ARCHITECTURAL REFACTORING FOR THE CLOUD: A DECISION-CENTRIC VIEW ON CLOUD MIGRATION

9th Symposium and Summer School On Service-Oriented Computing

Prof. Dr. Olaf Zimmermann

Distinguished (Chief/Lead) IT Architect, The Open Group

Institute für Software, HSR FHO

Hersonissos, June 30, 2015

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Objectives (Research Projects and this Presentation)

1. **Share Architectural Knowledge (AK)**
   - *Here:* Identify baseline/principles for Cloud Application Development (CAD)

2. **Advance state of the art in AK Management (AKM)**
   - *Here:* Establish Architectural Refactoring (AR) as a novel software evolution and reengineering practice – and apply it to CAD

3. **Help bridge the gap between agile practices and software architecture along the way.**
   - *Here:* Use user stories, introduce quality stories, propose interface to agile task management

© Olaf Zimmermann, 2015.

Abstract

Unlike code refactoring of programs, architectural refactoring of systems is not commonly practiced yet. However, legacy systems typically have to be refactored when migrating them to the cloud; otherwise, these systems may run in the cloud, but cannot fully benefit from cloud properties such as elasticity. One reason for the lack of adoption of architectural refactoring is that many of the involved artefacts are intangible – architectural refactoring therefore is harder to grasp than code refactoring.

To overcome this inhibitor, we take a task-centric view on the subject and introduce an architectural refactoring template that highlights the architectural decisions to be revisited when refactoring application architectures for the cloud; in our approach, architectural smells are derived from quality stories. We also present a number of common architectural refactorings and evaluate existing patterns regarding their cloud affinity.

The final contribution of this paper is the identification of an initial catalog of architectural refactorings for cloud application design. This refactoring catalog was compiled from the cloud patterns literature as well as project experiences. Cloud knowledge and supporting templates have been validated via action research and in cooperation with industry practitioners.

© Olaf Zimmermann, 2015.

Agenda

- Motivation
- Cloud Computing Fundamentals
- IDEAL Cloud Applications and Cloud Computing Patterns

- Architectural Refactoring vs. Code Refactoring
- Quality Attribute Stories and Architectural Refactoring Templates

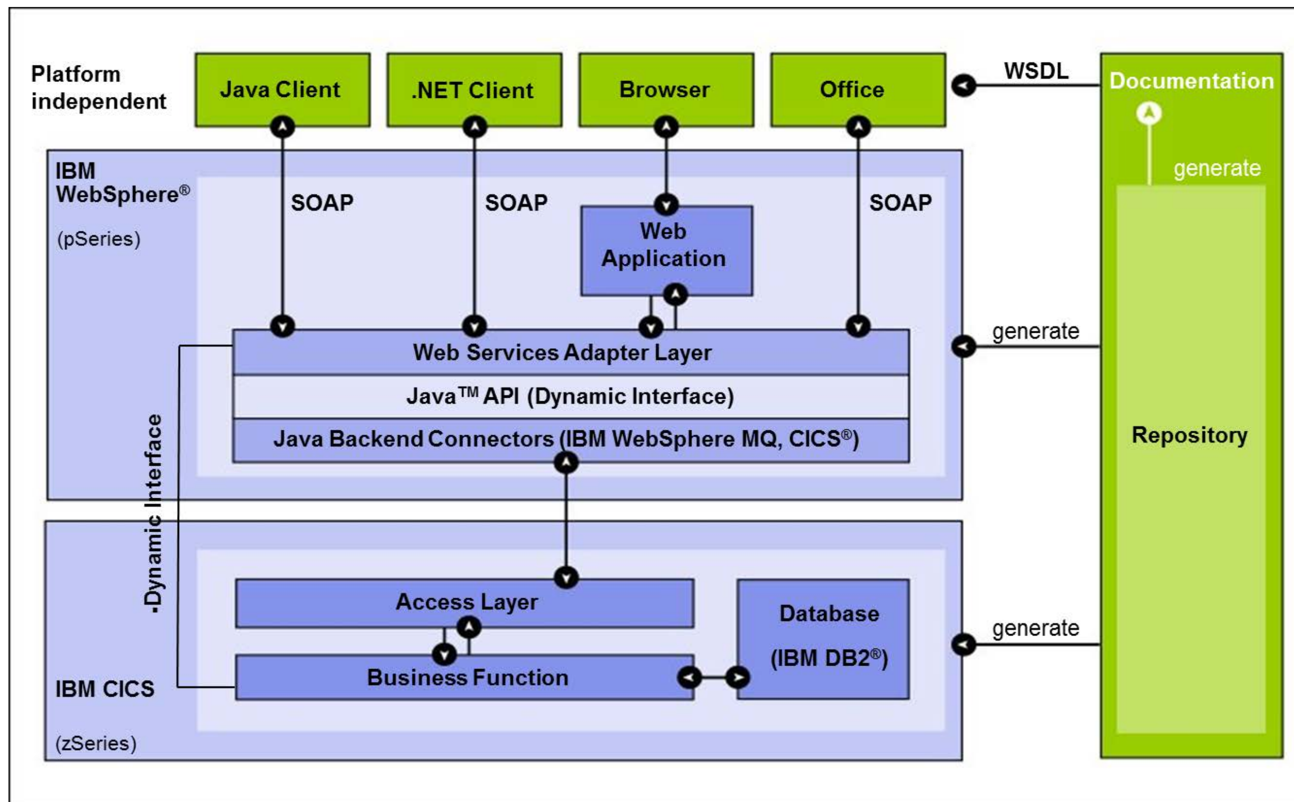
- Architectural Refactoring for the Cloud

- Tool Support for Architectural Knowledge Management (AKM)

© Olaf Zimmermann, 2015.

Motivation: Typical Service-Oriented System (Pre-Cloud Age)

- **Core banking application, shared service/service provider model**
  - Layers pattern, data stored in backend, Web frontend, Web services



Reference: ACM OOPSLA 2004 & Informatik-Spektrum Heft 2/2004

© Olaf Zimmermann, 2013.

Simple, User-Centered Definition of Cloud Computing

Cloud computing provides a set of computing resources with the following testable characteristics:

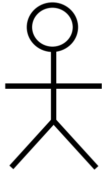
1. *On-demand*: the server is already setup and ready to be deployed (so the user can sign-up for the service without waiting)
2. *Self-service*: customer chooses what they want, when they want it (the user can use the service anytime, without waiting)
3. *Scalable*: customer can choose how much they want and ramp up if necessary (the user can scale-up the service when needed, without waiting for the provider to add more capacity)
4. *Measurable*: there's metering/reporting so you know you are getting what you pay for (the user can access measurable data to determine the status of the service)

In summary, cloud computing is OSSM (pronounced 'awesome').

Reference: B. Kepes, CloudU (online training, sponsored by RackSpace), Dave Nielsen, Cloud Camps, http://www.daveslist.com

© Olaf Zimmermann, 2015.

A Cloud User Story*: Cloud-Native Application Development



Lean Startup Developer

As a **developer and owner of a novel Web application who is unsure about user reception and business value of this software,**

I would like to **rapidly deploy my application into production without having to invest into hardware, data center space and operations staff (and be able to scale it up on demand)**

so that **I can get user feedback to improve my software and the business model – without investing too many resources and taking unnecessary financial risk.**

To do so, **I need to know the characteristics of cloud-native application architectures.**

*see agile practices guide at http://guide.agilealliance.org/

© Olaf Zimmermann, 2015.

Cloud Computing Patterns (CCP)

billing:
traffic in / out
processed messages
& transactions etc.

deploy / remove
application
components

monitoring:
traffic /
requests
etc.

buy

Self-Service Interface / API

Metering
&
Billing

0 0 9 1

$

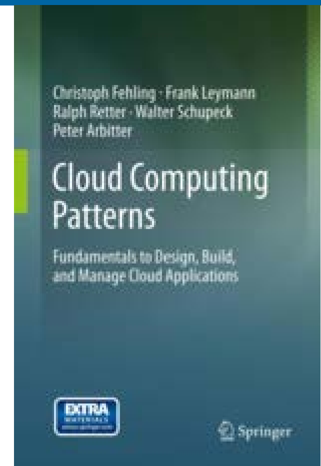
Component Management

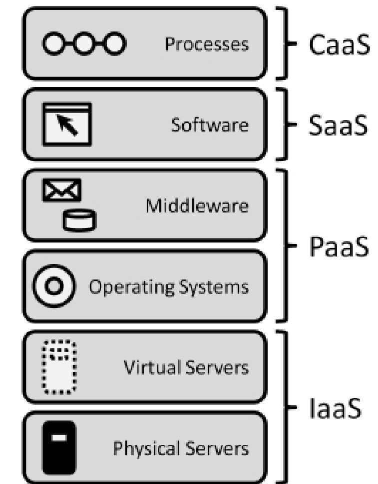
Monitoring

Component
Storage

Components

Execution
Environment



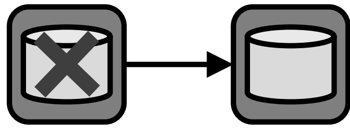


Reference: Cloud Computing Patterns, Springer 2014, http://cloudcomputingpatterns.org/

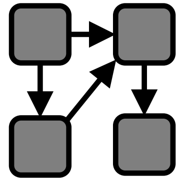
© Olaf Zimmermann, 2015.

IDEAL Cloud Application Properties (Fehling, Leymann et al.)

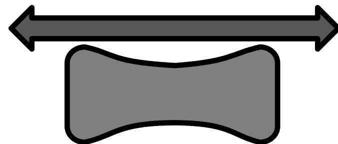
Reference: Cloud Computing Patterns, Springer 2014, http://cloudcomputingpatterns.org/



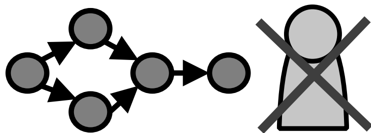
Isolated State: most of the application is *stateless* with respect to:
Session State: state of the communication with the application
Application State: data handled by the application



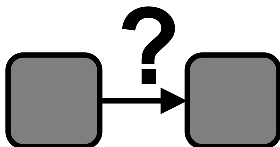
Distribution: applications are decomposed to…
… use multiple cloud resources
… support the fact that clouds are large globally distributed systems



Elasticity: applications can be scaled out dynamically
Scale out: performance increase through addition of resources
Scale up: performance increase by increasing resource capabilities



Automated Management: runtime tasks have to be handled quickly
Example: exploitation of pay-per-use by changing resource numbers
Example: resiliency by reacting to resource failures



Loose Coupling: influence of application components is limited
Example: failures should not impact other components
Example: addition / removal of components is simplified

© Olaf Zimmermann, 2015.

The Twelve-Factor App (Author: Heroku Co-Founder)

- **Not cloud-specific**
- **Mix of agile and DevOps practices**
- **In line with IDEAL, Amazon, ARC/CDAR**

http://12factor.net/

I. Codebase
One codebase tracked in revision control, many deploys

II. Dependencies
Explicitly declare and isolate dependencies

III. Config
Store config in the environment

IV. Backing Services
Treat backing services as attached resources

V. Build, release, run
Strictly separate build and run stages

VI. Processes
Execute the app as one or more stateless processes

VII. Port binding
Export services via port binding

VIII. Concurrency
Scale out via the process model

IX. Disposability
Maximize robustness with fast startup and graceful shutdown

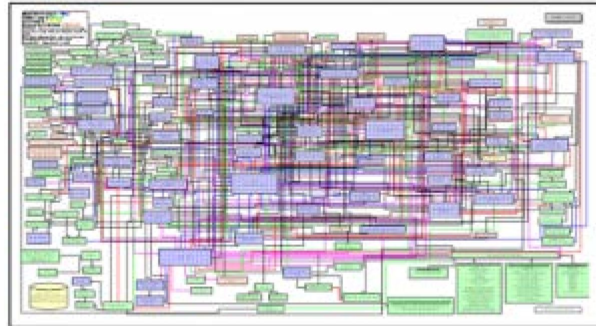
X. Dev/prod parity
Keep development, staging, and production as similar as possible

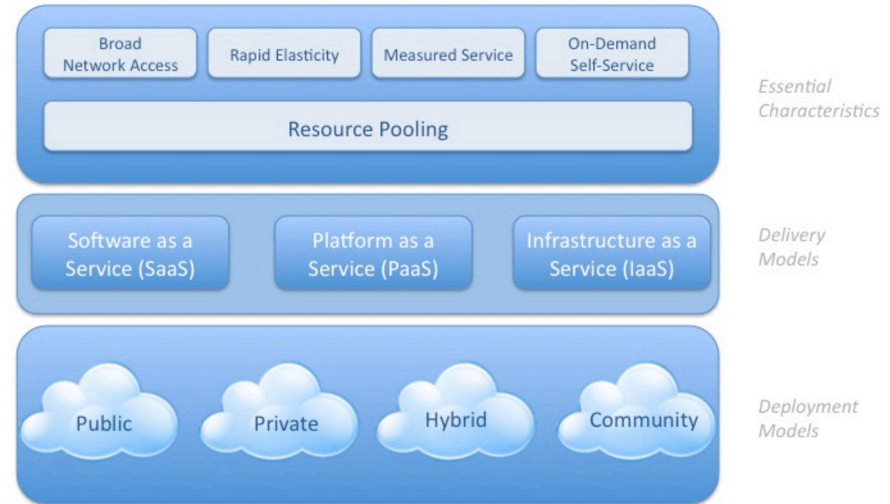
XI. Logs
Treat logs as event streams

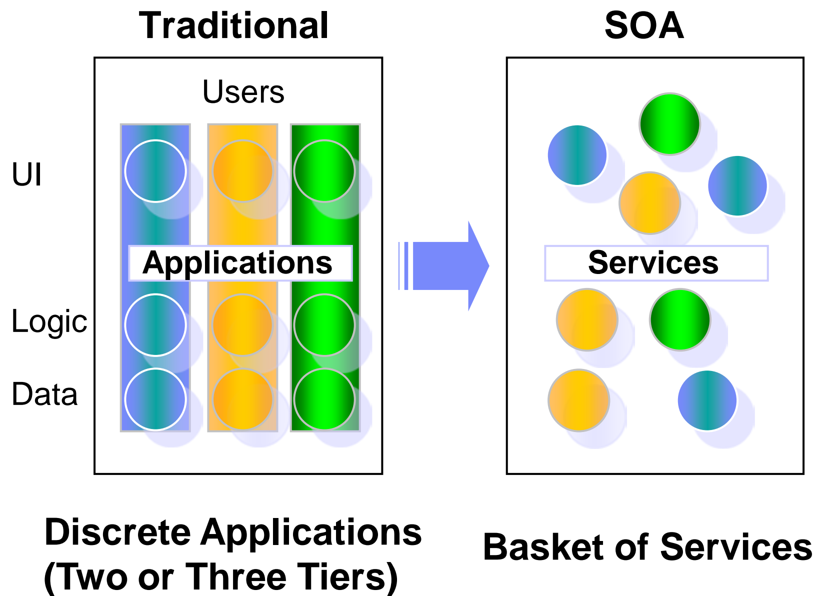
XII. Admin processes
Run admin/management tasks as one-off processes

© Olaf Zimmermann, 2015.

From Traditional Layer-Tier Architectures to Cloud Services







?

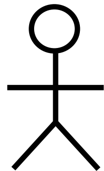
Discrete Applications (Two or Three Tiers)

Basket of Services

Decision-Centric Architectural Refactoring for Cloud (ARC)

© Olaf Zimmermann, 2015.

Another Cloud User Story: Cloud Migration



Application Maintainer

As a **developer who maintains and operates an existing application on behalf of a client,**

I would like to **move the on-premises production site into a cloud**

so that **I no longer have to worry about security updates and other administrative tasks on the operating system and the middleware level – and my client has to spend less on operations, which frees resources to develop new features.**

To do so, **I need to find out what/how my application architecture has to be changed to be ready for the cloud (first and foremost, it should be able to run in the cloud; as a second step, it should take advantage of cloud features such as elasticity).**

© Olaf Zimmermann, 2015.

Agenda

- **Motivation**
- **Cloud Computing Fundamentals**
- **IDEAL Cloud Applications and Cloud Computing Patterns**

- **Architectural Refactoring vs. Code Refactoring**
- **Quality Attribute Stories and Architectural Refactoring Templates**

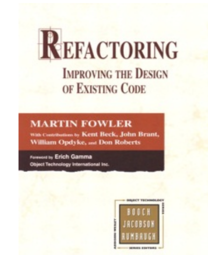
- **Architectural Refactoring for the Cloud**

- **Tool Support for Architectural Knowledge Management (AKM)**

© Olaf Zimmermann, 2015.

Code Refactoring vs. Architectural Refactoring

- **Refactoring are “small behavior-preserving transformations” (M. Fowler 1999)**
- **Code refactorings, e.g. “extract method”**
  - Operate on Abstract Syntax Tree (AST)
  - Based on compiler theory, so automation possible (e.g., in Eclipse Java/C++)
- **Catalog and commentry: http://refactoring.com/**



Refactor | Navigate | Search | Project | Run | Window | Help

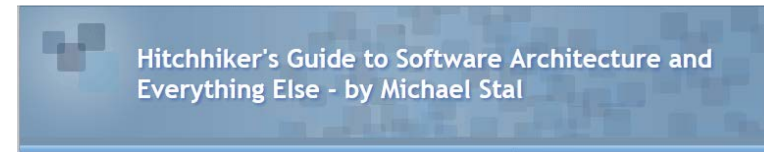
Rename...	Alt+Shift+R
Move...	Alt+Shift+V
Change Method Signature...	Alt+Shift+C
Extract Method...	Alt+Shift+M
Extract Local Variable...	Alt+Shift+L
Extract Constant...	
Inline...	Alt+Shift+I
Convert Anonymous Class to Nested...	
Convert Member Type to Top Level...	
Convert Local Variable to Field...	
Extract Superclass...	
Extract Interface...	
Use Supertype Where Possible...	
Push Down...	
Pull Up...	
Extract Class...	
Introduce Parameter Object...	
Introduce Indirection...	
Introduce Factory...	
Introduce Parameter...	
Encapsulate Field...	
Generalize Declared Type...	
Infer Generic Type Arguments...	
Migrate JAR File...	
Create Script...	
Apply Script...	
History...	

- **Architectural refactorings**
  - Resolve one or more *architectural smells*, have an impact on quality attributes
    - Architectural smell: suspicion that architecture is no longer adequate (“good enough”) under current requirements and constraints (which may differ form original ones)
  - Are carriers of reengineering knowledge (patterns?)
  - Can only be partially automated

© Olaf Zimmermann, 2015.

Early Work on Architectural Refactoring: Michael Stal (Siemens)

- **First [blog post](#) on architecture refactoring**
  - Motivation, discussion, pattern template
- **[OOPSLA 2007 tutorial](#) (and OOP session)**
  - First catalog of architectural refactorings
- **CompArch/WICSA 2011 industry day keynote**
  - Catalog update (available [here](#))



Thursday, January 25, 2007

Architecture Refactoring

- *Partition Responsibilities*: If a component or subsystem got too many responsibilities, partition the component or subsystem into multiple parts, each of which with semantically related functionality.
- *Extract Service*: If a subsystem does not provide any interfaces to its environment but is subject of external integration, extract service interface.
- Introduce decoupling layer: If components directly depend on system details, introduce decoupling layer(s).
- *Rename Entity*: If entities got unintuitive names, introduce appropriate naming scheme.
- *Break Cycle*: When encountering a cycle on subsystem level, break it.
- Merge functionality: If there is broad coheshion between two modules, merge them.
- *Orthogonalize*: If two parts of an architecture introduce different solutions for the same problem, choose one preferred solution and eliminate the other.
- *Introduce strict layering*: If in a layered system, a layer accesses lower layers without necessity (relaxed layering), enforce strict layering.
- *Introduce hierarchies*: If several entities are only variants of a particular entity, introduce a hierarchy.
- *Introduce Interceptor hooks*: If we have to open an architecture for out-of-band functionality according to the Open/Close principle interceptors should be introduced.
- *Eliminate dependencies by dependency injection*: Reduce direct and wide-spread dependencies of Parts in a Whole/Part setting by introducing a central runtime component (Whole´) that centralizes dependency handling with dependency injection.

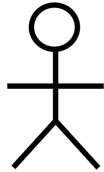
Some Examples

1. Rename Entities
2. Remove Duplicates
3. Introduce Abstraction Hierarchies
4. Remove Unncessary Abstractions
5. Substitute Mediation with Adaptation
6. Break Dependency Cycles
7. Inject Dependencies
8. Insert Transparency Layer
9. Reduce Dependencies with Facades
10. Merge Subsystems
11. Split Subsystems
12. Enforce Strict Layering
13. Move Entities
14. Add Strategies
15. Enforce Symmetry
16. Extract Interface
17. Enforce Contract
18. Provide Extension Interfaces
19. Substitute Inheritance with Delegation
20. Provide Interoperability Layers
21. Aspectify
22. Integrate DSLs
23. Add Uniform Support to Runtime Aspects
24. Add Configuration Subsystem
25. Introduce the Open/Close Principle
26. Optimize with Caching
27. Replace Singleton
28. Separate Synchronous and Asynchronous Processing
29. Replace Remote Methods with Messages
30. Add Object Manager
31. Change Unidirectional Association to Bidirectional

© Olaf Zimmermann, 2015.

Quality Story* (Inspired by User Stories) – Example

* missing in agile practices guide so far
(tradeoff: benefit vs. negative consequences)



DevOps Engineer

http://en.blog.doodle.com/2011/04/14/doodles-technology-landscape

As a **Development and Operations (DevOps) engineer at a social network/media firm,**

I would like to **be able to add attributes to my database w/o having to migrate data – without changing the functional scope of the system –**

so that in future versions of the system**:**

- **New features of the Web software can be introduced more often.**
- **It is no longer needed to migrate the large amount of existing data to new schemas.**
- **We become independent of the provider of the current RDBMS.**

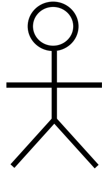
To achieve this goal, I am willing to accept**:**

- **Data access and data validation logic becomes more complex.**
- **Five developer days have to be invested .**
- **Technical feasibility and performance have to be validated in a PoC.**

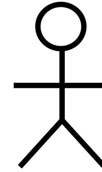
© Olaf Zimmermann, 2015

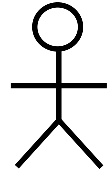
Quality Story (Inspired by User Stories) – Template

Software Maintainer



Operator, Identity and Access Manager (IAM), Database Administrator





Release Architect, Product Manager, Application Owner

As a **[role concerned with system quality, e.g. a leadership or maintenance role],**

I would like to **[achieve quality goal A]**
– without changing the functional scope of the system –

so that in future versions of the system:

- **[technical debt reduction effect]**
- **[improved service level/system property]**
- **[positive impact on other technical constraints and environment]**

To achieve this goal, I am willing to invest/accept **:**

- **[impact on other quality attributes, e.g. performance penalty for security feature]**
- **[impact on project plan (cost, timeline)]**
- **[impact on technical dependencies and risk]**

© Olaf Zimmermann, 2015

Architectural Refactorings – Decision-/Task-Centric Template

Architectural Refactoring: **[Name]**	
Context (viewpoint, refinement level): • […]	*Quality attributes and stories (forces):* • […]
Architectural smell (refactoring driver): • […]	
Architectural decision(s) to be revisited: • […]	
Refactoring (solution sketch/evolution outline): • […]	
Affected components and connectors (if modelled explicitly): • […]	
Execution tasks (in agile planning tool and/or full-fledged design method): • […]	

© Olaf Zimmermann, 2015.

Architectural Refactoring – Example (of Filled Out Template)

***Architectural Refactoring:* Move Responsibility**	
Context (viewpoint, refinement level): • Logical viewpoint, platform-independent level	*Quality attributes and stories (forces):* • High cohesion and low coupling (metrics)
Smell (refactoring driver): • A component seems to be overloaded and cluttered with diffuse features in its external interface	
Architectural decision(s) to be revisited: • Approach to modularization and component partitioning • Use of industry reference models • API design guidelines	
Refactoring (solution sketch/evolution outline): • Assess cohesion and coupling of a particular component (are responsibilities semantically related?) • Move a responsibility that breaks cohesion to another component (note: this can be an existing component or a new one; one or more responsibilities can be moved at once)	
Affected components and connectors (if modelled explicitly): • Component that currently provides a certain service (i.e., operation/feature) • Component that will take over this responsibility	
Execution tasks (in agile planning tool and/or full-fledged design method): • Updates to component specification in word processor, drawing tool, documentation wiki • Edit operations in UML or other modeling tool • Updates to component realizations in code (note: architecturally evident coding style to be followed)	

© Olaf Zimmermann, 2015.

Fundamental Refactorings (Logical/Functional Viewpoint)

- **Add Layer**
- **Collapse Layers (into Single One)**
- **Add Tier**
- **Collapse Tiers (into One)**
- **Split Component**
- **Merge Components**
- **Move Responsibility (to New/to Existing Component)**
  - Examples: component initialization, input validation, execution strategy
- **Split Connector into Component and Connectors (Re-ify Collaboration)**
- **Expose Component Interface as Remote Service**
- **Replace Service Provider**
  - Note: some of these refactorings have impact on component collaborations

Architectural Refactoring: [Name]	
Context (viewpoint, refinement level): • [...]	*Quality attributes and stories (forces):* • [...]
Smell (refactoring driver): • [...]	
Architectural decision(s) to be revisited: • [...]	
Refactoring (solution sketch/evolution outline): • [...]	
Affected components and connectors (if modelled explicitly): • [...]	
Execution tasks (in agile planning tool and/or full-fledged design method): • [...]	

© Olaf Zimmermann, 2015.

Fundamental Refactorings (Infrastructure/Deployment Viewpoint)

- Move Deployment Unit (from One Server Node to Another)
- Introduce Clustering
- Add Cluster Node
- Add Load Balancer
- Add Firewall
- Introduce Cache
- Change Caching Policy
- Load Lazier
- Move Application State Management to Client/to Server/to Database
- Scale Up
- Scale Out

Architectural Refactoring: [Name]	
Context (viewpoint, refinement level): • [...]	*Quality attributes and stories (forces):* • [...]
Smell (refactoring driver): • [...]	
Architectural decision(s) to be revisited: • [...]	
Refactoring (solution sketch/evolution outline): • [...]	
Affected components and connectors (if modelled explicitly): • [...]	
Execution tasks (in agile planning tool and/or full-fledged design method): • [...]	

© Olaf Zimmermann, 2015.

Architectural Refactoring – Example (in Infrastructure Viewpoint)

Architectural Refactoring: **Introduce Cache**

Context (viewpoint, refinement level):	*Quality attributes and stories (forces):*
• Logical, platform-specific refinements	• Performance (response time)

Smell (refactoring driver):
- A data store cannot handle concurrent queries (read requests) in reasonable time

Architectural decision(s) to be revisited:
- Lookup strategy
- Data structure selection
- Location of data store including replication (in memory, on disk)

Refactoring (solution sketch/evolution outline):
- Add an intermediate data structure such as memcached to speed up lookups
- Design cache interface and behavior (e.g., cache size and cache cleanup policies) and cache item identifier (e.g., URI for HTML page/request caching)

Affected components and connectors (if modelled explicitly):
- Cached data and its master data store
- Clients accessing this data
- IT infrastructure hosting the cache (e.g., memory and/or disk storage)

Execution tasks (in agile planning tool and/or full-fledged design method):
- Analyze read-write access profile
- Measure improvement potential of caching in a PoC, assess impact on test and operations (tech. risk)
- Implement cache, test cache usage, document caching policies and configuration options

© Olaf Zimmermann, 2015.

Agenda

- Motivation
- Cloud Computing Fundamentals
- IDEAL Cloud Applications and Cloud Computing Patterns
- Architectural Refactoring vs. Code Refactoring
- Quality Attribute Stories and Architectural Refactoring Templates
- Architectural Refactoring for the Cloud
- Tool Support for Architectural Knowledge Management (AKM)

© Olaf Zimmermann, 2015.

Towards an Architectural Refactoring Catalog for Cloud

- **Change cloud application architecture pattern(s):**
  - E.g. from server session state to database session state management to support horizontal scaling (sharding)
  - E.g. from normalized to partitioned/replicated master data to support NoSQL storage of transactional data
  - E.g. from flat rate to usage-based billing to support elasticity in a cost-efficient manner

***Architectural Refactoring:* [Name]**	
Context (viewpoint, refinement level): • [...]	*Quality attributes and stories (forces):* • [...]
Smell (refactoring driver): • [...]	
Architectural decision(s) to be revisited: • [...]	
Refactoring (solution sketch/evolution outline): • [...]	
Affected components and connectors (if modelled explicitly): • [...]	
Execution tasks (in agile planning tool and/or full-fledged design method): • [...]	

© Olaf Zimmermann, 2015.

Architectural Refactoring for Cloud – Example: De-SQL

***Architectural Refactoring:* De-SQL**	
Context (viewpoint, refinement level): • Logical viewpoint, data viewpoint (all levels)	*Quality attributes and stories (forces):* • Flexibility, data integrity
Architectural smell (refactoring driver): • It takes rather long to update the data model and to migrate existing data	
Architectural decision(s) to be revisited: • Choice of data modeling paradigm (current decision is: relational) • Choice of metamodel and query language (current decision is: SQL)	
Refactoring (solution sketch/evolution outline): • Use document-oriented database such as MongoDB instead of RDBMS such as MySQL • Redesign transaction management and database administration	
Affected components and connectors (if modelled explicitly): • Database • Data access layer	
Execution tasks (in agile planning tool and/or full-fledged design method): • Design document layout (i.e., the pendant to the machine-readable SQL DDL) • Define index for document access • Write new data access layer, implement SQLish query capabilities yourself • Decide on transaction boundaries (if any), document database administration (CRUD, backup)	

© Olaf Zimmermann, 2015.

Candidate Architectural Refactorings for Cloud (Draft Catalog)

Category	Refactorings		
IaaS	Virtualize Server	Virtualize Storage	Virtualize Network
IaaS, PaaS	Swap Cloud Provider	Change Operating System	Open Port
PaaS	"De-SQL"	"BASEify" (remove "ACID")	Replace DBMS
PaaS	Change Messaging QoS	Upgrade Queue Endpoint(s)	Swap Messaging Provider
SaaS/application	Increase Concurrency	Add Cache	Precompute Results
SaaS/application	(CCP book, CBDI-SAE)	(all Stal refactorings)	(PoEAA/Fowler patterns)
Scalability	Change Strategy (Scale Up vs. Scale Out)	Replace Own Cache with Provider Capability	Add Cloud Resource (xaaS)
Performance	Add Lazy Loading	Move State to Database	
Communication	Change Message Exchange Pattern	Replace Transport Protocol	Change Protocol Provider
User management	Swap IAM Provider	Replicate Credential Store	Federate Identities
Service/deployment model changes	Move Workload to Cloud (use XaaS)	Privatize Deployment, Publicize Deployment	Merge Deployments (Use Hybrid Cloud)

© Olaf Zimmermann, 2015.

Agenda

- Motivation
- Cloud Computing Fundamentals
- IDEAL Cloud Applications and Cloud Computing Patterns
- Architectural Refactoring vs. Code Refactoring
- Quality Attribute Stories and Architectural Refactoring Templates
- Architectural Refactoring for the Cloud
- Tool Support for Architectural Knowledge Management (AKM)

© Olaf Zimmermann, 2015.

How to Manage this Architectural (Reengineering) Knowledge?

- **Needed: Rich text, hyperlinks, collaboration (among users), tool integration**

Architectural Refactoring: **[Name]**	
Context (viewpoint, refinement level): • […]	*Quality attributes and stories (forces):* • […]
Smell (refactoring driver): • […]	
Architectural decision(s) to be revisited: • […]	
Refactoring (solution sketch/evolution outline): • […]	
Affected components and connectors (if modelled explicitly): • […]	
Execution tasks (in agile planning tool and/or full-fledged design method): • […]	

- **Delivery: Knowledge base/software architecture handbook/Q&A site?**
  - “ArchiPedia”, StackOverflow (for Designers/Maintainers), ar-aaS?

© Olaf Zimmermann, 2015.

Architectural Refactoring Tool Support (Thesis Projects)

- **Web Collaboration Tool based on AngularJS, Play, Postgres/MySQL**

Architectural Refactoring Tool

ARs deal with architecture documentation and the architecture's manifestation in the code and run-time artifacts. A single architectural syntax tree doesn't exist. ARs pertain to components and connectors (modeled, sketched, or represented implicitly in code), design decision logs (in structured or unstructured text), and planning artifacts such as work items in project management tools. ARs address architectural smells, which are suspicions or indications that something in the architecture is no longer adequate under the current requirements and constraints, which might differ from the original ones. An AR, then, is a coordinated set of deliberate architectural activities that remove a particular architectural smell and improve at least one quality attribute without changing the system's scope and functionality. An AR might negatively influence other quality attributes, owing to conflicting requirements and tradeoffs.

© Olaf Zimmermann, 2015.

More Information – Architectural Refactoring

- **Architectural Refactoring**
  - M. Stal, Refactoring Software Architecture, Chapter 3 in Agile Software Architecture, Elsevier 2013

CSDL Home » IEEE Software » 2015 vol.32 » Issue No.02 - Mar.-Apr.

IEEE Software

Architectural Refactoring: A Task-Centric View on Software Evolution

Issue No.02 - Mar.-Apr. (2015 vol.32)
pp: 26-29

Olaf Zimmermann , Institute for Software at the University of Applied Sciences of Eastern Switzerland, Rapperswil

DOI Bookmark: http://doi.ieeecomputersociety.org/10.1109/MS.2015.37

ABSTRACT

A refactoring aims to improve a certain quality while preserving others. For example, code refactoring restructures code to make it more maintainable without changing its observable behavior. Given the success of code refactoring, it's surprising that architectural refactoring (AR) hasn't taken off yet. This article examines AR from a new angle: as an evolution technique that revisits architectural decisions and identifies related design, implementation, and documentation tasks.

INDEX TERMS

Software development, Computer architecture, Context modeling, Software engineering, Catalogs, Pragmatics, Software architecture,software engineering, refactoring, software evolution, software development

CITATION

Olaf Zimmermann, "Architectural Refactoring: A Task-Centric View on Software Evolution", *IEEE Software*, vol.32, no. 2, pp. 26-29, Mar.-Apr. 2015, doi:10.1109/MS.2015.37

© Olaf Zimmermann, 2015.

ARCHITECTURAL REFACTORING FOR THE CLOUD: A DECISION-CENTRIC VIEW ON CLOUD MIGRATION – BACKGROUND INFORMATION

9th Symposium and Summer School On Service-Oriented Computing

Prof. Dr. Olaf Zimmermann

Distinguished (Chief/Lead) IT Architect, The Open Group

Institute für Software, HSR FHO

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

What are Architectural Decisions (ADs)? Why Care?

Reference: SEI SATURN 2010 (IBM presentation)

- **"The design decisions that are costly to change" (Grady Booch, 2009)**
- **A more elaborate definition:**

"Architectural decisions capture key design issues and the rationale behind chosen solutions. They are conscious design decisions concerning a software-intensive system as a whole or one or more of its core components and connectors in any given view. The outcome of architectural decisions influences the system's nonfunctional characteristics including its software quality attributes."

- **From IBM UMF work product description ART 0513 (since 1998):**

"The purpose of the Architectural Decisions work product is to:

- Provide a single place to find important architectural decisions
- Make explicit the rationale and justification of architectural decisions
- Preserve design integrity in the provision of functionality and its allocation to system components
- Ensure that the architecture is extensible and can support an evolving system
- Provide a reference of documented decisions for new people who join the project
- Avoid unnecessary reconsideration of the same issues"

© Olaf Zimmermann, 2015.

Y-Template for Architectural Decision Capturing

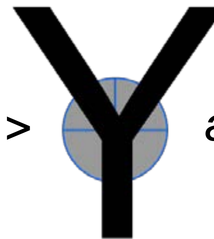
Reference: Sustainable Architectural Design Decisions, IEEE Software 30(6): 46-53 (2013)

- **Link to (non-)functional requirements and design context**
- **Tradeoffs between quality attributes**

In the context of <use case uc and/or component co>,

… facing <non-functional concern c>,

… we decided for <option o1>



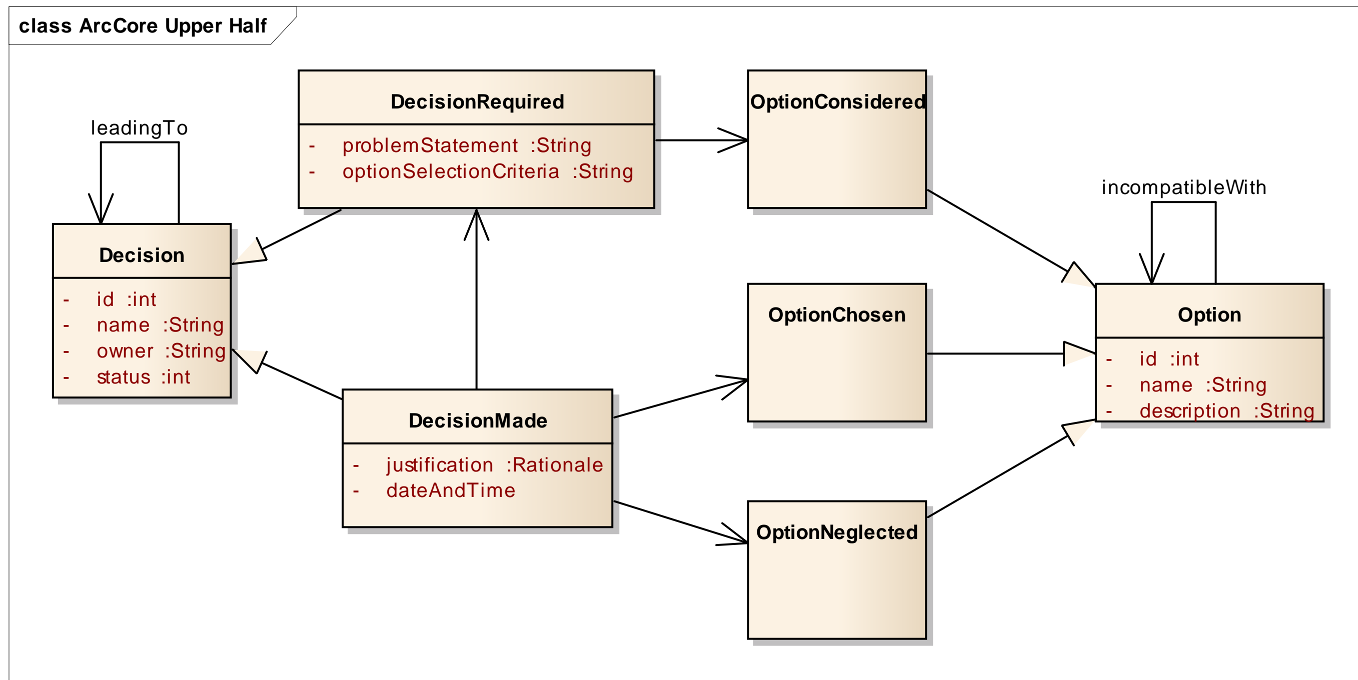
and neglected <options o2 to oN>,

… to achieve <quality q>,

… accepting downside <consequence c>.

© Olaf Zimmermann, 2015.

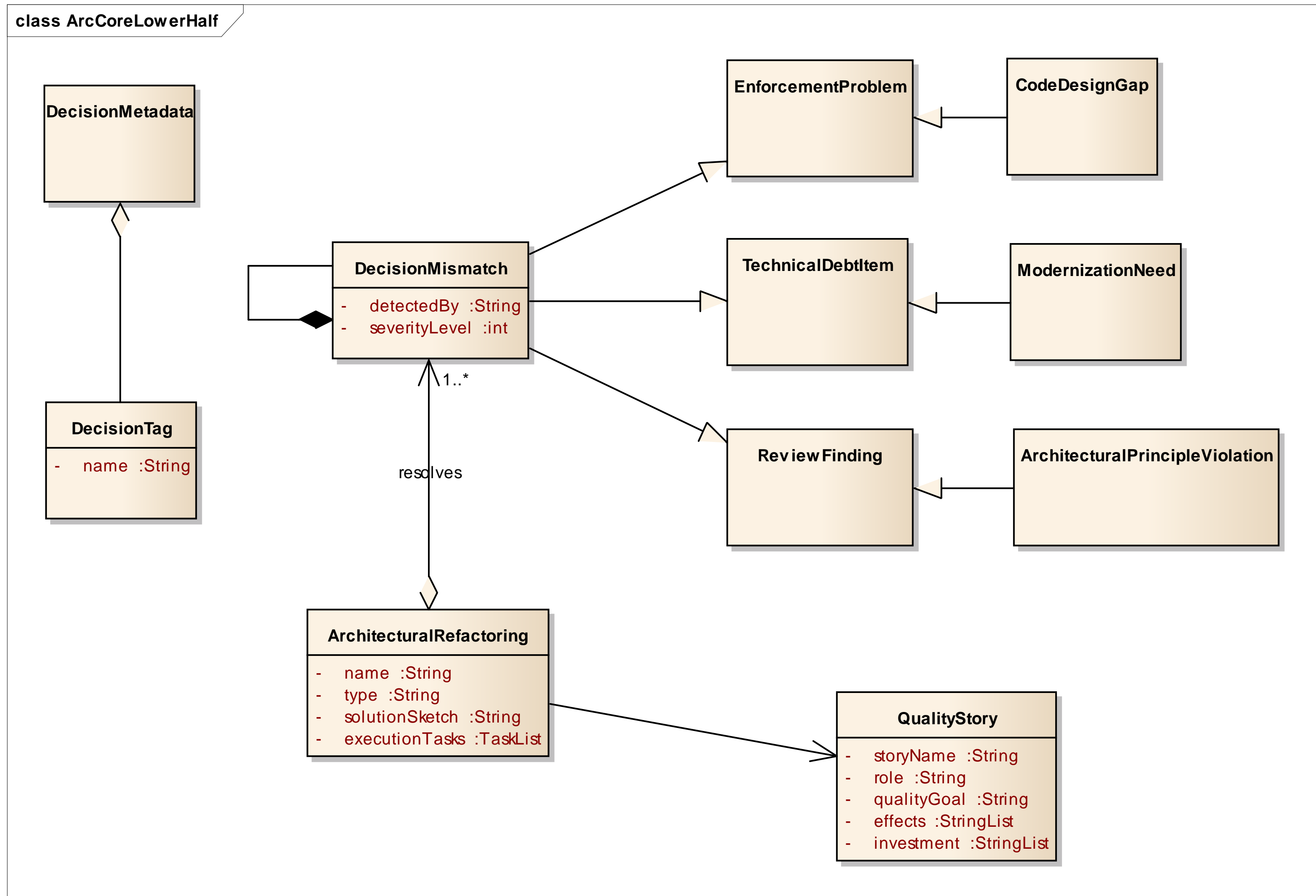
ARC Metamodel (at an Initial State of Elaboration) (1/2)



- **Refactoring need arises from decision *mismatches***
  - Decision actually made vs. recommended decision (IDEAL)
  - Same problem (to be) solved differently (choosing different option)
  - Refactoring improves at least one quality attribute and preserves functionality

© Olaf Zimmermann, 2015.

ARC Metamodel (at an Initial State of Elaboration) (2/2)



HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

© Olaf Zimmermann, 2015.

IFS INSTITUTE FOR SOFTWARE

PaaS Platform Selection/Evaluation Criteria (1/2)

- **Support for defining cloud characteristics ("OSSM"):**
  - On demand, Self service, Scalable, Measured
- **Billing model and Service Level Agreements (SLAs)**
  - Accountability of provider, penalties/refunds, customer obligations
- **Physical location (of data)**
- **Country/place of jurisdiction (CH/EU/other)**
- **Service scope**
  - Platform middleware versions?
  - Limitations:
    - Can main programs (batch jobs) be run?
    - Can JEE EARs be deployed?

© Olaf Zimmermann, 2015.

PaaS Platform Selection/Evaluation Criteria (2/2)

- **Deployment process and tools; standardization**
  - Web console
  - Management APIs
  - Local SDK (command line tools, Eclipse plugins)
  - Topology and Orchestration Specification for Cloud Applications (TOSCA)
- **User/programmer documentation incl. getting started information**
- **Cloud services lifecycle, e.g. hibernation due to inactivity/restart time?**
- **Operational model (runtime topologies)**
  - Inbound traffic, outbound traffic, cloud-internal communication
- **Domain and port management capabilities for user**
  - E.g. own URIs/domain names possible (DNS management)?
  - Can virtual hosts (custom DNS entries) be defined?
- **API security and VPN support**
  - Credentials, storage locations

© Olaf Zimmermann, 2015.

Architectural Principles for Cloud-Native Applications

- **Design application startup and restart procedures as lean as possible**
  - How long does it take your application server to display an “open for e-business” message after a restart (process and/or hardware)?
- **Let all components implement the Service Layer pattern**
  - Define with Remote Facades and expose them with JAX-WS or JAX-RS
  - Use messaging for cloud-internal communication and integration
- **Define all Data Transfer Objects (DTOs) to be serializable**
  - See experiment with DDD Sample in PaaS Provider 1 (Spring MVC)
- **Use Internet security technologies to satisfy application security needs**
  - E.g. often no connectivity to company-internal LDAP or Active Directory
- **Model all communication dependencies explicitly and consult IT infrastructure architects both on provider and on consumer side**
  - E.g. one PaaS Provider requires inbound port 5000 connectivity to support remote terminals (required for platform/instance management)

© Olaf Zimmermann, 2015.

Good Cloud Design Practices

- **Avoid calls to proprietary platform libraries (e.g., via JNI)**
- **Limit usage of expensive operations, e.g. SecureRandom in Java SE**
- **Do not define resource identifiers such as IP addresses statically**
- **Prefer HTTP over raw socket communication even for cloud-internal integration (or use messaging capabilities offered by cloud provider)**
- **Do not expect cloud messaging to have the same semantics and QoS as traditional messaging systems (at-least-once vs. exactly-once delivery)**
- **Do not expect NoSQL storage to provide the same level of programming and database management convenience as mature SQL database systems**
- **Do not expect cloud provider to handle backup and recovery of application data for you**
- **Be prepared to log resource consumption on same level of detail as provider (in case bill from provider contains suspicious items)**

© Olaf Zimmermann, 2015.

Cloud Affinity of PoEAA Patterns (1/3)

PoEAA Pattern	Suitability for Cloud	Comment
Client Session State	Yes and no	As good or bas as in traditional deployment (security?)
Server Session State	No (I in IDEAL violated)	Also hinders scale out
Database Session State	Yes	Can use DB (e.g. NoSQL)
Model-View-Controller	Yes (with persistent model)	Web frontends are cloud-affine
Front Controller	Yes (Web frontends)	See above
Page Controller	Yes (Web frontends)	See above
Application Controller	Yes (Web frontends)	See above
other Presentation Layer Patterns	Yes (Web frontends)	See above

Patterns of Enterprise Application Architecture Patterns (PoEAA):
http://martinfowler.com/eaaCatalog/

© Olaf Zimmermann, 2015.

Cloud Affinity of PoEAA Patterns (2/3)

PoEAA Pattern	Suitability for Cloud	Comment
Transaction Script	Yes	Procedures should be self contained (stateless interactions)
Domain Model	Depends on complexity of domain model	Object tree in main memory might limit scale out (and database partitioning)
Table Module	No or implementation dependent	Big data sets problematic unless partitioned (e.g. map-reduce)
Service Layer	Yes	SOA and REST design principles should be adhered to, e.g. no object references in domain model, but only instances of Data Transfer Object in interface (larger discussion required)
Remote Facade	Yes	Can be introduced for cloud enablement of existing solutions; can wrap calls to PaaS provider to support maintainability and portability

Patterns of Enterprise Application Architecture Patterns (PoEAA):
http://martinfowler.com/eaaCatalog/

© Olaf Zimmermann, 2015.

Cloud Affinity of PoEAA Patterns (3/3)

PoEAA Pattern	Suitability for Cloud	Comment
Active Record	Limited	Good when RDB exists in cloud or when records have simple structures; complex structures can be difficult to handle for NoSQL storage (mapping need)
Row Data Gateway	Yes	Fits scale out
Table Data Gateway	No or implementation dependent	Big data sets problematic unless partitioned (e.g. map-reduce)
System Transaction	Depends on cloud storage capabilities (NoSQL?)	Larger discussion required (CAP BASE vs. ACID etc.)
Business Transaction	Yes	If cloud design best practices are adhered to (statelessness etc.)

Patterns of Enterprise Application Architecture Patterns (PoEAA):
http://martinfowler.com/eaaCatalog/

© Olaf Zimmermann, 2015.