Architectural Refactoring – a Task-Centric View on Software Evolution

Author: Olaf Zimmermann, ozimmerm@hsr.ch

This is the Author's Copy of DOI: http://doi.ieeecomputersociety.org/10.1109/MS.2015.37

1. Introduction

Software-intensive systems often have to be reengineered, e.g. due to unpredictable business context changes and technology innovations. Many reengineering activities affect the software architecture of these systems. Given the success of the agile practice of code refactoring, it is rather surprising that architectural refactoring has not taken off yet – a first patterns-based catalog of architectural refactorings was presented in 2007 [1]. In this article I look at architectural refactoring from another angle. I first position architectural refactoring as an evolution technique that revisits architectural decisions made. I then present an example, deduce a task-centric architectural refactoring template, and outline a catalog of common architectural refactorings. I conclude with a discussion of potential impact and tool support.

2. Introducing Architectural Refactoring

The goal of a refactoring is to improve a certain quality while preserving others. For instance, code refactoring is a technique for restructuring code to make it more maintainable without changing its observable behavior [2]. Code refactorings work on machine-readable entities such as packages, classes and methods; hence, they can leverage data structures from compiler construction such as abstract syntax trees. Architectural refactorings deal with architecture documentation and the manifestation of the architecture in the code and runtime artefacts. Hence, a single architectural syntax tree does not exist – architectural refactorings pertain to:

- components and connectors (modelled, sketched, or represented implicitly in code)
- design decision logs (which come as structured or unstructured text)
- planning artefacts such as work items in project management tools.

An *architectural smell* is a suspicion (or indicator) that something in the architecture is no longer adequate under the current requirements and constraints, which may differ from the originally specified ones. An *Architectural Refactoring (AR)* then is a coordinated set of deliberate architectural activities that removes a particular architectural smell and improves at least one quality attribute without changing the scope and functionality of the system. An AR can possibly have a negative influence on other quality attributes, due to conflicting requirements and trade-offs.

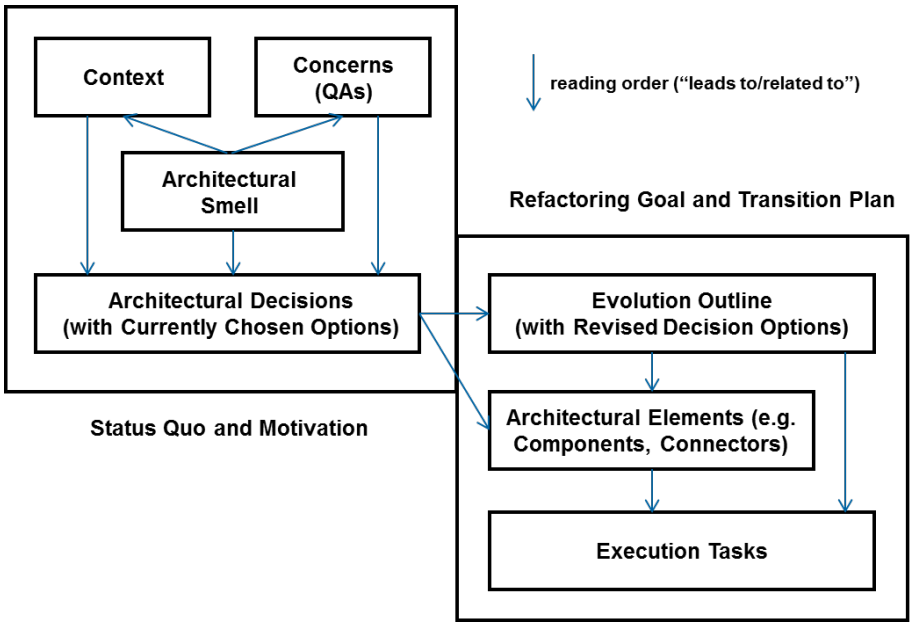

Figure 1 – The Anatomy of an Architectural Refactoring

In my view, an architectural refactoring revisits certain architectural decisions [3] and selects alternate solutions to a given set of design problems. Decision execution leads to related engineering tasks; hence, the revision of a group of architectural decisions causes reengineering tasks. These tasks can be grouped in categories:

- tasks to realize structural changes in a design; such changes have a larger scope than code refactorings and deal with components, subsystems and systems of systems (and their interfaces),
- implementation and configuration tasks in development and operations (depending on the viewpoint the architectural refactoring pertains to),
- documentation and communication tasks, e.g. modelling activity, technical writing assignment, or design workshop preparation and facilitation.

My view on ARs complements Michael Stal's one. He used a pattern format to document his ARs, which include *Breaking Dependency Cycles* and *Splitting Subsystems*, addressing architectural smells such as *Unclear Roles of Entities* and *Dependency Cycles* [4].

3. Full Example and Task-Centric Template

In their technology blog, the chief technicians at Doodle explain why they switched from MySQL to MongoDB after several years of production use of their collaborative online calendar scheduling service [5].

The architectural smell in this example was that it took too long to migrate large production databases after an SQL schema change (such as the adding a column to a table). The affected quality attributes were productivity of the development and operations teams, as well as performance and scalability of database and data access layer. The root cause for the symptoms indicated by the smell was that relational database management systems are not designed for this usage scenario – they can handle it, but not optimally. The solution was to revisit the architectural decisions regarding database paradigm, query APIs and database provider. A decision was made to use the document-oriented paradigm, one flavour of schemaless NoSQL, and MongoDB as document database provider. Migration management

was improved at the expense of administration and coding effort – new solutions for data access, transaction and backup management were required.

The Doodle example clearly qualifies as an AR: it revisits certain architectural decisions to improve a quality attribute, but is not a code refactoring. The following structured AR representation makes it easy to comprehend (and apply in a similar project context):

Architectural Refactoring Name How can the AR be recognized and referenced easily?	***De-SQL***
Context Where (and under which circumstances) is this AR eligible?	Logical viewpoint and deployment viewpoint, both conceptual level (database paradigm) and asset level (MySQL vs. MongoDB) of abstraction
Stakeholder concerns (including quality attributes and design forces) Which non-functional requirements and constraints are impacted by this AR?	Flexibility (w.r.t. data model changes), data integrity, migration time
Architectural smell When and why should this AR be considered?	It takes rather long to migrate existing database content when data model (database schema) is updated
Architectural decision(s) to be revisited Which design problems pertain to this AR, and which design options are currently chosen to resolve them?	• Choice of data modeling paradigm (current decision is: relational) • Choice of metamodel and query language (current decision is SQL) • Choice of database management system (current decision is MySQL)
Evolution outline (solution sketch) Which design options should be chosen now? How does the target solution look like?	• Use document-oriented database such as MongoDB instead of relational database such as MySQL • Redesign transaction management and database administration
Affected architectural elements Which design model elements have to be changed, e. g., components and connectors (if modelled explicitly)?	Database tier (e.g. server process, backup and restore facilities); data access layer (e.g. patterns for commands and queries, connection pools)
Execution tasks How can the AR be applied and validated?	• Design document layout (i.e., the pendant to the machine-readable SQL DDL) • Write new data access layer, implement SQLish query capabilities within application • Decide on transaction boundaries (if any) • Document database administration changes (e.g., command-line queries and update scripts, backup procedures) • Compare old and new solution according to success criteria (e.g. migration time, performance of data access layer)

This example also proposes an AR documentation template. Each row in the above table contributes one template entry. The resulting template structure is illustrated in Figure1.

The AR name should be expressive, e.g. metaphor. Unlike pattern names (which typically are nouns), AR names should be verbs (just like names of code refactorings). The context section may include information about the abstraction level in a software engineering method or an enterprise architecture management framework. Since the AR describes a design change, two solution sketches may be provided, one illustrating the design before the AR is

applied, and one the design resulting from the application of the AR. Architectural elements form a link to the structural design, which might be modelled explicitly, sketched informally or represented implicitly in code. Some of the execution tasks can possibly be automated (just like the execution of many code refactorings), but not all of them (as ARs operate on a higher level of abstraction and a larger scale). The task description may refer to work item types in agile planning tools or to activities in software engineering methods.

4. An Architectural Refactoring Catalog

Let's now go broad and cover additional ARs in four viewpoints. The table shows basic ARs in two dimensions, architectural viewpoints and type of change:

Viewpoint	*Elaboration ARs*	*Adjustment ARs*	*Simplification ARs*
Logical Viewpoint (VP)	Split Component Responsibility	Expose Internal Feature as Component Responsibility	Merge Component Responsibilities
	Shift Responsibility to New Component	Shift Responsibility to Existing Component	Merge Components
	Split Layer (a.k.a. Move Components to New Layer)	Replace Layer	Join Adjacent Layers (a.k.a. Collapse Layers)
Process VP	Distribute Processing (Introduce Concurrency)	Change Distribution Algorithm (e.g. from Round Robin to Priority-Driven)	Consolidate Processing (Remove Concurency)
	Introduce Cache	Change Cache Entry Lookup Key (Calculation)	Remove Cache
	Prepopulate Cache (Load More Eagerly)	Change Cache Cleanup Strategy	Start with Empty Cache (Load Lazier)
Deployment VP	Assign Logical Component to New Deployment Unit	Change Scaling Strategy (e.g. from vertical scale up to horizontal scale out)	Merge Deployment Units
	Split Deployment Unit	Move Deployment Unit (from one server node to another)	Consolidate Nodes
Physical VP (Operational Model)	Factor Out Node into New Tier	Split Tier	Collapse Tiers
	Introduce Clustering	Change Load Balancing and Failover Policy	Remove Clustering

All of these ARs can be represented as instances of the task-centric template from above; e.g. the tasks for Introduce Cache include deciding on a lookup key and invalidation strategy, cache distribution, etc.

5. Final Thoughts

While code refactoring is a mainstream practice today, architectural refactoring has not been studied much yet. In this article, I took a task-centric view here and introduced an architectural refactoring template by example; it collects the architectural decisions to be revisited and the design, development, and documentation tasks to be conducted when an architectural refactoring is applied. I also outlined a catalog of general-purpose architectural refactorings.

In the future, domain- and style-specific AR catalogs might appear, e.g. for financial services software, game development, or cloud computing. Three candidate ARs for a prospective architectural refactoring catalog for enterprise application modernization are:

- Move session state management (e.g, from client or mid-tier server to database to improve horizontal scaling and to better leverage cloud elasticity).
- Replace scalar parameters with data transfer object in service interface contract (to reduce number of remote calls).
- Streamline Web client (to reduce client workload and processing capabilities).

ARs provide an opportunity for cross-community collaboration, for instance between:

- Architecture and development: AR execution may involve one or more code refactorings, which have to be stitched together,
- Architecture and project management: AR descriptions that are organized according to the architectural refactoring template can be used as planning tasks, and the need for architectural refactoring is an expression of technical debt.
- Architecture and operations ("ArchOps"): ARs in the deployment viewpoint can serve as a communication means here.

It remains to be seen how ARs can be shared and executed most efficiently – are templates and catalogs good enough as knowledge carriers? Or are modelling and collaboration tools more appropriate? A Web-based delivery of knowledge has a natural appeal (as e.g. Wikipedia shows). Code refactoring started with a book and formal groundwork; refactoring tools e.g. in Eclipse were developed much later after content and theory had been established and experience had been gained. Any AR tool support would need to tie in with modelling tools supporting UML or architecture description languages. Such tool support yet has to emerge.

References

[1] M. Stal, Architecture Refactoring blog post, OOP and OOPSLA tutorials, http://stal.blogspot.ch/2007/01/architecture-refactoring.html
[2] M. Fowler, http://martinfowler.com/bliki/DefinitionOfRefactoring.html
[3] O. Zimmermann, Architectural Decisions as Reusable Design Assets. IEEE Software, vol. 28, no. 1, pp. 64-69, Jan./Feb. 2011, doi:10.1109/MS.2011.3
[4] M. Stal, Refactoring Software Architectures, in: A. Babar, A: W: Brown, I. Mistrik (Eds.), Agile Software Architecture, Morgan Kaufman, 2014.
[5] Doodle Blog, Doodle's Technology Landscape, http://en.blog.doodle.com/2011/04/14/doodles-technology-landscape/ and http://en.blog.doodle.com/2013/11/18/doodles-technology-landscape-2