provide a better understanding on the problems attached to each unintended dependency.

RQ$_3$ *How are the unintended dependencies in multi-lingual systems resolved by developers?*
The overarching idea with which we started was to develop techniques for identifying and removing unintended dependencies to improve the quality of a software system. As a step towards this direction, we further elicited how the interviewed practitioners resolve unintended dependencies. So, we provide advice on how to manage and deal with unintended dependencies in real-world multi-lingual systems.

By answering these research questions, we contribute insights into developers' perceptions about unintended dependencies in multi-lingual systems. Our insights provide supportive evidence for existing research on software quality, extend it with new insights, and serve as a starting point to develop new tools.

4 STUDY DESIGN

To answer our research questions, we conducted an exploratory field study using semi-structured interviews. This research strategy allows us to generate knowledge from a realistic context, at the cost of limited precision and generalizability [41]. Consequently, our results are a stepping stone towards creating a body of knowledge on unintended dependencies in multi-lingual systems. We conducted our interviews within ASML following the guidelines by

Strandberg [42]. The interviews have been approved by ASML and adhere to the research integrity process at Eindhoven University of Technology (TU/e). In particular, we submitted our interview design and an ethical review form for evaluation, both of which were approved by the university's Ethical Review Board. Next, we describe the design of the interviews, how we analyzed the data, how we recruited participants, and the participants' demographics.

4.1 Interview Design

We designed and conducted our interviews during a collaboration between ASML and TU/e in which we analyzed dependencies in ASML's multi-lingual systems. During this collaboration, we learned that particularly unintended dependencies are of interest to ASML to ensure its systems' quality. To obtain a better understanding of such dependencies, we defined the described research goal and research questions, which guided the design of our semi-structured interviews. During the actual design, we employed an incremental and iterative process: In the beginning, the first author of this paper proposed a number of questions relevant for understanding unintended dependencies. Then, we collaboratively refined and extended these questions among all authors, reflecting on our experiences and following the guidelines by Strandberg [42] to ensure a useful design. For external validation, we conducted two pilot interviews with volunteers at ASML, based on which we adjusted the questions and conduct of the interviews. Finally, a data steward and the Ethics Review Board at TU/e reviewed our interview design regarding, for instance, the collection of personal data, the storing of data, and the consent form.

In Table 1, we provide an overview of the questions we asked during our interviews and their mapping to our research questions. Note that we updated the wording of two questions (Q_4, Q_{10}) to make these clearer for this paper. Specifically, we added the texts in square brackets, which were previously an "it" (Q_4) or missing (Q_{10}) and were both not needed within the overall structure of our interviews because the questions referred to concrete examples previously provided by our interviewees. First, we aimed to capture the concept of unintended dependencies ($Q_{1–3}$) from the perspectives of our interviewees and ASML to tackle **RQ_1**. Building on these definitions and concrete examples the interviewees identified, we aimed to elaborate on the causes and consequences ($Q_{4–7}$) of the unintended dependencies to answer **RQ_2**. Lastly, we investigated how these dependencies could be resolved ($Q_{6–10}$) to address **RQ_3**. Since these questions were only guiding our semi-structured interviews, we sometimes deviated into more details, depending on the experiences of our interviewees.

Regarding the consent form, we provided general information about the purpose of our study, its objectives, its methodology, the risks involved, as well as that only anonymized and aggregated results will be shared with ASML or in a publication. To guarantee the integrity of our participants, we stipulated a discussion on consent and withdrawal based on the consent form prior to the conduct of each interview. We conducted the interviews online using Microsoft Teams. This allowed the interviewees to be within a chosen comfortable setup that also ensures anonymity. Moreover, the online setup allowed us to let Microsoft Teams automatically create transcripts without the need for recording or manually transcribing the interviews. To ensure the transcripts' quality, the first author as the interviewer took notes during the interviews and manually corrected the transcripts directly after each meeting (e.g., fixing rare words that were wrongly transcribed). We stored the transcripts only in one personal SURFDrive,[2] a secure Dutch cloud storage for education and research.

4.2 Data Analysis

To analyze the interview transcripts, we performed a thematic analysis as "a method for identifying, analyzing[,] and reporting patterns (themes) within data" following the guidelines by Braun and Clarke [7]. The customary first step in this analysis is to use coding techniques to associate sections of the transcripts with codes that are then used to identify themes. For this purpose, the first author employed open coding (also called inductive coding) using ATLAS.ti[3] to manually extract codes from the transcripts [12]. During open coding, codes are elicited from the study sources themselves without defining them from preexisting knowledge or a theory in advance. We chose this method due to the lack of related work and the exploratory nature of our study. In more detail, we combined two types of coding subcategories: in-vivo and descriptive coding. With in-vivo coding, we extracted labels from the transcribed words of the interviewees [30], while we used descriptive coding to summarize the contents of text fragments into labels [17]. The remaining authors checked the assignments of codes to quotes and resolved any varying opinions through collaborative discussions. After coding all transcripts, we segregated the codes into groups based on recurring patterns and themes following a card-sorting methodology [47]. We exemplify non-confidential codes and themes in our codebook within our discussion (cf. Section 6).

4.3 Recruitment

To conduct our interviews, we aimed to recruit domain experts working in software development at ASML. For this purpose, the first author started to invite ASML employees who were proposed by the third author or who were identified as experts by the first author during his time at ASML. After this convenience sampling, we aimed to mitigate biases and increase diversity by inviting employees from different teams at ASML based on snowball sampling (i.e., referrals of our interviewees) [43]. We invited participants via an internal messaging app of ASML, also stating that the participation would be confidential and voluntary (i.e., not participating or dropping out was possible without any consequences). In the end, we stopped conducting more interviews when we reached saturation across multiple interviews, meaning that the newly collected data did not reveal new insights into unintended dependencies [22].

4.4 Participants' Demographics

Overall, we invited 39 ASML employees from six different software-development teams across four departments. Of these, 17 agreed to and participated in an interview (43.6 % participation rate). Our participants represent four different roles at ASML, namely

[2] https://www.surf.nl
[3] https://atlasti.com

Table 1: Overview of our interview questions.

ID	interview question	RQ
Q_1	What do you consider an unintended dependency at the symbol level, between components?	RQ_1
Q_2	Does ASML have a standard definition of what an unintended dependency at the symbol level, between components is? If so, which one?	RQ_1
Q_3	Have you encountered unintended dependencies within ASML? If so, can you explain or possibly draw the situation?	RQ_1
Q_4	Does [an unintended dependency] still exist in projects you have worked on? If so, why does it still exist?	RQ_2
Q_5	Are there benefits to resolving this type of unintended dependencies between components? If so, which ones?	RQ_2
Q_6	What do you think are the challenges regarding detecting unintended dependencies at the symbol level, between components?	RQ_2 & RQ_3
Q_7	Are there challenges related to resolving this type of unintended dependencies between components? If so, which ones?	RQ_2 & RQ_3
Q_8	How do you detect unintended dependencies at the symbol level, between components?	RQ_3
Q_9	Are there guidelines within ASML to remove unintended dependencies at the symbol level, between components?	RQ_3
Q_{10}	Do you know a possible solution to the situation [of a specific unintended dependency]? If not, how was the situation removed?	RQ_3

software engineers (11) who focus on implementing ASML's systems, and thus its dependencies;
software architects (4) who are responsible for designing the overall structure of the systems, and thus define its dependencies;
test architects (2) who ensure a system's quality and reliability by designing tests, which includes testing dependencies; and
Scrum master (1) who is in charge of coordinating a whole development team.

Note that one participant is both a software engineer and a test architect, which is why the total number of roles is 18. Four of our participants had 0–5, seven 5–10, five 10–15, and two more than 15 years of experience in software engineering. Since our interviewees span various teams (and consequently subsystems), departments, as well as roles and have varying extents of experience, we argue that our results provide representative insights into unintended dependencies in multi-lingual systems at ASML.

5 UNINTENDED DEPENDENCIES CATALOG

In this section, we present the results of our study. In particular, we provide a comprehensive catalog of the eight unintended dependencies in multi-lingual systems we synthesized from the interviews. For each of these unintended dependencies, we provide a short definition (RQ_1), summarize its causes and consequences (RQ_2), and describe resolution strategies used by our interviewees (RQ_3). The number after each dependency name indicates the number of transcripts from which we extracted information about this dependency. We order the catalog by the number of occurrences as an indication of a dependency's frequency and alphabetical afterwards.

5.1 Upwards Dependency (9)

Within a layered architecture, components are typically located in a specific layer to improve the separation of concerns, with each layer representing a certain role and responsibility in the system. Typically, requests in the system are expected to travel from higher layers to lower layers, and data should move the other way around from lower-layer components to higher-layer components [36]. In fact, lower-layer components should not be concerned with the processed data of components on a higher layer. Nine interviewees described upwards dependencies as unintended, since they reverse the flow of requests and potentially data. We display a conceptual example for such a dependency in Figure 2a, in which the component "hardware driver" acts as client by requesting symbols from the component "controller," which is on a higher layer. Please note that we merge this type of dependency from two kinds of descriptions by our interviewees: Most described this case from the perspective of the functionality ("upwards-functionality dependency"), whereas others considered the perspective of data flow ("downwards data flow dependency"). We merge these two cases, because they represent the same dependency on a conceptual level, specifically a lower-layer component requesting and receiving symbols (functionality, data) from a higher-layer component.

Causes and Consequences. Upwards dependencies can arise if a new functionality is introduced at a higher layer and found useful for a lower layer. Instead of restructuring the system, it is faster and less expensive to introduce an upwards dependency—even though it is not intended to exist. Similarly, during refactoring, a large component located at a higher layer may be partitioned into smaller ones that are relocated into lower layers. This process is error-prone and can lead to upwards dependencies if some of the relocated components still depend on the ones remaining in the higher layer. If downwards data flow is involved, the main cause for upwards dependencies is technical debt; developers opting for an easy solution that is detrimental to the system's design to meet a deadline [20]. Moreover, reversing the data flow is used as a workaround in cases in which unidirectional data flow becomes complex due to components requiring information at different times while executing the system. Finally, a component may not sufficiently abstract and hide its data, exposing the data to lower-layer components. Upwards dependencies can negatively impact the complexity and maintainability of a system. Also, they can lead to unexpected behavior, since requests and data are moving into their reverse directions. For instance, this can introduce serious semantic errors due to a loss of control if different unintended components can modify data or call any functionality. If upwards dependencies are not handled, they can easily lead to cyclic dependencies (cf. Section 5.2).

Resolution Strategies. Our interviewees stated that this type of unintended dependencies can be resolved by reorganizing functionalities. For instance, the functionality causing the dependency can be extracted and relocated into another component. Where to locate the functionality (e.g., a lower-layer component, new component with different programming language) depends on the design principles employed and the programming languages of the components. Another solution is to introduce a duplicate of the functionality, but code duplication is typically considered problematic, too [23]. In parallel, our participants noted that they struggled to resolve

upwards dependencies that are solely connected to data flow, which implies multi-lingual, inter-component data-flow analyses may be an interesting research direction.

5.2 Cyclic Dependency (7)

A cyclic dependency is a well-known architecture smell [18, 32, 38] that occurs if two or more components depend on each other directly or indirectly, creating a loop among them. We display such a situation in Figure 2b, in which the cycle is introduced by component C calling component A even though component A is already calling component C via component B. Note that a cyclic dependency may be introduced due to an upward-functionality dependency (cf. Section 5.1) if the components span different layers. However, as in our example in Figure 2b, the components may also be located in the same layer. In this case, the cyclic dependency does not involve an upwards dependency.

Causes and Consequences. Cyclic dependencies are often caused by the complexity of large (multi-lingual) software systems and a sub-optimal separation of concerns. Particularly legacy systems and components that grow and evolve are prone to cyclic dependencies. As a result, the maintainability of components and systems decreases further, because changes in one component typically cause ripple effects that require changes in other components, too. Finally, cyclic dependencies may introduce serious runtime problems like deadlocks, making the system unreliable.

Resolution Strategies. Essentially, cyclic dependencies can be resolved the same way as upward-functionality dependencies (cf. Section 5.1). So, functionality must be relocated to get rid of this unintended dependency. As for upward-functionality dependencies, the multi-lingual nature of a system may restrain the options for relocating individual functionalities compared to a mono-lingual system. For instance, a functionality may not be transferable to another programming language, even if that language would be better suited (e.g., more performant), because this would introduce further indirections with consequent (unintended) dependencies.

5.3 God-Component Dependency (6)

God components are conceptually identical to the well-known god-class or long-class smell [27]. A god component is large and highly coupled internally, exhibiting many unrelated or uncategorized functionalities (cf. Figure 2c). Due to the lack of separation of concerns, god components cause many unintended and illogical dependencies with other components.

Causes and Consequences. God components and their many dependencies often arise from legacy code that has not been updated or was written before design patterns and clear reference architectures were used. They can also emerge if deadlines must be met, emphasizing fast delivery over maintaining the architecture of the system. The many and illogical dependencies associated with a god component make it difficult to understand, maintain, analyze, and extend the god and all depending components—problems further challenged by different programming languages being involved.

Resolution Strategies. Unfortunately, splitting up god components to reduce the number of dependencies is challenging, expensive, and may cause new unintended dependencies. In particular, our interviewees experienced that a god component is hard to split up incrementally, which may cause even more problems compared to a complete rework. Moreover, god components typically require developers to investigate many different components that are involved in the dependencies and more source code (potentially in different programming languages) compared to other types of unintended dependencies.

5.4 Component Configuration Dependency (2)

A component configuration is a set of constants and settings used to define the interaction between the software of a system, the hardware on which it is deployed, and the network that it uses. According to ASML's architecture, such settings are specified in the bridge that allows components to interact with each other. A component configuration dependency manifests when a component depends on the configuration settings of another component as a reuse mechanism. However, the configuration is usually component-specific and reusing it can cause unexpected behavior. For instance, in Figure 2d component B uses the configuration of component A. If the hardware on which the two are deployed is different, component B may not behave as envisioned.

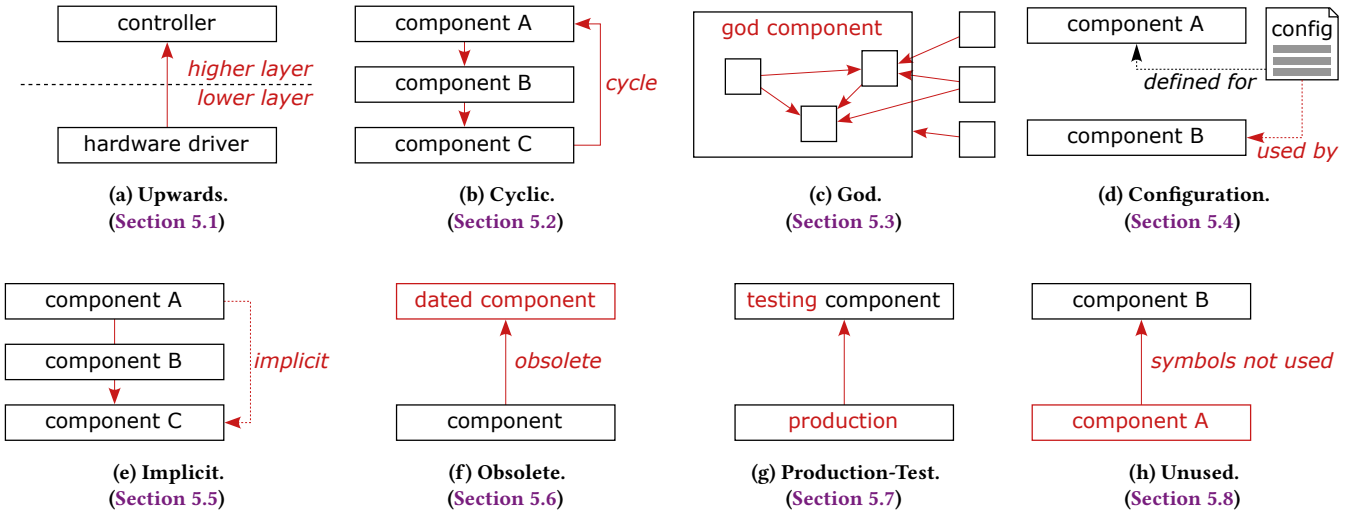
Causes and Consequences. Managing a single set of configurations and not multiple can be handy. Consequently, developers are tempted to depend on the configurations established in existing components, especially if such configurations use similar resources to the ones of their component. Also, a lack of understanding regarding the architecture of the system can lead to the believe that configuration settings for two components are identical. However, this may not be the case and any change in a configuration may silently impact other components, thereby breaking functionalities.

Resolution Strategies. The simplest solution for this unintended dependency is to define an own configuration for each component. An even cleaner solution would be to implement some form of configuration inheritance or templates that allow to reuse configurations while also adapting them. Still, according to our interviewees, raising the awareness of developers about this problem and about current practice are the main challenges for avoiding or resolving this unintended dependency.

5.5 Implicit Dependency (2)

An implicit dependency connects two components without being specified in the build scope definition file. Such implicit dependencies are unintended, since they have never been specified. As we exemplify in Figure 2e, implicit dependencies occur due to transitive dependencies between components (brought all together by explicit dependencies), reflection, or inheritance in the execution environment of a system [34].

Causes and Consequences. Functions that can be accessed through implicit dependencies are made available to developers by language workbenches and language features, such as code completion and inheritance. In such cases, no action by the developers is required to make the functionality available within a component, and thus the explicit dependency definition is not needed. Still, any change to a provided symbol or to a factor that enables the implicit dependency in a component may result in unexpected behavior.



(a) Upwards. (Section 5.1) **(b) Cyclic.** (Section 5.2) **(c) God.** (Section 5.3) **(d) Configuration.** (Section 5.4)

(e) Implicit. (Section 5.5) **(f) Obsolete.** (Section 5.6) **(g) Production-Test.** (Section 5.7) **(h) Unused.** (Section 5.8)

Figure 2: Examples of the eight unintended dependencies in our catalog, marked with red arrows and labels.

Resolution Strategies. Detecting implicit dependencies in multi-lingual systems currently requires a manual analysis, which is costly, time intensive, and error-prone. While some assistance through static code analysis is available, more mature techniques are needed in industry. Additionally, tracing between externally used functionality and explicit dependencies may be useful as a baseline to detect implicit dependencies. Otherwise, developers need to manually identify and specify implicit dependencies between multi-lingual components.

5.6 Obsolete Dependency (2)

An obsolete dependency (cf. Figure 2f) is a dependency that is:

dated: a new release of the component is available;
inactive: the maintenance of a component as a whole has stopped or the project has been abandoned, or;
deprecated: the developers of the component discourage others to use the dependency.

In any case, the dependency has become obsolete and should be removed from the system.

Causes and Consequences. Removing dependencies may result in breaking changes if the client component still requires the requested functionality or data and has not been updated accordingly. Thus, it may be easier and more reliable to simply keep obsolete dependencies until the developers are sure that these are not relevant within the system anymore. The lack of tools and support when it comes to anticipating such scenarios increases the uncertainty and stagnates maintenance. Moreover, a lack of complete or maintained documentation can also play a role [25, 26], since it hinders knowledge acquisition to proceed with removing obsolete dependencies and re-engineering the impacted components. Consequently, the client component may incur technical debt (costs incurred due to postponing fixes or needed code changes [14]) and technical lag (lag between the expected dependency release and its used version in the client component [20]).

Resolution Strategies. To cope with an obsolete dependency, either the client or the provider component must be changed. Changing the provider can be done by updating the current functionality or providing alternative functionalities that can replace deprecated ones. Changing the client means that the dependency is upgraded to a newer version of the provider component (fighting technical lag), migrated to an active provider component, or the deprecated functionality is replaced by a different one within the same provider component. Knowing when to perform any of these changes or their impact in terms of security vulnerabilities, breaking changes, or bugs in multi-lingual systems are the most critical challenges mentioned by our interviewees.

5.7 Production-Test Dependency (2)

A production-test dependency occurs when production components depend on functionalities provided by testing components (cf. Figure 2g). While promoting code reuse and avoiding duplication are well-established design principles, dependencies from code in production to code in testing components should not be established. Specifically, testing code should rely on production code to test its behavior and reliability, but not the other way around.

Causes and Consequences. This unintended dependency usually occurs when functionality required to test the production code is introduced in the system's tests. Logically, such functionality may be useful in the production code, too. For instance, one of our interviewees mentioned the implementation of additional code to calibrate printing facilities of a machine during testing. Later, the production code started to depend on this functionality, aiming to reuse the code and avoid code duplication. However, this decision undermined the separation of concerns in the system design, involved unexpected code into the production system, and required that testing code would be managed as production code.

Resolution Strategies. A simple way to resolve this unintended dependency is to extract the valued functionality from the testing component and move it into a production component, which may

involve changing the programming language of the code. This way, the code can be reused without introducing code duplication. So, the developers can safeguard that the system is designed as intended.

5.8 Unused Dependency (2)

Unused dependencies refer to cases in which a dependency between two components is explicitly defined, but the symbols of the provider component are not actively used in the client component (cf. Figure 2h). For instance, a client component may request a variable, but then never use this variable within its implementation. Too many unused dependencies easily result in so-called bloated dependencies, dependencies that are compiled together with the client component's code by a build tool but that are not required at compilation or runtime [40].

Causes and Consequences. Similarly to obsolete dependencies (cf. Section 5.6), software engineers could be afraid to break functionality that is currently in place because they are not sure if they can safely remove an unused dependency. Software maintenance activities like refactoring or planning the introduction of a new dependency can also cause an unintended dependency. Another cause are developers removing the provider component's uses, but forgetting to remove the specification of the dependency (i.e., in the DSL and build scope definition file). As a result, software developers may be unaware of the consequent change, which can remain unnoticed in the client component's code.

Resolution Strategies. The main challenge for removing unused dependencies lingers in accurately identifying such dependencies in multi-lingual components. This can be performed automatically via tooling that verifies whether there are any provider component uses in the client component's code (besides the import statements). It can also be done by providing detection support via the programming language itself, which is the case for Go: Go generates a compilation error when unused dependencies are imported [13]. Luckily, after an unintended dependency is identified, its removal from the import statements and other specification files is trivial.

6 DISCUSSION

We now reflect on our catalog by first answering our research questions before discussing the unintended dependencies' properties, connections to existing research, and implications. We also underpin our discussion with anonymous quotations (italic in quotes) for which we display example codes (bold in square brackets) we assigned during our thematic analysis (cf. Section 4.2).

6.1 Answering the Research Questions

RQ_1: Types of Unintended Dependencies. Overall, we identified eight different types of unintended dependencies for the multi-lingual systems at ASML (cf. Section 5). As we already assumed, most of these types are similar or identical to architecture and code smells in mono-lingual systems (e.g., cyclic dependency) or represent violations of general software-design recommendations (e.g., upwards dependency). In contrast, we noticed that the multi-lingual nature of ASML's systems adds additional complexity to these dependencies and challenges particularly their resolution. So, we argue that our catalog contributes a complementary overview of unintended dependencies in software systems that expands on the related work. We discuss the differences between mono-lingual and multi-lingual systems regarding the unintended dependencies we identified in more detail within Section 6.2.

RQ_2: Causes and Consequences. We found different causes for which unintended dependencies are introduced in a system. Most prominently, some functionality that developers want to reuse is available in some component, but establishing the respective dependency violates a design rule. However, re-engineering the functionality into an own or fitting component requires additional time, is expensive, is error prone, and may require a language change. As a consequence, re-engineering components is often not of immediate interest for management or the developers, who first focus on delivering a working production system. In particular, they risk to introduce an unintended dependency to meet a deadline:

> *"I think you will not find any software developer or architect that is fine with these dependencies. I think we all want to solve them, and I think that's something we all have in common. [...] But then if you look at the amount of other features that we have to implement for each delivery and with the tight deadlines that we have, the first things to go are these kind of improvement actions."* **[codes: cause]**

One interviewee summarized why unintended dependencies exist:

> *"That's why for me, it's not really unintended. It was intended taking the risk and those things are still there."* **[codes: cause]**

Moreover, while the unintended dependencies may violate specified design rules, they are not by default causing a system to misbehave or cause an error:

> *"Trust me, the current code is working with its violations. [...] I'm in software, tell me what is wrong with this. I know there is a rule that is being violated, but so what, why is it wrong? Apparently it is not so bad, because for four years it was there [...]"* **[codes: consequence]**

Consequently, it is not of utmost importance for most developers to immediately resolve such dependencies, until they actually cause misbehavior or the negative consequences of complicating program comprehension and maintenance accumulate too much (i.e., technical debt [29]). In summary, the causes and consequences of unintended dependencies are comparable to those of software smells.

RQ_3: Resolution Strategies. Most of the unintended dependencies we identified for multi-lingual systems require the developers to restructure the system architecture, moving functionalities between components, layers, and even subsystems. While this is similar to mono-lingual systems, we found that the resolution strategies typically are more complicated. In particular, moving functionalities between programming languages is more complicated than if they could simply be reused as they are. The developers have to rewrite the code in such cases, which can be very cumbersome due to specifics of the languages and the technical environment:

> *"[...] as far as I remember if you have dependencies in a C++ [component] you cannot use the external component without*

having it defined in the build scope definition. And the build scope definition is being passed, the whole repo is being passed [...] In Python you can import also the dependencies that you don't have in this build scope definition, and this is a bit more tricky, then someone manually need[s] to notice it. I believe we don't have [an] automated way to find it [...]"

[codes: detect, build scope, multi-technology]

Additionally, the functionality may have been implemented in a certain language to fulfill a specific requirement (e.g., performance), so moving to another language may not be ideal. Such factors also connect to budgeting, since resolving unintended dependencies between languages is often very expensive and while ASML aims to resolve them, the costs are sometimes too high:

"[...] so we have one cyclic dependency which is between two classes. One in C and second in C++. This situation is a little bit connected to how our components are built, with different languages. This component was in the past one component that was written in C, and then we extracted part of it, and then it became C++ and C. Then we have some dependencies like cyclic between them, because one depended on the other, and vice versa. Which is now really hard to change as we cannot change this code that is written in C, because when we want to change it, it will require us to make some bigger changes [which] takes too much time. When we estimated it, this was 100 story points and no one would give us that amount of budget for fixing one cyclic dependency, at least not for now."

[codes: god, cyclic, multi-technology, resolution]

To summarize, it seems more challenging in multi-lingual systems to decide when and how to resolve unintended dependencies, because there are additional factors developers have to consider.

6.2 Mono-Lingual Versus Multi-Lingual Systems

Reflecting on the answers to our researcher questions, our findings suggest that unintended dependencies in multi-lingual systems do not differ dramatically from the ones reported for mono-lingual systems. For instance, cyclic dependencies, god-component dependencies, unused dependencies, and implicit dependencies are well-known software smells [6, 18, 34, 38]. In fact, after analyzing our catalog, we argue that we could transfer all unintended dependencies easily to mono-lingual systems. However, by addressing **RQ$_3$**, we noticed additional challenges when it comes to detecting and resolving these dependencies in multi-lingual systems, which deserve further investigation form the research community.

Detecting Unintended Dependencies. For detecting unintended dependencies in mono-lingual systems, diverse static and dynamic analysis techniques (e.g., using data-flow analysis, control-flow graphs, dependency graphs) have been proposed in research [16, 19, 38]. However, applying these techniques to industrial multi-lingual systems and building the underlying data structures needed is not trivial. In particular, providing a framework that, among others, manages multi-lingual dependencies, includes configuration settings, covers multiple components in different languages, provides a multi-lingual representation of the analysis results, and scales to large real-world systems, is crucial but also challenging for detecting unintended dependencies in multi-lingual systems. Compared to mono-lingual systems, it is particularly interesting to properly represent multi-lingual dependencies similar to established meta-data files, such as Makefiles (C++) or requirements files (Python). It is an interesting research direction to explore how to provide these representations and analyses for multi-lingual systems.

ASML has tackled these problems by building its internal DSL, the build scope definition file, as well as a tool that runs checks on dependencies within the build system. This tool diagnoses the dependencies of the components within a system and pops up after a build has finished. To make the tool work and detect dependencies, ASML uses an allow-list containing a list of components that can depend on each other. The list itself is based on ASML's reference architecture. However, this can also cause problems, since any dependency that is not defined in this allow-list will be flagged by the tool. In turn, software architects need to manually keep the list up-to-date, which is an error-prone and challenging endeavor. Unfortunately, designing and implementing automation to keep all dependencies between the different tools synchronized, while also ensuring unintended dependencies are either not allowed or only accepted under exceptional cases is a challenging problem. Moreover, if they experience any problems, the developers need to follow a reactive strategy to cope with these issues rather than anticipating the impact of the introduced changes on the system's design via automated analyses. Lastly, more abstract analyses including the detection of the extracted unintended dependencies have not been incorporated into the tool. It would be of great benefit to ASML and similar software companies to develop frameworks and techniques that automate or at least facilitate the management and analyses of such properties of multi-lingual systems.

Resolving Unintended Dependencies. As we discussed in Section 6.1, deciding when and how to resolve multi-lingual dependencies is challenging and expensive—raising new problems compared to mono-lingual systems. We argue that this is one of the key differences regarding dependencies in mono-lingual and multi-lingual systems. For example, resolving upwards or cyclic dependencies must take into account that the involved components may be in different programming languages. Then, the functionality causing a dependency cannot always be relocated freely without re-implementing it in another language, which may have other drawbacks like performance bottlenecks or needing to relocate additional components that had a dependency to that component. Balancing the pros and cons of the resolution strategies is more complicated in multi-lingual systems compared to mono-lingual ones. Consequently, further research on the different resolution strategies for unintended dependencies in multi-lingual would be highly beneficial, including the definition of resolution patterns, corresponding automation, and analyses to assess the pros and cons depending on relevant non-functional requirements.

6.3 Benefits of Resolving the Dependencies

Unintended dependencies are violations of recommendations and rarely breaking a system. As such, they are still used if time is scarce and sometimes live for a long time within a system. Consequently, as for many types of smells, the question arises why to resolve such dependencies considering the high costs and challenges in the context multi-lingual systems? During our interviews, we were

able to collect various benefits based on our interviewees' experiences, which support that resolving such dependencies and smells is beneficial in the long run. For instance, avoiding unintended dependencies leads to a more robust system, meaning that it is less prone to breaking changes and becomes more reliable:

> *"Well, the benefits would be that it would be more robust, the system in general. Because if you don't depend on a lot of things, basically the things that you can break are mathematically limited. [...]"* **[codes: benefit, robustness]**

Other interviewees mentioned that code with fewer unintended dependencies is easier to comprehend, maintain, and test:

> *"[...] it will make our code simpler and then make it easier to maintain. Also, this will make it easier to test, because when we have cyclic dependencies one class is relying on another, so I imagine that creating a proper unit test for that will be hard."* **[codes: benefit, comprehension, maintenance, test]**

Another interviewee emphasized these points particularly with respect to testing and avoiding runtime errors like deadlocks:

> *"Your components are better testable in isolation. So you can quickly test your changes and therefore develop your components more quickly. It gives [a] better understanding of the system. If the dependencies are few, then it's easier to also oversee the context, or even the system as a whole of how it is supposed to work. If everything is top down then there are no deadlocks or surprise behaviors in your system. And it will favor your locality of change. [...] I think it's easier to change also your component in a backwards compatible way, so you can make safe changes. Without rippling it out throughout the system."* **[codes: benefit, test, deadlock, safety]**

The reasons for such benefits are logical. A system becomes more modular, less tangled, and its components have a clear task:

> *"For sure. You have better control over the system. It doesn't become a spaghetti system. Yeah the behavior is easier to understand on a top level. The flow of the information is easier to understand. The system is better functionally [split]. Also you don't have really responsibility [split] between the different parts of the system. You have everything in a one place, in the best situation. So I would say it's very important to have your dependencies correct."* **[codes: benefit, separation of concerns]**

In summary, while it is often not the primary concern and seems unnecessary since the system behaves correctly, most interviewees see clear benefits in resolving unintended dependencies. We argue that these benefits also hold true for quality issues in mono-lingual systems, providing real-world evidence on those lines of research.

6.4 Implications

We outline below key implications for practitioners and researchers.

For Practitioners. We noticed that some unintended dependencies seem linked to the design of the component bridges and other artifacts (e.g., allow-list) used to define legit dependencies as well as to configuration settings. Thus, software architects and software engineers should carefully assess the design and implementation of the communication between components written in different programming languages. For instance, they need to check whether the communication protocol among multi-lingual components is sufficient. Moreover, how to define the actual dependencies is essential, particularly with respect to what dependencies are allowed, how to enforce that the corresponding rules are uphold, and how to synchronize these between relevant artifacts.

On a final note, regarding any software system, developers face the dilemma between maintaining legacy code (fighting technical debt and lag) and introducing new features that add value to customers. For instance, one interviewee mentioned that

> *"[maintenance] adds no value immediately, but if you look at the long term, it makes development of software much easier, much faster, and much safer."* **[codes: benefit, maintenance]**

Another interviewee added that

> *"the biggest challenge is that [maintenance] is not something that adds direct value to the customer. So it's not very easy to find the budget for this."* **[codes: maintenance]**

To tackle this issue, an organization as a whole must value maintenance as an ongoing process to improve quality in the long run.

For Researchers. Managing dependencies in software systems is challenging, and dealing with unintended ones in multi-lingual systems poses even more complexity. Throughout this paper, we have sketched directions for future research that are based on industrial needs, such as developing new techniques for identifying unintended dependencies and (automatically executable) patterns for resolving these. Our catalog of unintended dependencies and the corresponding insights provide a stepping stone towards this direction, emphasizing the challenges of dealing with complex multi-lingual systems in practice. Furthermore, our findings provide real-world supportive evidence on the negative consequences of software smells and design flaws, while also highlighting the benefits of resolving them. In this context, we underpin the need for lifting existing techniques for dealing with such quality issues from mono-lingual to multi-lingual systems to facilitate developers' tasks. Also, similar to software smells in mono-lingual systems, research on the evolution, impact, and accumulation of unintended dependencies in multi-lingual systems is an open research direction.

7 THREATS TO VALIDITY

As any qualitative survey building on interviews at one company, the generalizability of our results is inherently limited. In particular, there may be potential biases involved due to the teams' or company's perspective on certain code structures and coding standards. Moreover, our interviewees are employees at ASML who have roles related to software engineering. Although the questions are specific to software engineering, their answers are also narrowed to these specific roles. Other insights may be identified when interviewing other roles on the topic. Since our research is specific to ASML, the types of unintended dependencies we identified may not be applicable to every other system. Especially, ASML operates in a highly specialized domain and has implemented its own solution

for establishing dependencies between multi-lingual components. Thus, the unintended dependencies we identified may not apply to other domains or companies. Nonetheless, since our findings align to existing research and provide industrial experiences that were lacking in these directions, we argue that such threats are small and our contributions are valuable for practitioners and researchers.

8 CONCLUSION

In this paper, we have investigated eight types of unintended dependencies in multi-lingual software systems at ASML by interviewing 17 practitioners. Not surprisingly, we found that the types of dependencies in multi-lingual systems are often similar or even identical to smells in mono-lingual systems. However, we found and showcased that unintended dependencies in multi-lingual systems can be more problematic to resolve and may cause more severe problems than in mono-lingual systems. Such findings provide supportive real-world evidence on the severity of smells and unintended dependencies in software engineering. Moreover, they highlight the need to lift existing techniques for identifying, managing, and resolving unintended dependencies for mono-lingual towards multi-lingual systems. With our contributions, we provide helpful advice for practitioners on managing unintended dependencies. For researchers, we contribute a stepping stone for future research and new techniques in this direction.

ACKNOWLEDGMENTS

We thank all of our interviewees for contributing to our work and allowing us to build on their experiences.

REFERENCES

[1] Hani Abdeen, Stéphane Ducasse, Houari Sahraoui, and Ilham Alloui. 2009. Automatic Package Coupling and Cycle Minimization. In *Working Conference on Reverse Engineering (WCRE)*. IEEE, 103–112. https://doi.org/10.1109/WCRE.2009.13

[2] Mouna Abidi, Manel Grichi, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2019. Code Smells for Multi-Language Systems. In *European Conference on Pattern Languages of Programs (EuroPLop)*. ACM. https://doi.org/10.1145/3361149.3361161

[3] Mouna Abidi, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2019. Anti-Patterns for Multi-Language Systems. In *European Conference on Pattern Languages of Programs (EuroPLop)*. ACM, 1–14. https://doi.org/10.1145/3361149.3364227

[4] Mouna Abidi, Md Saidur Rahman, Moses Openja, and Foutse Khomh. 2021. Are Multi-Language Design Smells Fault-Prone? An Empirical Study. *ACM Transactions on Software Engineering and Methodology* 30, 3 (2021), 29:1–56. https://doi.org/10.1145/3432690

[5] Khalid Alkharabsheh, Yania Crespo, Esperanza Manso, and José A. Taboada. 2019. Software Design Smell Detection: A Systematic Mapping Study. *Software Quality Journal* 27 (2019), 1069–1148. https://doi.org/10.1007/s11219-018-9424-8

[6] Umberto Azadi, Francesca A. Fontana, and Davide Taibi. 2019. Architectural Smells Detected by Tools: A Catalogue Proposal. In *International Conference on Technical Debt (TechDebt)*. IEEE, 88–97. https://doi.org/10.1109/TechDebt.2019.00027

[7] Virginia Braun and Victoria Clarke. 2006. Using Thematic Analysis in Psychology. *Qualitative Research in Psychology* 3, 2 (2006), 77–101. https://doi.org/10.1191/1478088706qp063oa

[8] Yulu Cao, Lin Chen, Wanwangying Ma, Yanhui Li, Yuming Zhou, and Linzhang Wang. 2023. Towards Better Dependency Management: A First Look at Dependency Smells in Python Projects. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1741–1765. https://doi.org/10.1109/TSE.2022.3191353

[9] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. 2009. Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Transactions on Software Engineering* 35, 6 (2009), 864–878. https://doi.org/10.1109/TSE.2009.42

[10] Ching-Chi Chuang, Luis Cruz, Robbert van Dalen, Vladimir Mikovski, and Arie van Deursen. 2022. Removing Dependencies from Large Software Projects: Are You Really Sure?. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 105–115. https://doi.org/10.1109/SCAM55253.2022.00017

[11] Timothy Clem and Patrick Thomson. 2021. Static Analysis at GitHub: An Experience Report. *Queue* 19, 4 (2021), 42–67. https://doi.org/10.1145/3487019.3487022

[12] Juliet Corbin and Anselm Strauss. 2014. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage.

[13] Russ Cox. 2019. Surviving Software Dependencies. *Communications of the ACM* 62, 9 (2019), 36–43. https://doi.org/10.1145/3347446

[14] Ward Cunningham. 1992. The WyCash Portfolio Management System. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA-Addendum)*. ACM, 29–30. https://doi.org/10.1145/157709.157715

[15] Veronika Dashuber, Michael Philippsen, and Johannes Weigend. 2021. A Layered Software City for Dependency Visualization. In *International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP)*. SCITEPRESS, 15–26. https://doi.org/10.5220/0010180200150026

[16] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. 2020. Escaping Dependency Hell: Finding Build Dependency Errors with the Unified Dependency Graph. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 463–474. https://doi.org/10.1145/3395363.3397388

[17] Graham R. Gibbs. 2007. *Analyzing Qualitative Data*. Chapter Thematic Coding and Categorizing, 38–56. https://doi.org/10.4135/9781849208574

[18] Philipp Gnoyke, Sandro Schulze, and Jacob Krüger. 2021. An Evolutionary Analysis of Software-Architecture Smells. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 413–424. https://doi.org/10.1109/ICSME52107.2021.00043

[19] Philipp Gnoyke, Sandro Schulze, and Jacob Krüger. 2023. On Developing and Improving Tools for Architecture-Smell Tracking in Java Systems. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 248–253. https://doi.org/10.1109/SCAM59687.2023.00034

[20] Jesus M. Gonzalez-Barahona, Paul Sherwood, Gregorio Robles, and Daniel Izquierdo. 2017. Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is. In *International Conference on Open Source Systems (OSS)*. Springer, 182–192. https://doi.org/10.1007/978-3-319-57735-7_17

[21] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. 2018. An Extensible, Regular-Expression-Based Tool for Multi-Language Mutant Generation. In *International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. ACM, 25–28. https://doi.org/10.1145/3183440.3183485

[22] Monique M. Hennink, Bonnie N. Kaiser, and Vincent C. Marconi. 2017. Code Saturation Versus Meaning Saturation: How Many Interviews Are Enough? *Qualitative Health Research* 27, 4 (2017), 591–608. https://doi.org/10.1177/1049732316665344

[23] Judith F. Islam, Manishankar Mondal, and Chanchal K. Roy. 2016. Bug Replication in Code Clones: An Empirical Study. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 68–78. https://doi.org/10.1109/SANER.2016.78

[24] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. 2022. Dependency Smells in JavaScript Projects. *IEEE Transactions on Software Engineering* 48, 10 (2022), 3790–3807. https://doi.org/10.1109/TSE.2021.3106247

[25] Jacob Krüger and Regina Hebig. 2020. What Developers (Care to) Recall: An Interview Survey on Smaller Systems. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 46–57. https://doi.org/10.1109/ICSME46990.2020.00015

[26] Jacob Krüger and Regina Hebig. 2023. To Memorize or to Document: A Survey of Developers' Views on Knowledge Availability. In *International Conference on Product Focused Software Process Improvement (PROFES)*. Springer, 39–56. https://doi.org/10.1007/978-3-031-49266-2_3

[27] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations. *Journal of Systems and Software* 167 (2020), 110610:1–36. https://doi.org/10.1016/j.jss.2020.110610

[28] James Lewis and Martin Fowler. 2014. Microservices: A Definition of this New Architectural Term. https://martinfowler.com/articles/microservices.html

[29] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A Systematic Mapping Study on Technical Debt and its Management. *Journal of Systems and Software* 101 (2015), 193–220. https://doi.org/10.1016/j.jss.2014.12.027

[30] Jimmie Manning. 2017. In Vivo Coding. *The International Encyclopedia of Communication Research Methods* 24 (2017), 1–2. https://doi.org/10.1002/9781118901731.iecrm0270

[31] Tom Mens. 2016. Research Trends in Structural Software Complexity. *CoRR* (2016), 1608.01533:1–10. https://doi.org/10.48550/arXiv.1608.01533

[32] Haris Mumtaz, Paramvir Singh, and Kelly Blincoe. 2021. A Systematic Mapping Study on Architectural Smells Detection. *Journal of Systems and Software* 173 (2021), 110885:1–28. https://doi.org/10.1016/j.jss.2020.110885

[33] Andrew Neitsch, Kenny Wong, and Michael W. Godfrey. 2012. Build System Issues in Multilanguage Software. In *International Conference on Software Maintenance (ICSM)*. IEEE, 140–149. https://doi.org/10.1109/ICSM.2012.6405265

[34] Lina Ochoa, Thomas Degueule, and Jurgen Vinju. 2018. An Empirical Evaluation of OSGi Dependencies Best Practices in the Eclipse IDE. In *International Coference on Mining Software Repositories (MSR)*. ACM, 170–180. https://doi.org/10.1145/3196398.3196416

[35] José Pereira dos Reis, Fernando Brito e Abreu, Glauco de Figueiredo Carneiro, and Craig Anslow. 2022. Code Smells Detection and Visualization: A Systematic Literature Review. *Archives of Computational Methods in Engineering* 29, 1 (2022), 47–94. https://doi.org/10.1007/s11831-021-09566-x

[36] Mark Richards. 2015. *Software Architecture Patterns*. O'Reilly.

[37] Ganesh Samarthyam, Girish Suryanarayana, and Tushar Sharma. 2016. Refactoring for Software Architecture Smells. In *International Workshop on Software Refactoring (IWoR)*. ACM, 1–4. https://doi.org/10.1145/2975945.2975946

[38] Darius Sas, Paris Avgeriou, and Francesca Arcelli Fontana. 2019. Investigating Instability Architectural Smells Evolution: An Exploratory Case Study. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 557–567. https://doi.org/10.1109/ICSME.2019.00090

[39] Tushar Sharma and Diomidis Spinellis. 2018. A Survey on Software Smells. *Journal of Systems and Software* 138 (2018), 158–173. https://doi.org/10.1016/j.jss.2017.12.034

[40] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem. *Empirical Software Engineering* 26, 3 (2021), 45:1–44. https://doi.org/10.1007/s10664-020-09914-8

[41] Klaas-Jan Stol and Brian Fitzgerald. 2018. The ABC of Software Engineering Research. *ACM Transactions on Software Engineering and Methodology* 27, 3 (2018), 11:1–51. https://doi.org/10.1145/3241743

[42] Per E. Strandberg. 2019. Ethical Interviews in Software Engineering. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–11. https://doi.org/10.1109/ESEM.2019.8870192

[43] Samuel J. Stratton. 2021. Population Research: Convenience Sampling Strategies. *Prehospital and Disaster Medicine* 36, 4 (2021), 373–374. https://doi.org/10.1017/S1049023X21000649

[44] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. 2001. The Structure and Value of Modularity in Software Design. *ACM SIGSOFT Software Engineering Notes* 26, 5 (2001), 99–108. https://doi.org/10.1145/503271.503224

[45] Hongyi Sun, Waileung Ha, Pei-Lee Teh, and Jianglin Huang. 2017. A Case Study on Implementing Modularity in Software Development. *Journal of Computer Information Systems* 57, 2 (2017), 130–138. https://doi.org/10.1080/08874417.2016.1183430

[46] Eva van Emden and Leon Moonen. 2002. Java Quality Assurance by Detecting Code Smells. In *Working Conference on Reverse Engineering (WCRE)*. IEEE, 97–106. https://doi.org/10.1109/WCRE.2002.1173068

[47] Thomas Zimmermann. 2016. Card-Sorting: From Text to Themes. In *Perspectives on Data Science for Software Engineering*. Elsevier, 137–141. https://doi.org/10.1016/b978-0-12-804206-9.00027-1