A Visualization for Client-Server Architecture Assessment

Nour Jihene Agouf[1], Soufyane Labsari[2]. Stéphane Ducasse[2], Anne Etien[2], Nicolas Anquetil[2]
1: Arolla and Univ. Lille, CNRS, Inria, Centrale Lille
2: Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL

***Abstract*—Maintaining large legacy systems often requires understanding their architecture. This is important since legacy system architecture decay over time and architecture violations may dramatically impact planned renovation actions. Merely reading source files is time-consuming and often highly inefficient. Visualizations have been proposed as a tool to support architecture understanding. Some software architecture visualizations decompose the software system architecture into layers, components, or slices from a structural viewpoint. Such visualizations, however, do not take into account the specificities of client-server applications. They do not help maintainers identify and understand software architecture violations. In this paper, we propose CLISERVO, a new visualization to help software maintainers detect architectural violations in client-server systems. CLISERVO classifies client-server entities into different levels of dependencies, shared entities, or ambiguous entities (*e.g.,* entities that belong abnormally to different layers). CLISERVO identifies and presents entities in their corresponding layers from two distinct *viewpoints*: *global overview* entities and *violations*, *i.e* ambiguous entities and illegal dependencies between layers. We validated our approach on three real-world industrial projects with access to their maintainers. We report the findings of 91 ambiguous entities, 29 purportedly shared and idle entities, 24 and 82 elements defined as shared but only used by the client or server, and 12 relations violating the layered architecture.**

***Index Terms*—Client-server, visualization, architectural violation**

I. Introduction

During the software lifecycle, the architecture often becomes inaccurate which results in architectural erosions [28], [31]. Consequently, recovering the existing architecture of legacy software is challenging [8]. Over the past decades, tools have been implemented to recover software architecture such as Rigi [33], SNIFF [37], Rose [9], MicroART [15] and ARCADE [35]. However, software architecture is a very fuzzy notion that lives mostly in the mind of the beholder. There is no one-size-fits-all, universal, definition of what is software architecture (see for example the 4+1 model [22]). Moreover, software architecture is materialized by coding conventions (such as class names, package dependencies, etc.) that are often not documented, not explicit in the code, and violated by programmers [21].

A high-level design description plays, de facto, an important role in successfully understanding and reasoning about large and complex software systems [4], [26]. The importance of visualizing software architecture has been extensively investigated as it can be of interest to various stakeholders such as architects, developers, testers, and project managers [13], [19], [36]. Visualizations are widely used and more efficient in representing large-scale software [17], [20], [23], [39]. With thousands of classes and millions of code elements, however, not dedicated visualizations do not scale up.

In this paper, we focus on a certain type of software architecture, *i.e.,* client-server architecture. Such architecture is nowadays very common. It has the advantage that, due to the use of a framework for the communication between the client and the server parts, for some entities, their membership to one part or the other is well-defined and not ambiguous. Note that concerning the ambiguous entities, judging their belonging based on their names or package names can be misleading. Other rules that define the entities belonging, sometimes only known by system experts have to be recovered: as any other rules, with time and after aging, such conventions are often lost or at least violated [6].

We propose a new visualization (CLISERVO [1]) that recovers the high-level architecture of software from software rules (low-level knowledge about the components inter-relations) extracted from the system or known by the experts. It breaks down a software system into layers and components from a structural viewpoint and attributes each component to its corresponding layer.

Applied to real industrial projects, due to their size, visualizations can become complex and not readable if all elements are represented. This is why CLISERVO supports two configurations: Big-Picture and Server-focus combined with two *viewpoints*. The first viewpoint represents all the elements (at the class level). The second one focuses on *rule violations* and only visualizes the elements violating at least one architectural rule.

The proposed approach has been evaluated on three real-world industrial software with validation with their maintainers. Finally, we report the findings of 91 ambiguous entities, 29 purportedly shared and idle entities, and 12 relations violating the layered architecture.

II. A Layered Architecture Identification

The client-server architecture is a distributed application structure that divides tasks or workloads between providers of

We thank Arolla for the funding of Nour J. Agouf's research, and our other industrial partner maintainers for their availability and willingness to perform our experiment.

[1]https://github.com/LABSARI/ClientServer-Visualization

a resource or service, called *servers*, and service requesters, called *clients*. Such an architecture mostly relies on three parts. The *client* part requests content or service from a server. In contrast, the *server* part runs programs to answer client requests. Finally, the shared part, corresponding mainly to the data transfer objects (DTOs), are resources transferred by the client part to the server to execute the programs on a given data or on the opposite transferred by the server to the client to display them. In this section, we show that assigning a class of a given layer is a challenging task.

A. Challenges for Layers Identification

These three parts (client, server, and shared) may eventually be well-identified during the design phase, through architectural rules. Some rules are structural *e.g.,* a DTO inherits from a specific class or a client class implements a dedicated interface. Other rules concern the behavior, *e.g.,* a class for which at least a method is called by a server class, belongs to the server part or, a class for which at least a method calls a client class is considered client side. However, these rules are not always documented. And even so, over time and several evolutions, these rules are violated. Thus in practice, the separation between the different parts is fuzzy and is no more clearly identified. For example, on real systems, it is not rare that some classes belonging to a (sub)package of a client may in fact play the role of server and vice versa. Some DTOs may be only used by one part, or even not used by either part. Finally, other elements that DTO may be shared between the client and the server or may be so complex that some of their methods play the role of the client and others of the server. The belonging of these elements to exactly one part may often need further investigation.

If a class satisfies only an architectural rule defining a layer, it is easy to assign it to this layer. In case a class satisfies several rules corresponding to different layers, there is an *ambiguity* and it is not possible to clearly identify the layer the class belongs to: violations are thus observed, since normally, a class should belong to a single layer.

In the context of a future migration such as the migration from a client application to another (*e.g.,* from GWT to Angular [38]) or the decomposition into micro-services of the server part, the shared elements which are not DTOs or the violations between client and server parts are problematic. In practice, they correspond to violations of software architectural rules.

B. Layers in the Server Part

Even when the client and server parts are clearly separated, for example, because they structurally belong to two different projects, architecture violations may occur. Indeed, the server part may be decomposed into several layers, such as the server interacting with the client part, the services corresponding to the core program, and the database access objects (DAOs) corresponding to the interface between the server and the database. Each part corresponds to a layer and the communications between them are strictly defined. The server elements can call services, which in their turn can call DAOs. All other communication between layers is considered a violation. For example, a DAO is not allowed to have dependencies on services or server elements.

Due to the challenges to assign classes to different architectural parts and understanding architecture violations, there is a need to support maintainers to understand why a class may play different roles and how rules are violated.

III. A Client-Server Architecture Visualization: CliServo

We propose a dedicated visualization, CliServo, to support client-server navigation and architectural violation identification. Figure 1 displays an annotated version of one of the two views of CliServo applied to a real industrial system. The remainder of the paper presents in detail the different aspects of the visualization: its two *configurations*, its two *viewpoints*, and the *levels* that can be applied to support the understanding of application architects.

This visualization structures the software into layers and uses the traditional node-link diagram to connect the layers' components. These layers rely on rules that can be *structural i.e.,* a class belongs to a specific hierarchy, or *behavioral i.e.,* a class calls or is used by another one. The constraints imposed by the membership of a class to a specific hierarchy are stronger (since the behavior and the state of the classes are shared) than those resulting from the use of/by a specific class. Consequently, we consider that the assignment to a layer is sure when it relies on a structural rule, and uncertain when it is based on behavioral rules.

CliServo offers two configurations: first, the *Big-Picture* showing all the parts (client, server, shared and purgatory) (See III-A) and the *Server focus* one showing the server part (See III-B), as we will show now.

A. Configuration 1: Big-Picture with Four Main Layered Parts

In the Big-Picture configuration, CliServo splits the system into four main layered parts (see Figure 2): *Client side*, *Server side*, *Shared Space* and *Purgatory*. The parts are layered because each element implied in a relation may be called by other elements (acting as subsequent layers). We explain each part:

- **Client part:** This layered part contains the core set of classes or interfaces, establishing the communication with the server part (a). In addition, it includes the classes using this core, either directly or indirectly through other classes (c, h). If the core part relies on a structural architectural rule, the remainder is built successively, level by level by considering first the direct clients of these original interfaces and then the direct clients of these resulting classes and so on.
- **Server part:** The server part has a similar structure as the client part. The core part is composed of (implementation) classes that enable communication with the client part (b). The remainder of the server part contains classes used by these original implementations directly

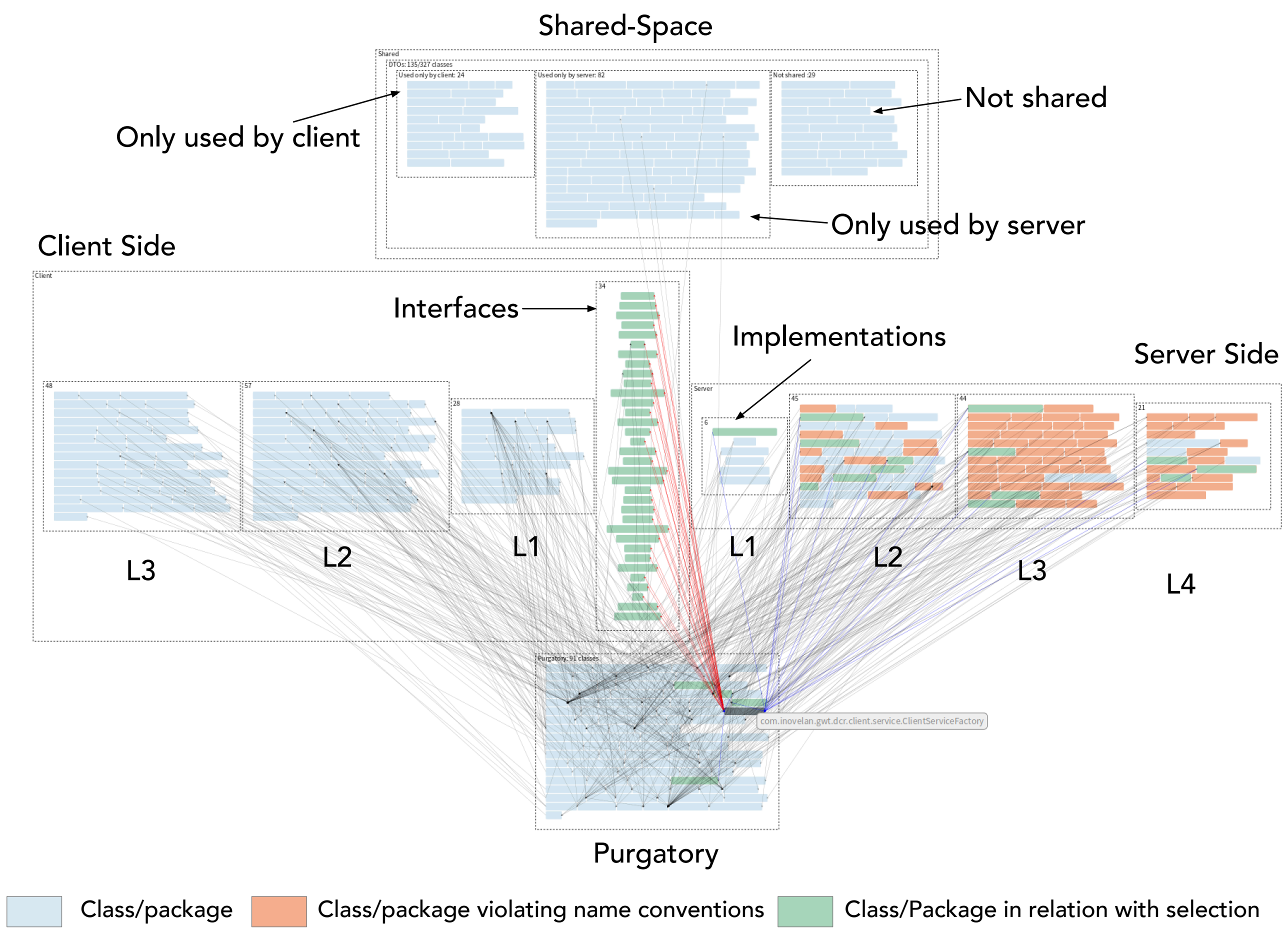

Fig. 1. An annotated version of CLISERVO's Big-Picture from Violation Viewpoint applied to WD: Four parts (*Client*, *Server*, *Shared* and *Purgatory*) and their relation/violation (L = Level). Note that the class names are deliberately blurred to respect the company constraints.

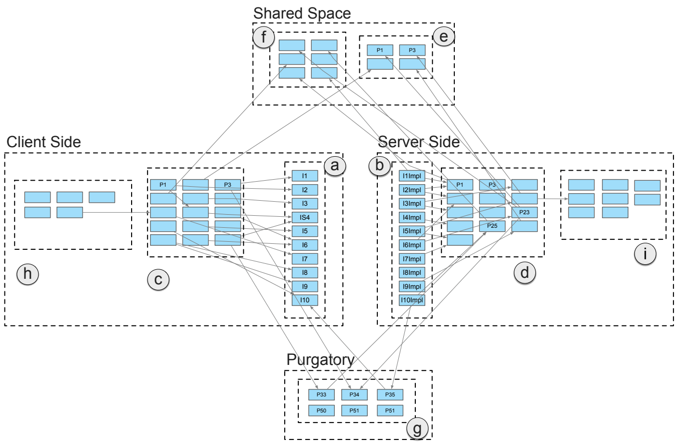

Fig. 2. Big-Picture in CLISERVO: It features four parts (Client Side, Server Side, Shared and Purgatory) – the Server part includes layers b, d and i; the Client part, a, c, h.

or indirectly (d, i). This part is built successively level by level.

- **Shared part:** This part contains DTO (Data Transfer Object) classes *i.e.,* software resources which *can* be used by both client-server entities (e, f). Hence, this part contains DTOs that are *meant* to be shared but may not be actually shared in the project.
- **Purgatory:** The purgatory part gathers elements that are not DTOs and whose classification into the client or the server parts is not clear (g). These elements are used by at least one entity of the server part and use at least one entity of the client part. Such entities are in relation to both client and server entities leading to ambiguity about their layer affiliation. Further analysis of such entities is needed to be attributed to their correct layer.

B. Configuration 2: Server-focus with Three Layers

It is possible to focus only on the server part, which is also composed of layers: *Server*, *Services*, and *DAOs*.

- **Server:** This layer corresponds to what has been previously described in the Big-Picture visualization (Section III-A). It includes the core part, *i.e.,* the (implementation) classes enabling communication with the client

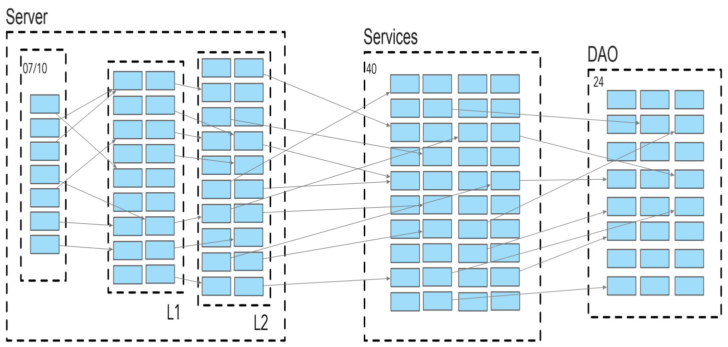

Fig. 3. Illustration of the visualization Server-focus.

part, and contains classes directly or indirectly used by the core (L1 and L2 respectively).
- **Services:** This layer corresponds to the services of the server application. It relies on a structural rule and thus corresponds to a specific class hierarchy.
- **Data Access Object (DAOs):** Entities of this layer represent the classes accessing the database of the software. They also result from a structural rule and represent a specific class hierarchy, such as the AbstractDao class in *Hibernate*.

C. Two Viewpoints for each Configuration

In addition to the *Big-Picture* and *Server-focus* configurations, the visualization features two viewpoints: *a general component overview view* and an *architectural violation view*. Both viewpoints use the same graphical elements (nodes and arrows) to quickly convey information about the software, the main difference is that the second viewpoint does not display all elements of the software but the necessary information to detect violations.

- **Big-Picture in Architectural Violation View.** When looking at *violations* applied to the Big-Picture overview, only the elements in relation to the purgatory are displayed in the client and server parts. Only the classes used by the server part and using the client part are considered ambiguous and are put in purgatory. However, all the elements in relation to these classes require specific attention to determine which relation is a mistake to remove a class from purgatory and put it either in the client or server part. In addition, the DTO part is decomposed into three sets, (i) the DTOs only linked with the client part, (ii) the DTOs only linked with the server part, and (iii) the DTOs linked with neither the client nor the server part. Indeed, since DTOs are considered shared, these three cases correspond to violations. Finally, since the client and server parts mainly rely on behavioral architectural rules, for each class, we check if its position in the client or server part is consistent with the name of the package in which it is. If it is not the case, there is a violation that is expressed by displaying the class with another color. For example, a class in the server part cannot be in a subpackage of a client package.
- **Big-Picture in General Component View.** The Big-Picture in the General Component View depicts ALL entities of the software, meaning entities in violations and entities that are directly or indirectly in relation to the core components (interfaces in client and their implementations in server). Including entities in purgatory. In this mode, the DTO part is not decomposed into three layers but groups all DTO entities in one layer, highlighting the ones in violation in a different color. This view could be of use in case the user wishes to see the violations present in the system and how they interact with the other components.
- **Server-focus in architectural Violation View.** In the Server-focused configuration from the architectural violation viewpoint, only the server elements calling directly a DAO (without going through a class of the Services layer) or called by a Services or DAO element are displayed in the visualization. Similarly, only the service class called by a DAO is displayed. All other elements respect the architectural rules and consequently are not displayed to simplify the visualization and scale.
- **Server-focus in General Component View.** This view focuses on the server part and offers a view of the entire entities in the layers of the server. Including the different violations explained in the violation view. All elements with respect or deviating from the architectural rules are consequently displayed.

In addition to the aforementioned viewpoints, the visualization also offers the hybrid view that allows the selective display of the violation view and general component view in different layers of the software as shown in Figure 5.

D. Nodes, Links, and Interaction

Inside layers, we place nodes to represent classes or packages. Such nodes are connected by links.

Nodes. The visualization presents classes, interfaces, and packages as nodes. Except if the communication protocol or the used framework imposed that the communication between the client and the server parts is performed through interfaces, the other nodes inside the layers correspond to classes or packages. For readability and scalability reasons, if the number of classes to represent in a layer is too high, classes are grouped in their corresponding packages. This threshold can be modified in the settings of the visualization. By default, we use 100.

The visualization is modular in displaying the following layers of the next distances of the core entities, meaning that the decomposition of the parts into layers is built progressively layer proceeding or following a layer for client and server parts, respectively. Consequently, new ambiguous entities in Purgatory may be added at each iteration.[2] when a new distance is computed.

[2]iteration means the addition of a new level representing a behavioral relation between a group of entities and another

Links. Given the use of the conventional node-link diagram, it is evident that an arrowed line means the dependency between interconnected nodes. To avoid overloading the visualization with links, by default, only the dependencies from or to classes of the purgatory or in violation of a rule are displayed. For instance, in Figure 1 the class ClientServiceFactory (selected class in purgatory) depends on all 34 interfaces defined on the client side, hence the red arrows and green color of nodes. On the other hand, 14 class nodes on the server part depend on ClientServiceFactory, hence the blue arrows and green color of nodes.

Interactions. Since the visualization is built as a tool, different interactions with its elements are provided. We list a few of them:

- The user can explore (sub-)nodes or links *i.e.,* investigate their properties and possibly access the corresponding source code.
- The user chooses to hide or show the links coming out or into a single node.
- The user chooses to hide or show all links coming out or into a layer thus improving clarity and not clutter the visualization with links.
- A package node can be expanded to show its internal sub-nodes. Such an expanding feature is also applicable to groups of nodes in layers.
- The visualization provides users with the ability to zoom in on layers and nodes, as well as zoom out to gain a broader perspective.
- The user can progressively display the following nodes of the next dependency level by interacting with it.

IV. Constructing the Visualization

This section provides an in-depth explanation of the construction process for the visualization. This process is automated but requires human intervention to adjust to it specific situations. It shows in particular that the proposed approach is generic enough to be applied to different client-server frameworks. This section delves into the process of identifying parts based on software rules, while also outlining the architectural violations that will be specifically examined.

A. Rule-based Parts/Layer Identification

The visualization relies on an abstract representation of the software using the Famix metamodel [7] and is integrated into the Moose metaplatform [3] [2]. Once the model is built, we import it into the Moose platform.

To construct the visualization, we extract (1) *essential structural rules* (inheritance, interfaces, annotations,...) and (2) *behavioral information* (calls, attribute accesses, ...) from the software entities to build the different parts/layers, both from the model. This extraction is performed using queries over the code model. It means that the visualization elements are populated by executing queries that represent rules that define the different elements of the visualization.

[3]Moose is an extensive platform for software and data analysis: https://moosetechnology.org

Such rules take into account the way the different frameworks expresses client-server relationships - *e.g.,* inheritance to certain classes as in frameworks such as GWT. In addition, these architectural rules are often adapted to reflect the knowledge of the system based its maintainers.

Structural Rules. *In the Big picture configuration*, the application of structural rules extracts the three key components in the visualization: the client core, the server core, and the shared space. The fourth part of the visualization which is the purgatory is based on a behavioral rule which is explained in the next paragraph. The client core corresponds to interfaces and their corresponding implementations as the server core, as depicted by (a) and (b) in Figure 2 respectively.

In the case of the case studies presented in this paper, the structural rules are mainly *inheritance* relations. For example, in WD and OJ based on the GWT RPC[4] protocol, client core (a) consists of interfaces implementing the RemoteService class. On the other hand, the server core (b) consists of classes that satisfy two structural conditions: (1) being direct or indirect subclasses of RemoteServiceServlet, and (2) implementing interfaces descending from the RemoteService class, meaning at least one of the interfaces in the client core.

For example, the use of GWT RPC in the analyzed projects guides the rules followed by the maintainers and provides standardized guidelines for extracting these core components.

The shared space also relies on the use of the GWT RPC framework. Classes of this part are all subclasses of the BaseModelData class provided by the GWT RPC framework.

In the server-focus configuration (Figures 3 and 5), the parts of the visualization are built following similar structural rules. The identification of the server, services, and DAOs also relies on inheritance. For instance, the server core classes all inherit from the RemoteServiceServlet and implement a descendent interface of the RemoteService interface. The services part classes are all subclasses of the AbstractServices class. Note that during the validation of our approach, a maintainer mentioned another condition to define this part which consists of following the tree structure of sub-packages. For instance, classes that are contained in packages following such a naming structure: x.y.services.z.w. This means that the services part can be identified using a disjunction of these two conditions which can be easily changed in the codebase of the visualization. Finally, the DAO layer contains classes that inherit from the AbstractDAO class.

If these rules enable us to build the visualization, they are either imposed by the use of a framework or by decisions made during the software development phase.

Behavioral Rules. The behavioral rules are responsible for identifying the different layers contained in the parts (client, server) of the visualization. The behavioral rules consist of method calls, attribute access, and class references. All layers in the client part are primarily built from the core of the client (interfaces (a) in Figure 2) and directly or indirectly calling the interfaces. Similarly, layers of the server part classes directly

[4]https://www.gwtproject.org

or indirectly have a behavioral relation with the core of the server (implementations (b)) meaning they are called by these implementations.

Additionally, the purgatory part in Figures 1 and 2 adds entities at each iteration. The visualization tool computes at each level the entities in a behavioral relation with both client and server entities and adds them to the collection of classes in purgatory. The same logic applies to entities of Shared Space (DTOs).

These structural and behavioral rules represent queries that capture the relationships between software components which are customizable depending on each software rules. They serve as flexible mechanisms for examining and assessing the interconnections among software elements.

B. *A language-Independent Metamodel*

The tool is implemented on top of the Moose analysis platform developed in the Pharo language [2]. Therefore the tool is independent of the language used in the analyzed project. For the moment, it was used for Java projects since we had access to the development teams. However, the approach itself is applicable to *all* client-server applications because it merely relies on the structural and behavioral rules inside the codebase for defining the distinct parts/layers of the visualization as elucidated in the previous section.

V. Big-Picture Visualization Configuration in Action

In this section, we present the application of CLISERVO to industrial systems. For confidential reasons, we changed the name of the software systems and cannot mention the name of the companies. Moreover, to respect one company's constraints we blurred the figures. While during our analyses we applied different configurations of CLISERVO, for space reasons, we report the general overview of CLISERVO visualization viewpoint on OJ project and the violation viewpoint on WD project. These projects are presented in the next sections.

A. OJ*: Distribution System of Updates*

OJ is a system of updates distribution, built for a list of clients such as small and big city halls. It was initially developed by a single developer in 2008 replacing an old updates downloading system. It was continuously maintained by the same developer then maintainers changed throughout the years. However, only ten maintainers were responsible for this system throughout the years, with few system evolutions because the system was not exposed to changing needs. It was conceived for one thing (distribution of updates) which is still functioning correctly. The system is decomposed into two projects, the client and the server project. The client project is built using GWT, and the server project is built in Java. The two projects use the RPC protocol automatically generated by the framework to ensure communication between the client and the server.

OJ counts 3277 classes spread in 599 packages. There are 3 packages named client and 2 named server packages. Altogether, the client packages contain 1051 classes. The server packages count 102 classes.

Analysis: Big-Picture in general component overview viewpoint. In Figure 4, we use the general component overview, where all the components are displayed. We added all levels to the visualization, in addition to the core interfaces and implementations respectively. The server part holds only four levels, whereas the client has five.

There are two main dashed boxes each containing at most five dashed smaller boxes. The big left dash box corresponds to the client part. The right one refers to the server. In each of these two boxes, the vertical dashed boxes on the right for the client and the left for the server correspond to the core implementing the communication protocol. The other dashed boxes correspond to the indirection levels successively computed. In the server part, some classes are in red, meaning that they are considered in the server part, but belong to client packages.

The DTOs have not been separated into several groups since they are all used by both the client and the server parts.

The purgatory is empty meaning that all the communications between the client and the server parts are done via the interfaces and the implementations as foreseen by GWT. Consequently, the separation between the two parts is pretty clear and net.

B. WD*: a Multidisciplinary Healthcare Management System*

With WD, doctors can plan meetings, enrol patients in a session, register proposed decisions, and validate and publish decisions. This software system is largely used in hospitals and has been developed around 18 years ago using a client-server architecture. The client part has been developed with GWT, the server part is in full Java, and the communication protocol uses RPC. The system evolved; new functionalities have been added. But also, technology has changed: GWT is no more maintained by Google. New versions of browsers do not support well the Typescript code transpiled from Java. It becomes urgent to migrate the client part to a new technology. In parallel, the company wants to modify the server part. However, after multiple evolutions, the client-server architecture drifted. It is impossible to just remove the client parts and replace them.

WD counts 6030 classes spread in 1044 packages. There are 8 packages named client and 7 named server. Altogether, the client packages contain 3016 classes. The server packages count 389 classes.

Analysis: Big-Picture in violation viewpoint. As for OJ, in Figure 1, we added four levels to the visualization, in addition to the core interfaces and implementations respectively. However, due to the state of the communication between the client and the server parts, we used the violation viewpoint.

On top, there are the DTOs. As explained before, in the violation viewpoint, they are separated into groups between those that are used only by the client (24), those only used

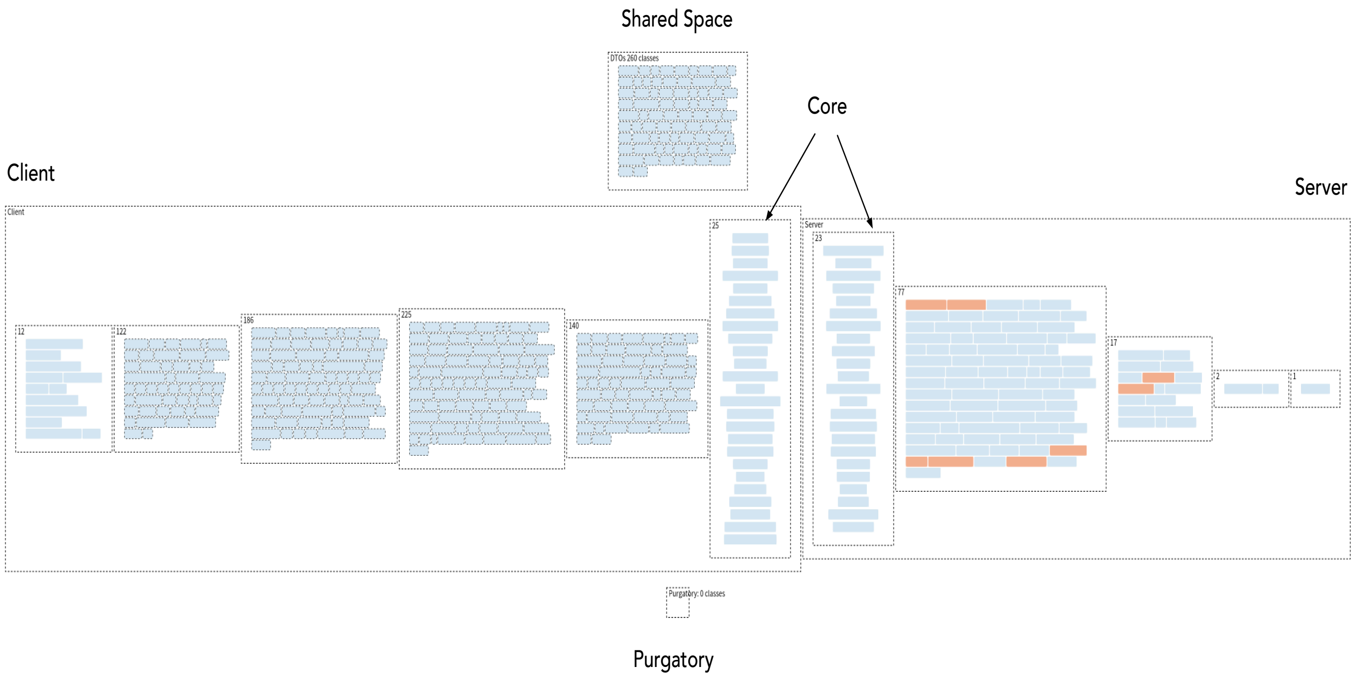

Fig. 4. The Big-Picture of CLISERVO visualization for the OJ software system in Global Component Overview (all nodes of the project)

by the server (82) and those not shared (29). The other DTOs (192) are not displayed in the visualization since, as expected, they are used by elements of the client and the server parts. Only the link from the purgatory at the bottom and the DTOs are displayed in the figure.

At the bottom, there is the purgatory containing the classes whose categorization is ambiguous. At this level of indirection from the core, there are 91 classes. They are used by the server part and they use the client part. Only the links from the purgatory to the client classes or from the server classes are displayed. Since the figure presents the violation viewpoint, all the classes in the client (respectively server) part are targets (respectively source) of such a link. We see that the separation between the client and the server parts is unclear. The communication does not always respect the framework and goes through other classes represented in purgatory. In addition, a lot of classes in the server part belong to client packages.

Note that the violation viewpoint, in this case, enabled the architect to focus on interest points to correct the architecture. When the violation viewpoint shows no more entity, the separation between parts is clear, and adopting the general component overview makes sense for OJ.

VI. Financial System: Server Focus Visualization Applied

EGF is a financial management system for local authorities. Once again, for confidential reasons, the name of the application has been changed and the figure anonymized. This application has been developed in full Java using the RMI protocol. The client and the server parts are physically separated into two different projects. This physical separation has consequences on the software logic: The categorization of each entity is clear, it either belongs to the server or the client part. Consequently, we focus on the server part, containing 4028 packages among which 180 are named dao and 352 service for a total of 11424 classes.

Analysis: Server focus in hybrid viewpoint. Figure 5 shows the Server-Focused visualization on the EGF industrial case.

To better highlight the differences, the visualization is displayed in a hybrid viewpoint: The server part is displayed in the violation viewpoint – this is why the user can see the ratio of displayed classes in each layer. On the opposite, the services and the DAO parts are displayed using the general viewpoint. All the entities of these parts are displayed. Since they are too many, classes are grouped by packages.

When the services bypass the services objects and talk directly to the DAO, these DAO are displayed in dark red, or more precisely, the packages, containing these classes are like so. In addition, to know which Service objects to focus on among all, entities in violation are in orange (those calling a server entity or called by a DAO).

VII. Validation

In this section, we report our analysis of the results of CLISERVO on the industrial projects. This was validated with the current maintainers of each project.

The validation followed the steps: (1) the authors performed the analysis of the three systems using CLISERVO. This phase lasted half a day. The analysis produced a list of potential

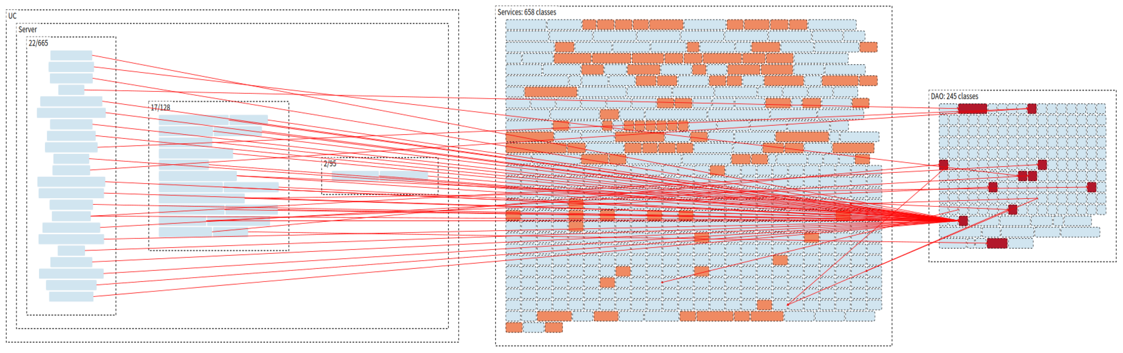

Fig. 5. The server focus of CLISERVO visualization for the EGF software system from the Violation viewpoint (limited to nodes participating to the violations).

violations and points to clarify; (2) all the raised points were discussed and validated with the current maintainers of the systems for half a day each. This phase leads to the opening of bug tickets or the identification of serious and larger problems for some cases (*i.e.,* migration to other front-ends). The subsequent sections summarize our findings.

A. OJ *results*

The results of the OJ project are depicted in Figure 4. The OJ system is an example of a well-conceived project with relatively few classes, few evolutions, and a limited number of different maintainers throughout the years. CLISERVO showed some classes in client packages are referenced from the server part (colored in orange in Figure 4). During the CLISERVO validation, the maintainers explained that these classes are referenced by server entities to avoid code duplication: Instead of creating a new class with the same code, maintainers preferred to refer directly to these classes.

The visualization also showed classes with the DTO keyword in their names appearing on the server part instead of on top in the Shared Space. This means that they do not follow the rules of inheritance on which the Shared Space is built. This was a false positive of the way the classification is done by our tool. Indeed the maintainer explained that such classes do not belong to the OJ system but to a framework developed by the company from GWT components to share common functionalities over projects.

B. WD *results*

The results of the WD project are depicted in Figure 1. The tool helps the maintainers to reclassify the messy package structure where client and server names were mixed in an ad-hoc fashion. In addition, 29 unused DTOs were identified, 24 DTOs only used by the client, and 82 only used by the server. Furthermore, 91 purgatory elements were identified:

- 17 entities using only one client entity and only one server entity,
- 4 entities using only one client entity and used by very few server entities,
- 13 entities using several client entities but only used by one server entity,
- 2 entities using several client entities but very few server entities,
- 4 entities using both client and server entities and being used by very few server entities,
- the rest are entities automatically categorized in purgatory because of indirect relations. For instance, an entity using only clients is used by an entity in purgatory.

Note that the server entities being used are mostly entities categorized in the server part but in client packages depicted in orange in Figure 1. The tool helped current project maintainers identify the idle DTOs and those in violation because they are not actually shared between both client and server parts. Moreover, they were mostly interested in the ambiguous entities in Purgatory since their belonging represents an important problem for them to eventually investigate and classify.

The results of the WD experiment allowed maintainers to reconsider the architecture of their software especially since they were in an architectural transition phase. WD followed a monolithic architecture changed into a client-server architecture. The reasons behind this architectural transition are primarily to make the software system more modular and easy to maintain. However, this transition was not fully successful because of the enormous number of classes (6030 classes) which made it harder for maintainers to correctly reorganize the whole software. The results we presented helped the WD maintainers in detecting entities contributing to the tight coupling of the software (*i.e.,* the entities in purgatory used by both client and server parts). Such cases blur the boundaries between the client and server parts and can lead to architectural confusion, making it harder to reason about and evolve the system. Also, a server part class might not be compatible with the constraints of the client part environment, leading to technical challenges and compatibility issues. Furthermore, sharing a class between the client and server parts may expose sensitive logic or data to the client part. This can potentially create security vulnerabilities if the client is compromised or if the client has access to sensitive information that should be handled securely on the server. Additionally, the visualization allowed maintainers to detect idle DTOs. Such entities are not only misleading entities that increase cognitive load but they also play a part in the low performance of the overall software since they unnecessarily occupy memory resources.

C. EGF *results*

The tool helps the maintainers to spot violations. An exceptional class from the DAO part is accessed by *all* the server implementation classes. One of the maintainers explained that each of their applications has some known and accepted design issues. This violation is one of them. However, CLISERVO also revealed seven other direct links from the implementations to the DAOs layers which are indeed considered as relations not respecting the architecture of their software.

Moreover, the tool showed eight direct links from classes in the first level of the server (classes directly used by the implementations) to the DAOs. One maintainer confirmed that such links do not represent violations because the calling classes found in the server layer are de facto classes that belong to the Services layer, however, organized according to a different structural packaging set (classes of the service subpackaging). Changing the rule of the Services layer to include classes with such a structure helps avoid these false positives. Nonetheless, to our knowledge of the system and communication with its maintainers, the naming conventions are not perfectly respected so adjusting the tool to an inconsistent rule might produce more unintended consequences.

Finally, CLISERVO helped the maintainers identify five confirmed violations coming from the DAOs to the services layer caused by the intertwining entities. The fact that the code was too coupled and spaghetti-like made it impossible for the maintainer to decide what to do. In addition, the visualization helped external experts of the projects and visualization specialists identify the violations which were claimed by the software maintainers.

VIII. THREATS TO VALIDITY

In this section, we discuss the threats to the validity of the experiment.

Internal Validity: To what extent we can draw a causal link between the treatment in the experiment and the response? The study exhibits robust internal validity as it employed a rigorous experimental design involving three different software systems from diverse domains, two different companies, and three project teams. The systems were real industry applications, and access was granted to their current maintainers, who possessed distinct levels of knowledge about the software systems. The study enabled architects to draw insightful conclusions and make modifications to the client-server architecture. Furthermore, it is noteworthy that the software systems are currently undergoing a remodularization phase in preparation for future migration. These factors contribute to the study's internal validity by providing a controlled environment for evaluating the impact of the tool on architectural decision-making within real-world software systems.

External Validity: Are our results generalizable for practice modernization? The CLISERVO visualization validation was applied on three industrial projects since it was not possible to take open-source projects. Indeed, unfortunately, on public repositories like GitHub, it is not easy to introduce search criteria relative to architectural matters. In addition, once found, access to the developers or to the architects of these projects is almost impossible. Consequently, we focused on industrial partners. The variability of the application domains, the sizes, and the development teams encourages us to believe in the generalization of the CLISERVO visualization.

Concerning other object-oriented languages than Java such as Python, C#, or Dart we did not apply our tool because parsers for such languages are not available at the time of writing. Nevertheless, the approach remains applicable to software employing a client-server layered architecture, since the structural and behavioral rules used to build the visualization can be adapted to specific situations.

Construct Validity: Are we measuring what we intend to measure? CLISERVO enabled us to identify both cases where the separation between the client and server is clear respecting the used framework and others where it is not the case. The focus on the server part highlighted an expected layered architecture but also enabled the identification of architectural rule violations between these layers. This leads us to the conclusion that the interest of the visualization is justified.

Reliability: To what extent can the results be reproduced when the research is repeated under the same conditions? One potential reliability threat in this study is the potential for variations in the interpretation of the results by maintainers. To address this threat, efforts were made to provide clear guidelines to maintainers, ensuring a standardized understanding and consistent results. By establishing explicit instructions for interpreting the visualization, potential discrepancies in the findings were minimized, enhancing the reliability of the study. Additionally, the availability of the CLISERVO on GitHub and its use of an importer specifically designed for Java projects contribute to the reproducibility and consistency of the study results across different projects and maintainers.

IX. RELATED WORK

Several approaches recover software architecture from different perspectives [8]. Some tools have been implemented to recover the architecture of the software such as Rigi [33], SNIFF [37], Rose [9], Dali [16], and Software Bookshelf [10]. Schmitt Laser *et al.,* [35] propose the ARCADE tool which is a research workbench for the recovery of software architecture, architectural smells and anti-patterns. It also uses two visualizations Eva [30] and ArcadeViz [24]. Granchelli *et al.,* [15] provide MicroART, a prototypical tool for architecture recovery of microservice-based systems and Deissenboeck *et al.,* [5] worked on the ConQAT tool for architecture conformance assessment capabilities.

Such tools work similarly to our tool in recovering the software architecture. However, CLISERVO is a visualization that takes into account the client-server architecture. To the best of our knowledge it is unique.

When studying the architecture of software some researchers focus on assessing the quality of the software through its architecture. For instance, Fontana *et al.,* [12] propose the *Arcan* tool for the detection of architectural dependencies. Samarthyam *et al.,* [34] motivate the need for refactoring of software code smells that decay the system quality from a high-level perspective by analyzing the impact of the evolution of both the Windows operating system and JDK. They report an elevated complexity and unhealthy dependencies between modules. Such analyses are important in assessing the quality degree of software and measuring the effect of evolution. Indeed, our tool can support the evolution of the software architecture by monitoring and comparing different versions of the same software using different views. Other tools such as the Hotspot Detector [29] detect smells at file and package levels. Lippert *et al.,* [25] also identified architectural smells at various levels: inside inheritance hierarchies, inside and between packages, subsystems, and layers. Maria *et al.,* outlines code smells as indicators of architecture degradation [27]. In this paper, we focus on the detection of the dependencies between classes (packages) from the software architecture layers perspective and entities used by both client and server whose classification is ambiguous. Other work relies on component-based software, Zhang *et al.,* [40] propose the Dedal architecture recovery model for component-based developed software to support design decisions by separating the representations of architecture specifications, configurations, and assemblies. Allier *et al.,* [1] help understand the architecture of a software system by grouping methods in terms of their owner classes to identify the interfaces of service candidates based on the internal structure of components. Although such approaches are per se interesting in software architecture recovery, CLISERVO approach recovers client-server software based on structural and behavioral rules and displays the result as views.

According to Ghanam *et al.,* [14] researchers and architects are more interested in visualizing the high-level design of the software. The work presented in this paper supports such a statement with regard to the use of visualizations in mapping the architecture of software systems. Moreover, the prominent city metaphor used by Wettel *et al.,* [39] and adopted by many other researchers to visualize the architecture of software [18], [32] and as such it was also applied to virtual reality [11]. In their work, Kobayashi *et al.,* [18] use the city metaphor but also consider the software layers and viewpoints conceptually similar to what we do in this paper. However, in this work, we do not use nor extend the city metaphor but use a 2D approach to represent the high-level design of the software. Nonetheless, Balzer *et al.,* [3] introduce the Software Landscapes visualization for the structure of large software systems. Although, the previously mentioned visualizations use metaphors to portray the architecture of the software they do not differentiate between client and server entities of the software.

X. CONCLUSION

Understanding source code and recovering its architecture is a challenging task, particularly for complex systems that have been developed over a long period of time or have undergone numerous changes and modifications. We propose a novel visualization called CLISERVO for recovering the architecture of client-server systems. CLISERVO help software maintainers detect architectural violations. By extracting structural and behavioral information from the software source code, CLISERVO constructs the different parts and layers of the software. Furthermore, the visualization incorporates DTOs present in the codebase, identifying both violations and adherence to architectural rules. Ambiguous entities that raise questions regarding their proper placement are also represented in a dedicated category called “purgatory”. To enhance scalability, ease of focus, and usability, CLISERVO offers two configurations: the Big-Picture view, which encompasses all software components (client, server, DTOs, and purgatory), and the Server-focus view, which focuses solely on the server part.

The application of CLISERVO to industrial projects has yielded highly promising results. For example, in the WD project, the visualization exposed DTO-related violations, including the presence of unused entities and entities exclusively used by either the client or the server. These violations undermine the software’s architectural integrity by introducing unnecessary complexity and consuming additional memory and resources. Moreover, they hinder software comprehension and navigation, especially in large-scale projects like WD, which comprise numerous classes (6030 classes). On the other hand, when applied to the OJ project, which demonstrates a well-conceived and maintained architecture, CLISERVO revealed relatively few violations, primarily false positives stemming from design decisions to avoid code duplication and the inclusion of DTO classes specific to the core software on which OJ is built. This gives insight into the architecture of the software. Additionally, by employing the server-focus configuration, CLISERVO exposed illegal dependencies between layers within the server part of the EGF project, including seven direct links from service implementations to DTOs and five violations from DTOs to the services layer.

The validation with the maintainers shows that, in a couple of hours, the CLISERVO users were able to spot architectural violations and qualify architecture design decisions. It shows also that CLISERVO users were able to conceptualize a coarse-grained quality model of the systems architecture (ranging from 3277 to 6030 classes), clearly identifying situations where future evolutions will be challenging. The visualization not only aids in detecting architectural violations but also facilitates the identification of unnecessary elements, thereby enhancing the maintainability and performance of client-server applications. For future work, we plan to extend the viewpoints and apply the visualization to a larger set of projects.

REFERENCES

[1] Allier, S., Sadou, S., Sahraoui, H., Fleurquin, R.: From object-oriented applications to component-oriented applications via component-oriented

architecture. In: 2011 Ninth Working IEEE/IFIP Conference on Software Architecture. pp. 214–223. IEEE (2011)

[2] Anquetil, N., Etien, A., Houekpetodji, M.H., Verhaeghe, B., Ducasse, S., Toullec, C., Djareddir, F., Sudich, J., Derras, M.: Modular moose: A new generation of software reengineering platform. In: International Conference on Software and Systems Reuse (ICSR'20). No. 12541 in LNCS (Dec 2020)

[3] Balzer, M., Noack, A., Deussen, O., Lewerentz, C.: Software landscapes: Visualizing the structure of large software systems. In: IEEE TCVG (2004)

[4] Clements, P., Kazman, R., Klein, M.: Evaluating software architectures: methods and case studies. 2002

[5] Deissenboeck, F., Heinemann, L., Hummel, B., Juergens, E.: Flexible architecture conformance assessment with conqat. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2. pp. 247–250 (2010)

[6] Demeyer, S., Ducasse, S., Nierstrasz, O.: Object-Oriented Reengineering Patterns. Morgan Kaufmann (2002)

[7] Ducasse, S., Anquetil, N., Bhatti, U., Cavalcante Hora, A., Laval, J., Girba, T.: MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Tech. rep., RMod – INRIA Lille-Nord Europe (2011)

[8] Ducasse, S., Pollet, D.: Software architecture reconstruction: A process-oriented taxonomy. IEEE Transactions on Software Engineering pp. 573–591 (2009)

[9] Egyed, A., Kruchten, P.B.: Rose/architect: a tool to visualize architecture. In: Proc. 32nd Annual Hawaii Conference on Systems Sciences (1999)

[10] Finnigan, P., Holt, R., Kalas, I., Kerr, S., Kontogiannis, K., Mueller, H., Mylopoulos, J., Perelgut, S., Stanley, M., Wong., K.: The software bookshelf. IBM Systems Journal pp. 564–593 (1997)

[11] Fittkau, F., Krause, A., Hasselbring, W.: Exploring software cities in virtual reality. In: Working Conference on Software Visualization. IEEE (2015)

[12] Fontana, F.A., Pigazzini, I., Roveda, R., Tamburri, D., Zanoni, M., Di Nitto, E.: Arcan: A tool for architectural smells detection. In: 2017 IEEE International Conference on Software Architecture Workshops. pp. 282–285. IEEE (2017)

[13] Gallagher, K., Hatch, A., Munro, M.: Software architecture visualization: An evaluation framework and its application. IEEE Transactions on SE pp. 260–270 (2008)

[14] Ghanam, Y., Carpendale, S.: A survey paper on software architecture visualization. University of Calgary, Tech. Rep p. 17 (2008)

[15] Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L., Di Salle, A.: Microart: A software architecture recovery tool for maintaining microservice-based systems. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW). pp. 298–302. IEEE (2017)

[16] Kazman, R., Carriere, S.J.: View extraction and view fusion in architectural understanding. In: Proceedings. Fifth International Conference on Software Reuse. pp. 290–299. IEEE (1998)

[17] Knight, C., Munro, M.: Visualising software-a key research area. In: Proceedings of the IEEE International Conference on Software Maintenance. p. 437 (1999)

[18] Kobayashi, K., Kamimura, M., Yano, K., Kato, K., Matsuo, A.: Sarf map: Visualizing software architecture from feature and layer viewpoints. In: International Conference on Program Comprehension. pp. 43–52. IEEE (2013)

[19] Koschke, R.: Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. Journal of Software Maintenance and Evolution: Research and Practice **15**(2), 87–109 (2003)

[20] Koschke, R.: Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. Journal of Software Maintenance and Evolution: Research and Practice pp. 87–109 (2003)

[21] Koschke, R., Simon, D.: Hierarchical reflexion models. In: Working Conference on Reverse Engineering. p. 36. IEEE Computer Society (2003)

[22] Kruchten, P.B.: The 4+1 view model of architecture. IEEE Software pp. 42–50 (1995)

[23] Langelier, G., Sahraoui, H.A., Poulin, P.: Visualization-based analysis of quality for large-scale software systems. In: ASE '05: Proceedings of the 20th international Conference on Automated software engineering. pp. 214–223. ACM, USA (2005)

[24] Le, D.M.: Architectural evolution and decay in software systems. Ph.D. thesis, University of Southern California (2018)

[25] Lippert, M., Roock, S.: Refactoring in large software projects: performing complex restructurings successfully. John Wiley & Sons (2006)

[26] Lung, C.H., Kalaichelvan, K.: An approach to quantitative software architecture sensitivity analysis. International Journal of SE and Knowledge Engineering pp. 97–114 (2000)

[27] Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N., von Staa, A.: Are automatically-detected code anomalies relevant to architectural modularity? an exploratory analysis of evolving systems. In: Proceedings of the 11th annual international conference on Aspect-oriented Software Development. pp. 167–178 (2012)

[28] Medvidovic, N., Egyed, A., Gruenbacher, P.: Stemming architectural erosion by architectural discovery and recovery. In: Proceedings of the 2nd Second International Workshop from Software Requirements to Architectures (2003)

[29] Mo, R., Cai, Y., Kazman, R., Xiao, L.: Hotspot patterns: The formal definition and automatic detection of architecture smells. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture. pp. 51–60. IEEE (2015)

[30] Nam, D., Lee, Y.K., Medvidovic, N.: Eva: A tool for visualizing software architectural evolution. In: Proceedings of the 40th international conference on software engineering: companion proceeedings. pp. 53–56 (2018)

[31] Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes **17**, 40–52 (1992)

[32] Pfahler, F., Minelli, R., Nagy, C., Lanza, M.: Visualizing evolving software cities. In: Working Conference on Software Visualization. IEEE (2020)

[33] Rigi home page, http://www.rigi.csc.uvic.ca/

[34] Samarthyam, G., Suryanarayana, G., Sharma, T.: Refactoring for software architecture smells. In: Proceedings of the 1st International Workshop on Software Refactoring. pp. 1–4 (2016)

[35] Schmitt Laser, M., Medvidovic, N., Le, D.M., Garcia, J.: Arcade: an extensible workbench for architecture recovery, change, and decay evaluation. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1546–1550 (2020)

[36] Sharafi, Z.: A systematic analysis of software architecture visualization techniques. In: 2011 IEEE 19th International Conference on Program Comprehension. pp. 254–257. IEEE (2011)

[37] TakeFive Software GmbH: SNiFF+ (1996)

[38] Verhaeghe, B., Shatnawi, A., Seriai, A., Anquetil, N., Etien, A., Ducasse, S., Derras, M.: Migrating GUI behavior: from GWT to Angular. In: International Conference on Software Maintenance and Evolution. Luxembourg (2021)

[39] Wettel, R., Lanza, M.: Visualizing software systems as cities. In: Working Conference on Software Visualization. pp. 92–99. IEEE (2007)

[40] Zhang, H., Urtado, C., Vauttier, S.: Architecture-centric component-based development needs a three-level adl. In: Software Architecture: 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Proceedings 4. pp. 295–310. Springer (2010)