Human Interaction and Emerging Technologies (IHIET 2022), Vol. 68, 2022, 502–509
https://doi.org/10.54941/ahfe1002770

Rapid Interactive Software-Architecture Design with Split-n-Join Actions

Anthony Savidis[1,2] and Anthony Peris[2]

[1]Institute of Computer Science, FORTH Heraklion, Crete, Greece
[2]Computer Science Department, University of Crete, Heraklion, Crete, Greece

ABSTRACT

The software architecture design process is an essential part of the software development lifecycle. During the early phases, architects combine creative thinking, cooperative sessions, exploratory analysis, quick sketching and abstract design aiming to craft an optimal initial architecture. During this startup stage, tools requiring exhaustive information, detailed modelling and elaborate specifications can be tedious and impractical, even frustrating. Also, due to evolution, continuous refinement, syncing and maintenance is required, something that should be handled easily, flexibly and abstractly. In this context, we present an architecture design tool for rapidly and easily composing and reshaping architecture diagrams. It is based on components and abstract operations, focusing on quick refinement and exploratory crafting, while playing repeatedly with split-n-join actions on components.

Keywords: Rapid software-architecture design, Software architecture prototyping, Interactive software-architecture design environments

INTRODUCTION

Today, in many architecture design tools, commonly relying on UML diagrams, the overall process is highly detailed, resulting in time consuming activities, asking designers to elaborate very early on technical aspects that are usually finalized much latter in the development lifecycle. Effectively, such tools are meant to be rapid interactive prototyping laboratories, but serve mostly as architecture documentation environments. However, because they require so fine-grained detail, which is transient, volatile and non-final in the early design phases, they are less preferred for initial experimentation and analysis. Effectively, it is impractical for architects to spend the required effort in supplying data for components, specifications and relationships when those frequently change after the early design phase.

In fact, it is acknowledged that software architectures are "intellectually graspable" abstractions of complex systems (Bass et al. 2012), with primary emphasis on concepts such as "components, connectors, and styles" (Shaw et al. 1996). Based on these remarks, our work focuses on supporting the very early stages of the architecture design process, putting primary emphasis on rapid interactive construction, ease-of-use, continuous experimentation, minimal information, and adoption of common architectural abstractions.

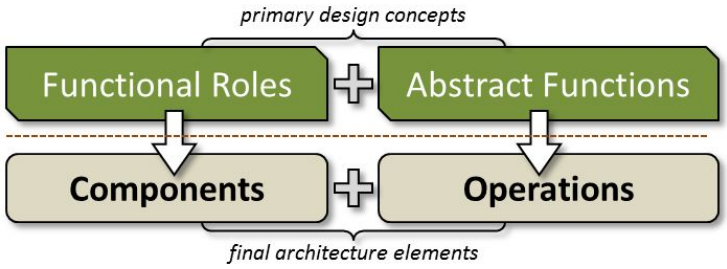

Figure 1: The emphasis on functional roles and abstract functions as the primary design concepts, leading to eventually derived architecture elements.

Basic Notions

Our process reflects two key notions (Figure 1): (i) *functional role*, mapping to *components* and driving the grouping of common operational features relating to a common and shared role-based responsibility; and (ii) *abstract functions*, relating to how the envisioned features may map to concrete operations of the new system.

Focusing primarily on components and operations, our tool reflects the exploratory nature of the design process by offering two key actions, namely *splitting* and *joining* components, besides typical creation and removal. This work is inspired by the quick object-oriented design method of CRC Cards (Beck & Cunningham, 1989) (Classes, Responsibilities and Collaborators), part of agile development, by adapting the original notions to fit with the scale and abstractions of the software architecture domain as Components, Operations and Connectors (COC). In our tool, the primary requirement has been the facilitation of rapid exploratory interactive design, with small effort on behalf of the user, making it a laboratory for testing where related ideas may be easily instantiated via the tool. Considering that the architecture structure changes frequently in this process, we identified most common actions architects perform when revisiting component roles, besides component insertion and removal:

- ***Split***: applied when a grouped operations, either belong to a component, or standalone, are identified to combine many different disciplines altogether that separately deserve representation (i.e. decomposition) as distinct items;
- ***Join***: performed when a few components are considered as weak or arbitrary to stand on the own, while in terms of their functional role they look as pieces of the same concept, likely requiring merging together under the same umbrella;
- ***Connect (Disconnect)*:** links components together with tagged *connectors* reflecting well-defined operational synergy between them.

Then, we treat operations as first-class elements of the design process, enabling to associate them directly to components, while freely moving them across components, or setting them as orphan (standalone) via the following activities:

- ***Set***: attaches an orphan operation to a component;
- ***Move***: reassigns an operation from one component to another;
- ***Reset***: detaches an operation from a component and sets it as orphan.

We discuss how such simple activities are fundamental and *capture the essential aspects in early architecture design* tasks, and show examples on the way we supported them interactively, keeping their delivery simple, quick and yet sufficient. For instance, component associations or synergies may change by simply rearranging links with the mouse, while operations are managed easily by typical drag-n-drop. Additionally, further component decomposition is supported, for sub-architecture analysis, enabling craft quickly more detailed processing structures, while their view may be toggled with just a click.

RELATED WORK

In (Perry & Wolf, 1992), the distinction between three different classes of architectural elements is made, namely, *components*, *data elements*, and *connecting elements*. This original view is very abstract and elegant, focusing on the conceptual representation, rather than on exhaustive and rigorous specifications. Latter, the explicit representation of data was abandoned since it could be better modelled as a data-related component. The notion of conceptual integrity is introduced in (Brooks, 1975) to emphasize that software architectures are visions of how systems behave, clearly separated from their implementations. In this context, architects are essentially "keepers of the vision", making sure that such an architecture vision is preserved and respected throughout the entire development lifecycle. Existing architecture design environments generally fall in two categories: (a) detailed specification tools, requiring the exhaustive modeling of many functional aspects of a system, sometimes leading to automatic generation of implementation modules; and (b) rapid design tools, with emphasis on graphic and visual prototyping.

The first category includes model-driven architecture (Kleppe, 2003), an approach that pushes architects in learning custom notations, sometimes being very close to programming notations, like Executable UML (Raistrick et al., 2004). Design tools like StarUML (https://staruml.io/) and Astash (https://astah.net/) focus on UML and involve specific implementation aspects like class diagrams much earlier than what needed. In the second category, tools like Archi (https://www.archimatetool.com/) and Graphor (https://gaphor.org/) emphasize graphical design with a toolset of visual architecture elements, but still involve quite detailed representations and symbolisms, emphasizing the early categorization of architecture elements.

Our critique in such tools is that, many times, the information required is far more detailed, while the supported actions, even when the emphasis is on rapid sketching, do not reflect a common process model, but seem almost like a graphical editor. For instance, in our case, component *merging or joining* is a semantic action applying on the components but also on their associated operations. Similarly, the notion of *operations*, as general or abstract functions, initially disengaged of components or functional containers, is not met

in other tools, although their notion appears during requirements elicitation and functional specifications, where planned system features give birth to a catalogue of general system operations.

DESIGN PROCESS

Our design process aims to be *rapid by design*, involving no particular custom symbolisms for components, just plain boxes, leaving designers to represent semantics following component roles, and optionally within related brief descriptions. In fact, we consider that many mission-specific symbolisms for components, implying functional responsibilities, are unnecessary, since they will disappear in the implementation process and the resulting source code. For example, assume a component role is *handling all user input*. Then, in many design tools, a graphical element denoting input must be used. In our case we direct architects embody all semantic information in the careful role description, given if possibly through a component name, thus something like "*user input handler*" would do the job. Also, we emphasize that architects communicate such semantic aspects explicitly to the implementation process, requesting that such role names are adopted so that the immediate connection to the architecture remains clear. In Figure 2 we provide an example outlining a few steps from the architecture design process of a classic platformer game (Super Mario). As outlined, all design actions are applied directly on either components or operations, while users are enabled to reconsider earlier decisions via changes easily and frequently. Saving the current state is supported, besides free undo / redo of every design activity.

The detailed semantic control flow of an architecture design process in our tool is depicted with two flowcharts, under Figure 3 and Figure 4, regarding component and operation management respectively. Effectively, architecture design is completed when the following condition is met: ***no further action on components or operations is necessary and every operation is attached to some component***. This is a very simple termination criterion and emphasizes the simplicity and abstraction of the architecture design process in general, whose main purpose should be to grasp, outline, integrate and correlate all the involved functional concepts in an expressively simple and understandable manner. Anything beyond that entails the danger of bringing too-many implementation details too-early in the design process. Also, this blending of abstract elements and relationships allows changes and refinements, during the implementation, to be conveniently applied. Clearly, in the implementation phase, new ideas for additional features will put on the table, and overall a better understanding of the system under development will be instantiated. This may lead to necessary corrections on the original architectural picture, something that is straightforward and minimal if the design tool expresses only the abstract elements and avoids lower-level details.

COMPONENTS AND OPERATIONS

During the interactive architecture design process the primary manipulated elements are components. However, the existence of such components is

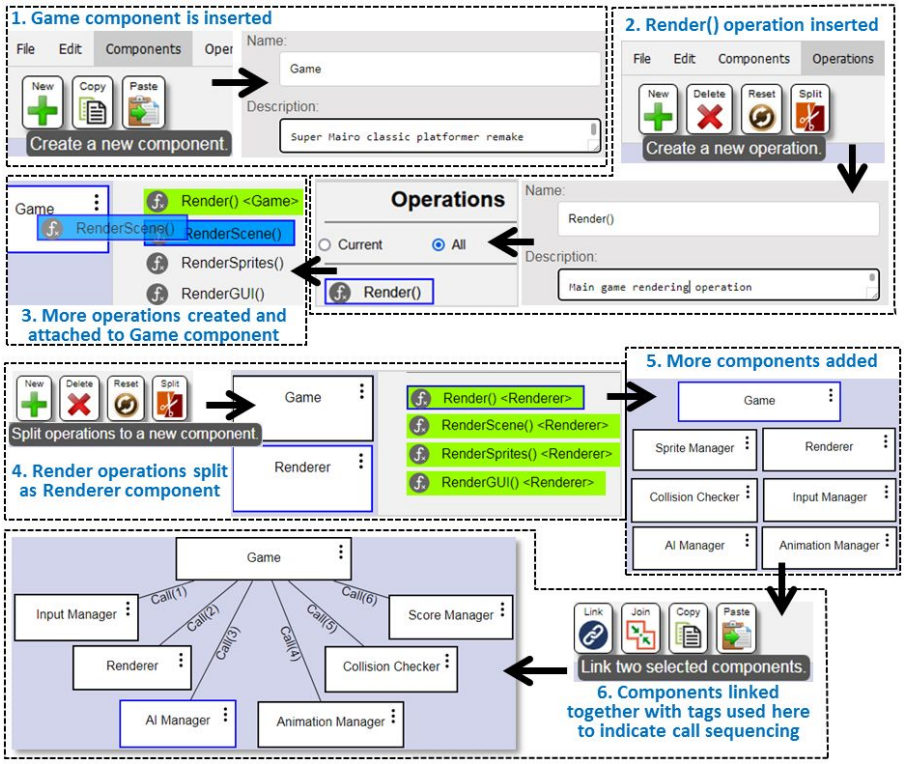

Figure 2: Overview of the tool facilities involved within a brief process for quick architecture creation and refinement; steps are numbered, arrows also indicate sequencing.

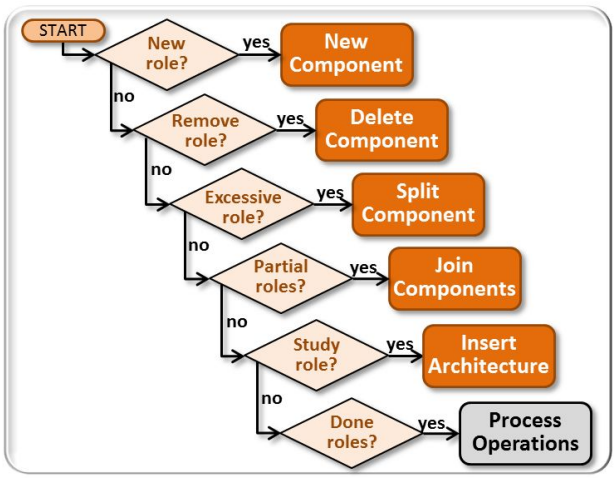

Figure 3: Components design flowchart - 'insert architecture' concerns the detailed design of a role's sub-architecture, implying another layer of architecture component analysis.

only justified by their responsibility to deliver a well-defined set of necessary system operations, the latter logically grouped altogether under the functional role of their host component. This interlinking is denoted under Figure 5

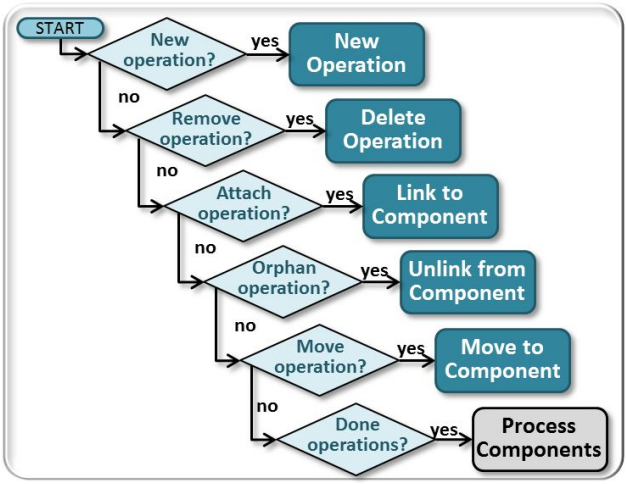

Figure 4: Operations design flowchart – operations are added in a separate catalogue as part of functional requirements analysis and initially may not be associated to components.

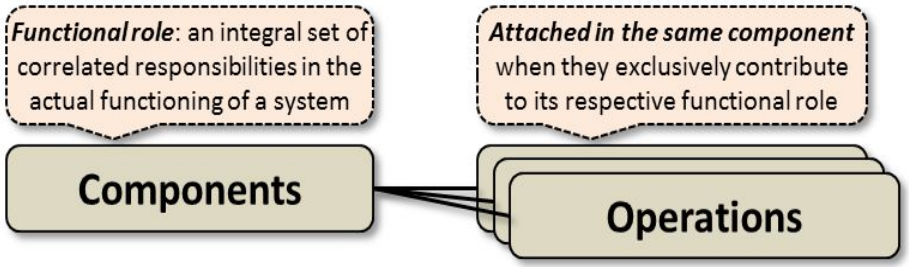

Figure 5: Bridging components with operations in our architecture design process.

showing that the notion of *role* is fundamental, around which the driving of our architecture design process is done.

An example showing the quick interlay of component and operation editing in our tool is outlined under Figure 6, in the context of web application architectures. In particular, a few general components are firstly introduced, and the *application logic* component is further analyzed via the subdivision option. This results in identifying a few sub-components to handle *Statistics*, *Users*, *Data Forms* and the *Q&A Catalogue*, followed by a preliminary analysis of some specific operations relating to the *Statistics* component, that are directly assigned to it.

Besides starting the architecture design process always from components, as it is common in most design tools, sometimes the requirements analysis phase identifies primary operations that can also serve as the starting point of architecture analysis. Now, this implies that the initial design activities should be operation-driven, and as such should allow handle them without even committing to a single component. This is a feature supported in our tool, enabling to start an operation-centric activity, and then forward in a stepwise fashion to group them into components via the *split* operation. An example is depicted under Figure 7, where, as part of an IDE (Integrated Development

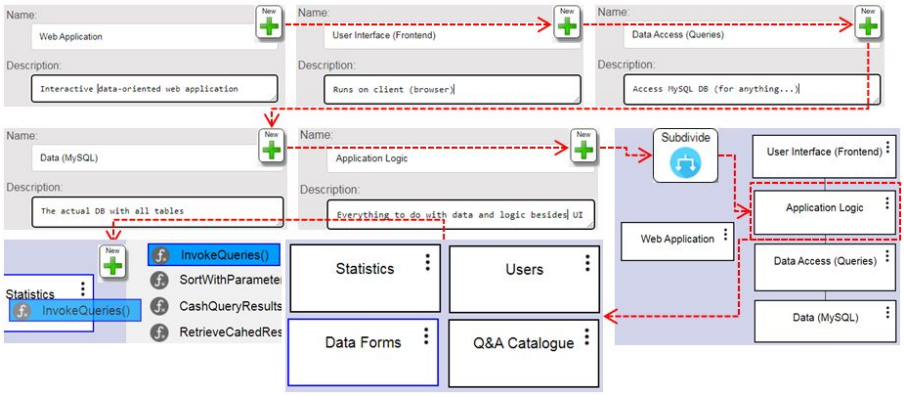

Figure 6: Crafting web-application architecture and sub-dividing further the internal architecture of the application logic component.

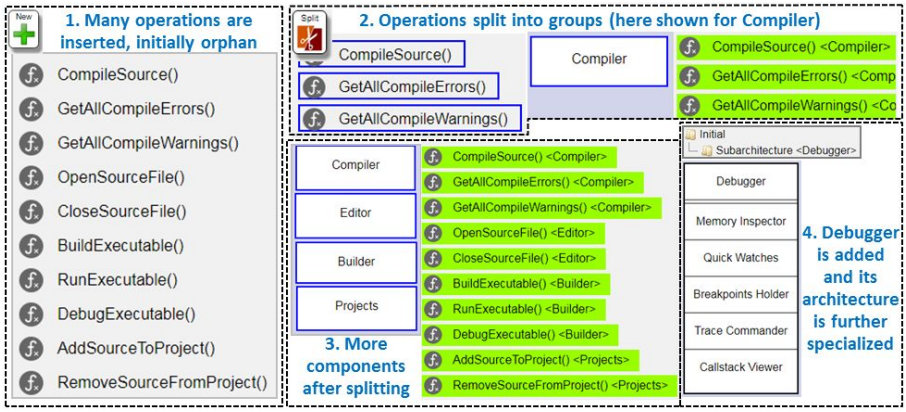

Figure 7: Initiating a design process based on key operations and then incrementally deriving detailed components by splitting the operations into separate groups.

Architecture), a free list of operation emerges, subsequently organized via splitting into components. Following, the Debugger component emerges and its sub-architecture is further analyzed via the subdivision feature. As shown at the bottom right of Figure 7, its internal components are also displayed; this is interactively configurable per component, meaning viewing the sub-architecture (if any) of components can be turned on or off.

CONCLUSION

Software architecture design is a critical and very demanding task that is carried out at the early phases of the software development lifecycle. Tool support for this process is known to be offered for as long as directives, guidelines and patterns exist, while emphasis is mainly put on precision, detail, elaboration and coverage of every single functional aspect that engineers may foresee in an analytic fashion. Unfortunately, most tools invested on providing numerous features and parametrization possibilities, turning architecture design to an exhaustive and even daunting task, introducing unnecessary forward links to implementation details.

However, the essence of architecture design is abstraction and conceptualization, or the less-is-more principle, enabling to dismiss all details that lay outside the high-level semantics and the vision of an abstract operational structure. Along these lines, we focused primarily on the task itself, supporting only those activities that commonly recur in the context of an exploratory architecture design process. We dropped any features related to implementation notations, and all visual symbolisms that could be simply represented with appropriate component or operation naming. Although our system is still in a prototype version, we quickly observed how rapid it is for software architects to capture an initial architecture and incrementally apply refinements as part of the fundamentally fluid design process.

We believe that more focus is needed on developing instruments exploiting the creative, exploratory and playful nature of the overall architecture design process, by offering facilities that involve small effort and little commitment in interactive design, so that anything can be conveniently and with minimal overhead altered, until eventually the design itself is considered as optimal and complete.

REFERENCES

Bass, L., Clements, P., Kazman, R. (2012). Software Architecture in Practice, Third Edition. Boston: Addison-Wesley.

Beck, K, Cunningham, W. (1989). A laboratory for teaching object oriented thinking. In Proceedings of OOPSLA '89 Conference proceedings on Object-oriented programming systems, languages and applications, ACM, pp 1–6.

Brooks, F. (1975). The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley.

Kleppe, A. (2003). MDA Explained, The Model Driven Architecture: Practice and Promise. Addison-Wesley

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. Communications of the ACM, Volume 15, Issue 12 (Dec. 1972), pp. 1053–1058.

Perry, D., Wolf, A. (1992). Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes. Volume 17, Issue 4 (Oct. 1992), pp 40–52

Raistrick, C., Francis, P., Wright, J., Carter, C., Wilkie, I. (2004). Model Driven Architecture with Executable UML. Cambridge University Press.

Shaw, M., Garlan, D. (1996). Software architecture - perspectives on an emerging discipline. Prentice Hall.

Soni, D., Nord, R., Hofmeister, C. (1995). Software architecture in industrial applications. In ICSE '95, proceedings of the 17th international conference on Software engineering (April 1995), pp 196–207.