Institut für Software

OOP 2014
SOFTWARE MEETS BUSINESS

ARCHITECTURAL REFACTORING – AGILE UMSETZUNG VON MODERNISIERUNGS-ENTSCHEIDUNGEN

IFS INSTITUTE FOR SOFTWARE

Prof. Dr. Olaf Zimmermann
Distinguished (Chief/Lead) IT Architect, The Open Group
ozimmerm@hsr.ch
München, 5. Februar 2014

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Who Am I

- **R+D und Professional Services-Erfahrung seit 1994**
  - em. IBM Executive IT Architect (senior certified by The Open Group)
    - Systems & Network Management, J2EE, Enterprise Application Integration/SOA
  - em. ABB Senior Principal Scientist
    - Enterprise Architecture Management/Legacy System Modernization/Remoting
- **Diverse Industrieprojekte und Coachings**
  - R+D und IT-Consulting für Middleware, Informationssysteme, Tools
  - Tutorials: UNIX/RDBMS, OOP/C++/J2EE, MDSE/MDA, SOA/XML
- **Schwerpunkt @ HSR: Entwurf verteilter Systeme**
  - Cloud, SOA, Web Application Development (Runtime)
  - Architekturentscheidungen, Architectural Refactoring (Build Time)

© Olaf Zimmermann, 2014.

Beschreibung des Vortrags

- **Abbau von Technical Debt durch Refactoring auf Ebene des Programmcodes ist eine verbreitete agile Praktik. Architectural Refactoring wurde als Konzept bereits vorgeschlagen, ist aber in der Praxis noch nicht verbreitet.**
- **Dieser Vortrag etabliert Architectural Refactoring als methodischen Ansatz zur schrittweisen Modernisierung von Legacy Software und stellt einen Katalog fundamentaler Refactorings vor.**
  - Add Layer, Add Tier
  - Merge Components, Move Responsibilities, Move Collaboration
  - Expose Interface as Service
- **Der Vortrag schliesst mit einem Ausblick auf domänenspezifische Refactorings z.B. für Cloud Computing und auf die Unterstützung der präsentierten Praktiken in agilen Tools.**
  - De-SQL
  - Move Session State Management to Database

© Olaf Zimmermann, 2014.

Agenda

1. **Motivation anhand von Fallbeispielen**
2. **Von Code Refactoring zu Architectural Refactoring**
   - Definitionen
   - Domain Model
3. **Struktur des Katalogs fundamentaler Refactorings**
4. **Ausgewählte Refactorings im Detail**
5. **Umsetzung im Projekt: agile Aktivitätenplanung**
6. **Ausblick auf Werkzeugunterstützung**
7. **Ausblick auf Architectural Refactoring for Cloud**

© Olaf Zimmermann, 2014.

Agenda

1. **Motivation**
2. Existing work
3. From code refactoring and architectural decisions to architectural refactoring
4. Catalog structure and refactoring template
5. Some basic architectural refactorings
6. Tool support
7. Architectural Refactoring for Cloud (ARC)

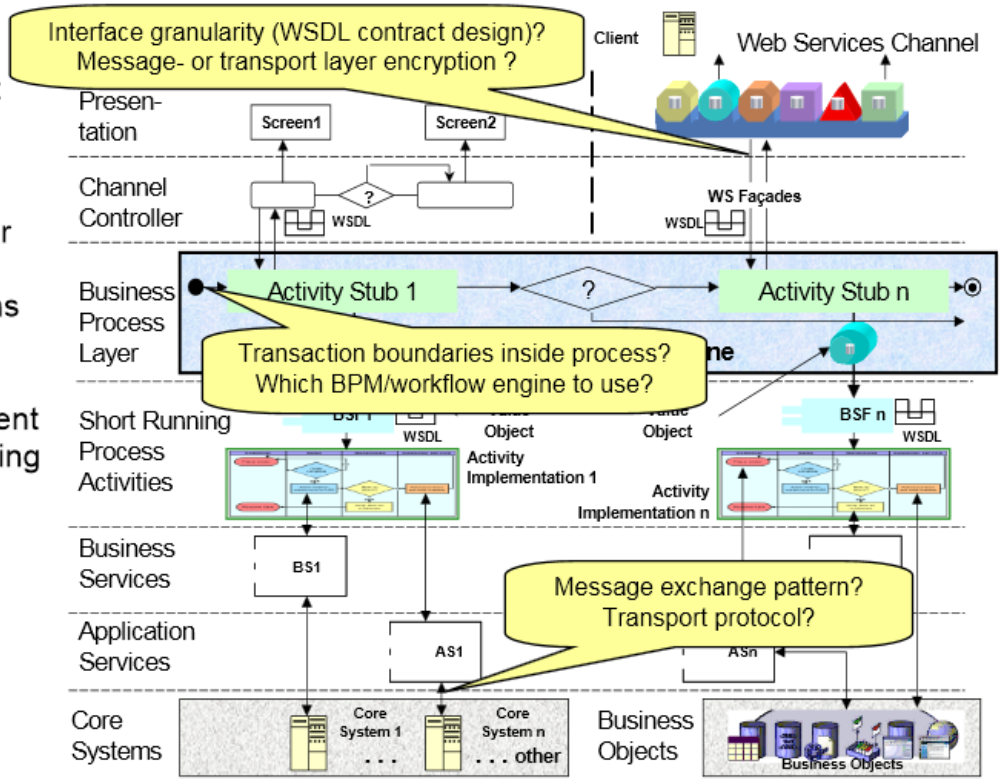
© Olaf Zimmermann, 2014.

Context: Architectural Decisions in Sample Architecture

Reference: IBM, ECOWS 2007

Multi-Channel Order Management SOA in the Telecommunications Industry (in production since Q1/2005) [OOPSLA 2005]

- **Functional domain**
  - Order entry management
  - Two business processes: new customer, relocation
  - Main SOA drivers: deeper automation grade, share services between domains
- **Service design**
  - Top-down from requirement and bottom-up from existing wholesaler systems
  - Recurring architectural decisions:
    - Protocol choices
    - Transactionality
    - Security policies
    - Interface granularity



© 2007 IBM Corporation

© Olaf Zimmermann, 2014.

Vision: From Architectural Decisions to Architectural Refactoring

- **An architectural refactoring revisits certain architectural decisions and selects different alternatives (options) for solving a given set of design problems**
  - This new definition of architectural refactoring put less emphasis on structure – and more emphasis on design rationale and related tasks
- **The revision of a group of related decisions may lead to:**
  - Structural changes in the design
  - Implementation and configuration activities (in development and/or operations, depending on viewpoint the refactoring pertains to)
  - Documentation tasks (modeling, technical writing)
- **In the context of legacy system modernization, some of the decisions to be revisited may have been taken a long time ago**
  - So rationale for original decision might have been lost
- **On agile projects, the design might be documented mostly tacitly**
  - E.g. in code, in agile planning tools, in carbon-based knowledge carriers

© Olaf Zimmermann, 2014.

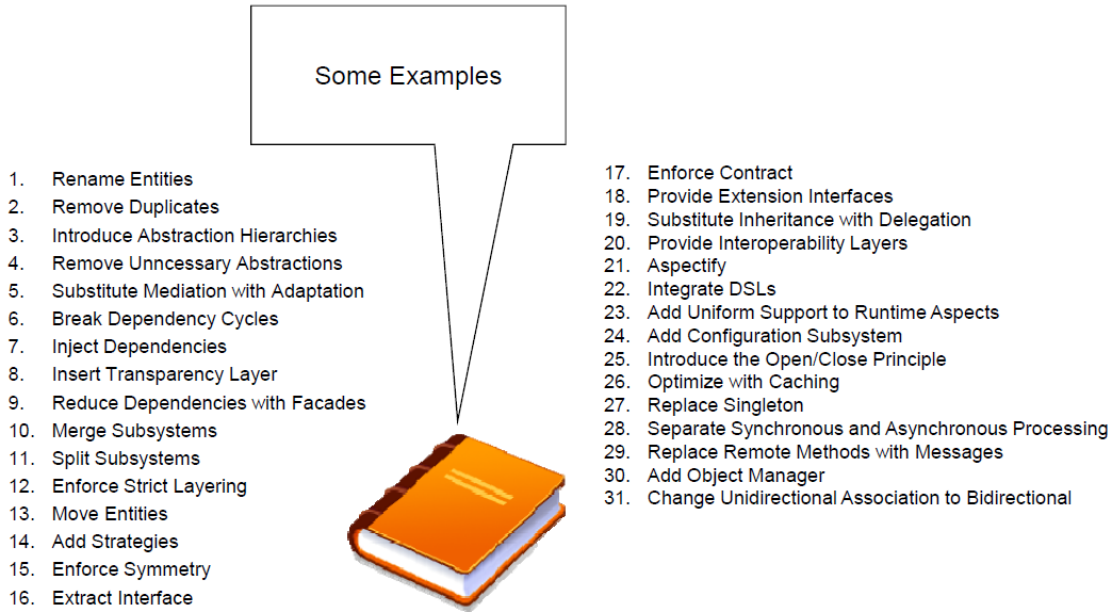
Agenda

1. Motivation
2. **Existing work**
3. From code refactoring and architectural decisions to architectural refactoring
4. Catalog structure and refactoring template
5. Some basic architectural refactorings
6. Tool support
7. Architectural Refactoring for Cloud (ARC)

© Olaf Zimmermann, 2014.

Existing Work (1/3): Seminal Work of Michael Stal

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



© Olaf Zimmermann, 2014.

Existing Work (2/3): Contributions by George Fairbanks

- **Just Enough Software Architecture (Marshall & Brainerd, 2010)**
  - Draws connection between architecture design and risk management
  - Connects software architecture design with agile practices
- **Netflix example in his blog picks up the refactoring theme:**
  - http://rhinoresearch.com/content/architecture-refactoring

Rhino Research

Software architecture training courses and consulting

Architecture Book | Training | Consulting | Architecture Topics | Blog | About

Architecture Refactoring

This example is best thought of as **architecture refactoring**. Each refactoring of the architecture was precipitated by a pressing failure risk. Object-level refactorings take a negligible amount of time and therefore need little justification, so you should just go ahead and rename that variable to be more expressive of its intent. An architecture refactoring is expensive, so it requires a significant risk to justify it.

Two important lessons are apparent. First, ***design does not exclusively happen up-front***. It is often reasonable to spend time up-front making the best choices you can, but it is optimistic to think you know enough to get all those design decisions right. You should anticipate spending time designing after your project's inception.

Second, ***failure risk can guide architecture refactoring***. By the time it is implemented, nearly every system is out of date compared to the best thinking of its developers. That is, some technical debt exists. Perhaps, in hindsight, you wish you had chosen a different architecture. Risks can help you decide how bad it will be if you keep your current architecture.

© Olaf Zimmermann, 2014.

Existing Work (3/3): Architectural Knowledge Management

- **IEC/IEEE/ISO 42010:2011 standard for architecture description**
  - Rationale as first class entity in architecture documentation

Reference: http://enterprise-strategy-architecture.blogspot.ch/2011/11/understanding-isoiecieee-420102011.html

- **Active research community investigating architectural decisions**
  - E.g. SOAD project: active, guiding role for recurring architectural decisions
- **See SEI SATURN 2013 BoD session report for state of the practice**

© Olaf Zimmermann, 2014.

AD Capture for Documentation Purposes

Reference: ABB, SATURN 2012

Unique identifier: AD-01, **Status:** approved, **Owner:** cholzim

Link to system concerns
See architectural principles stated on the enterprise level: IT security requirements

Rationale
In the context of the historian component, facing the data privacy requirements specified in the corporate security guidelines, we decided to encrypt the persistent storage to achieve confidentiality.

Constraints and assumptions:
Performance is good enough, certificate management can be handled.

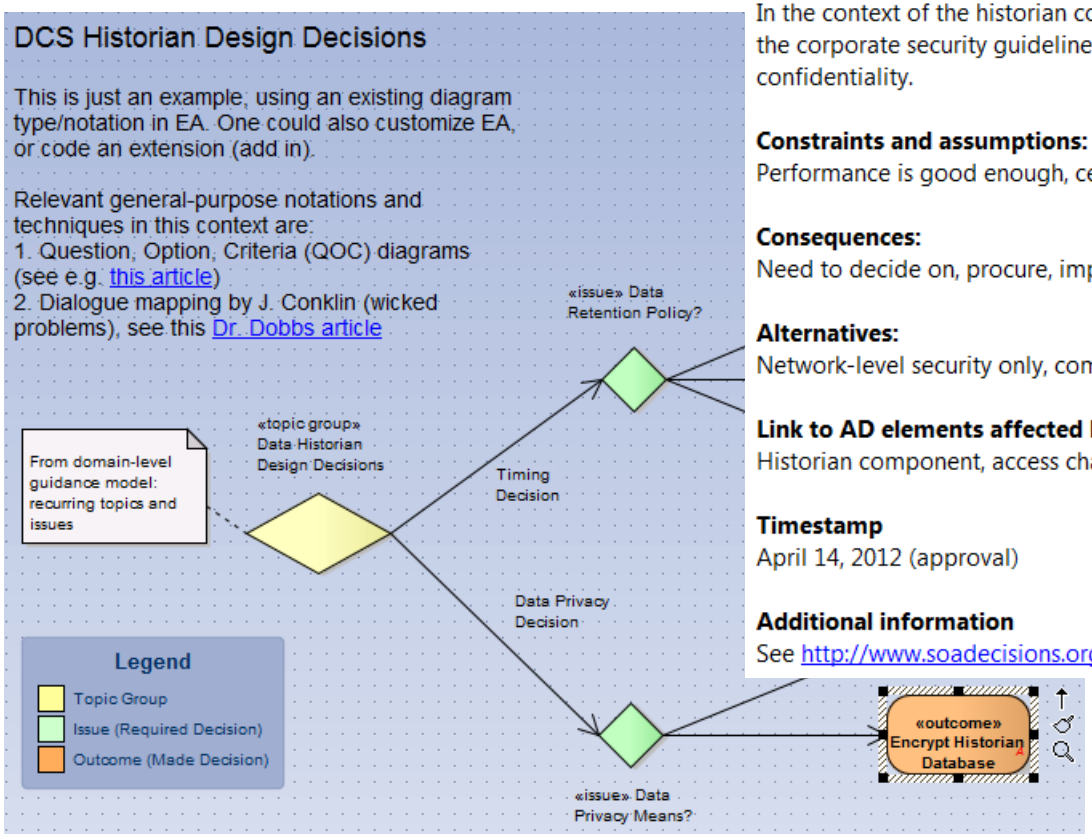
Consequences:
Need to decide on, procure, implement some crypto library and/or hardware (TBD)

Alternatives:
Network-level security only, combined with role-based access control

Link to AD elements affected by the decision:
Historian component, access channel, security subsystem

Timestamp
April 14, 2012 (approval)

Additional information
See http://www.soadecisions.org



© Olaf Zimmermann, 2014.

Y-Template

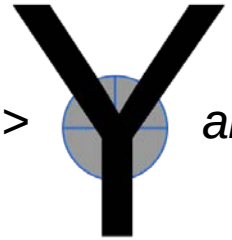
Reference: ABB, SATURN 2012

- **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>,

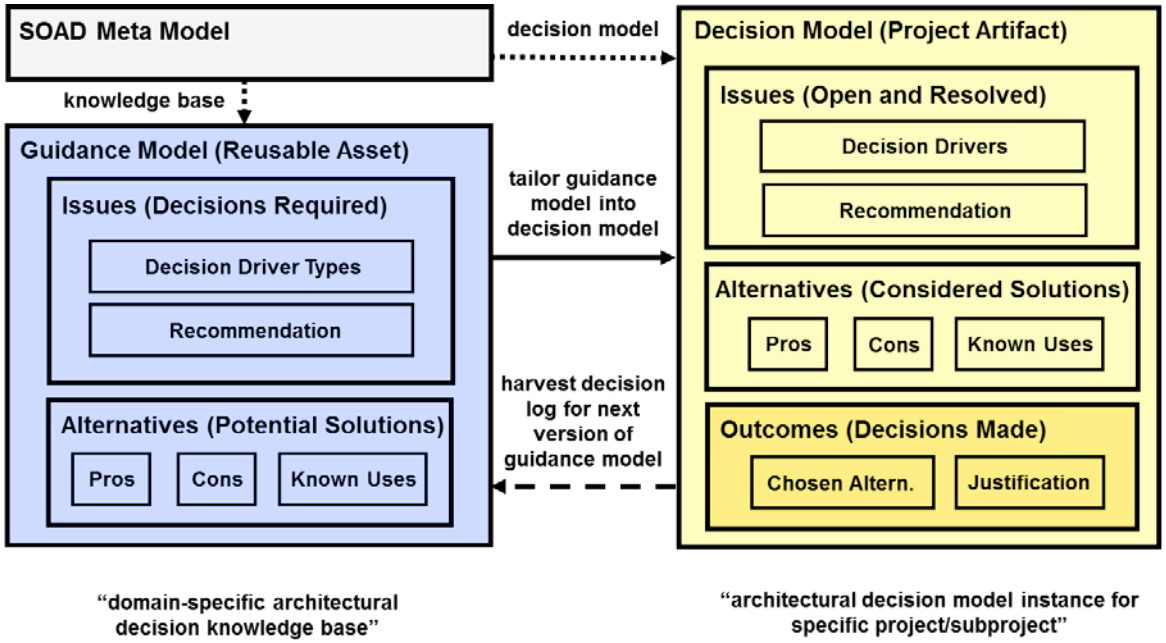
… to achieve <quality q>,

… accepting downside <consequence c>.

© Olaf Zimmermann, 2014.

SOA Decision Modeling (2006-2011): Generic Metamodel

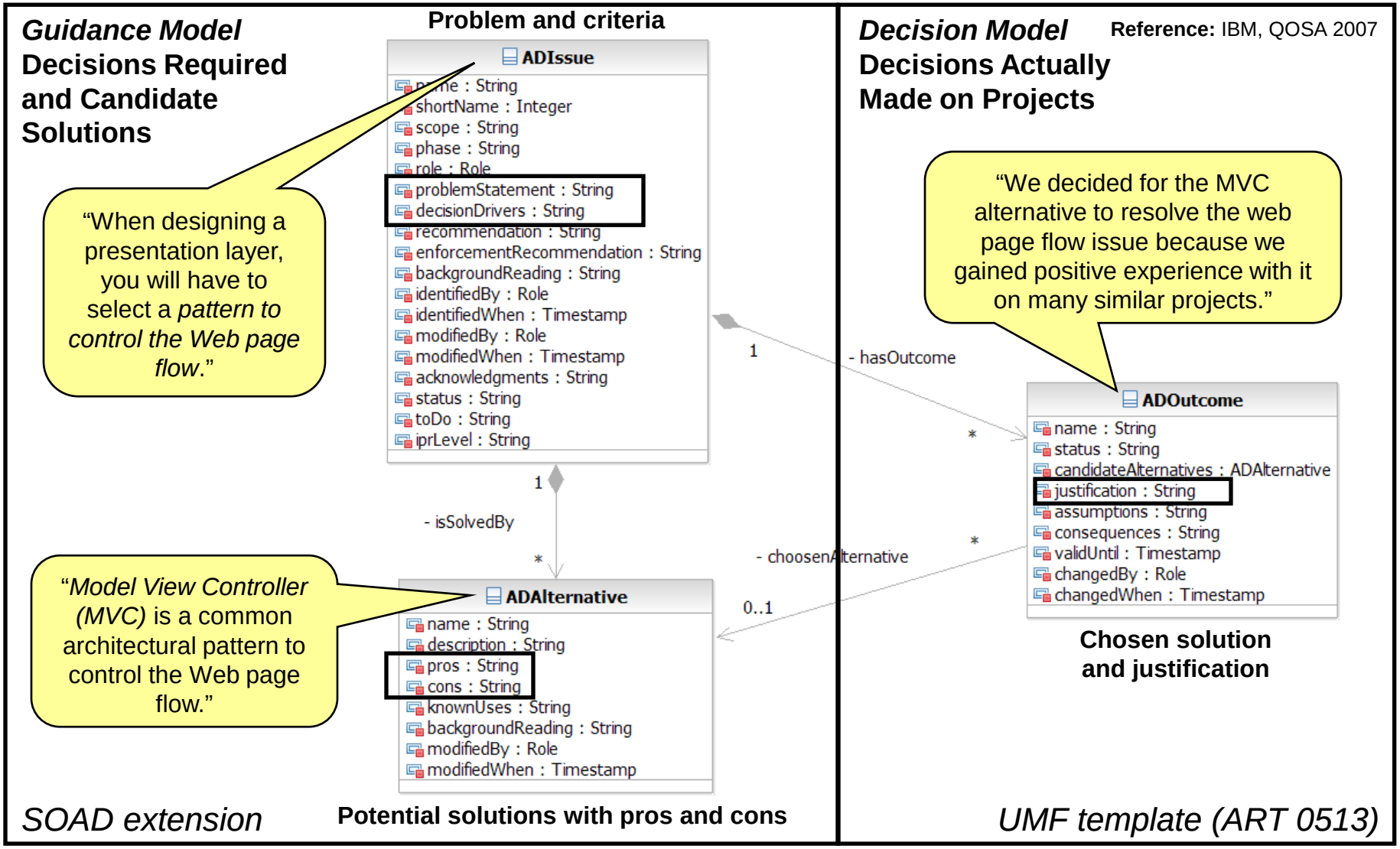
- **Existing metamodels and templates refactored and extended for reuse**
  - Before: documentation – after the fact (past tense)
  - With SOAD: design guidance – forward looking (future tense)



Reference: Architectural Decisions as Reusable Design Assets. IEEE Software 28(1): 64-69 (2011)

© Olaf Zimmermann, 2014.

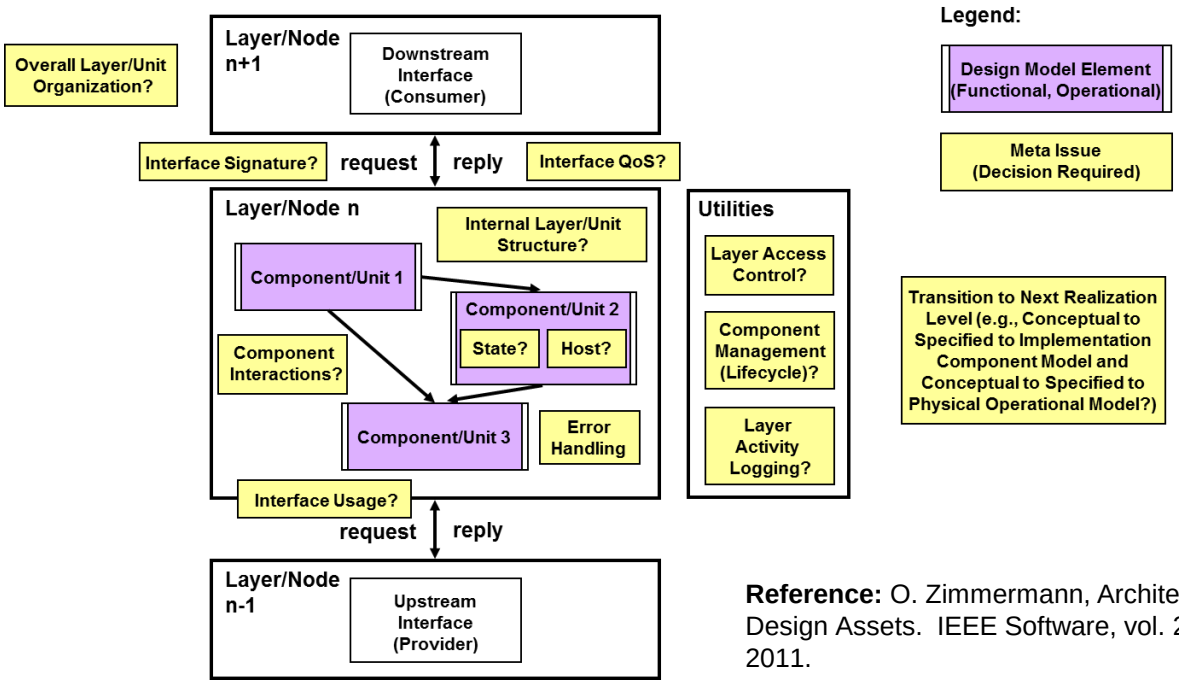
Entity Types and Associations in UML Metamodel



© Olaf Zimmermann, 2014.

SOAD Project (2006-2011): Issues Recurring in SOA Design

- **Patterns + recurring issues yield guidance models for a domain**
  - (Can be) applied to information system design and information integration
- **Issue catalog organized by layer/node type, by component/connector**



Reference: O. Zimmermann, Architectural Decisions as Reusable Design Assets. IEEE Software, vol. 28, no. 1, pp. 64-69, Jan./Feb. 2011.

© Olaf Zimmermann, 2014.

Agenda

1. Motivation
2. Existing work
3. **From code refactoring and architectural decisions to architectural refactoring**
4. Catalog structure and refactoring template
5. Some basic architectural refactorings
6. Tool support
7. Architectural Refactoring for Cloud (ARC)

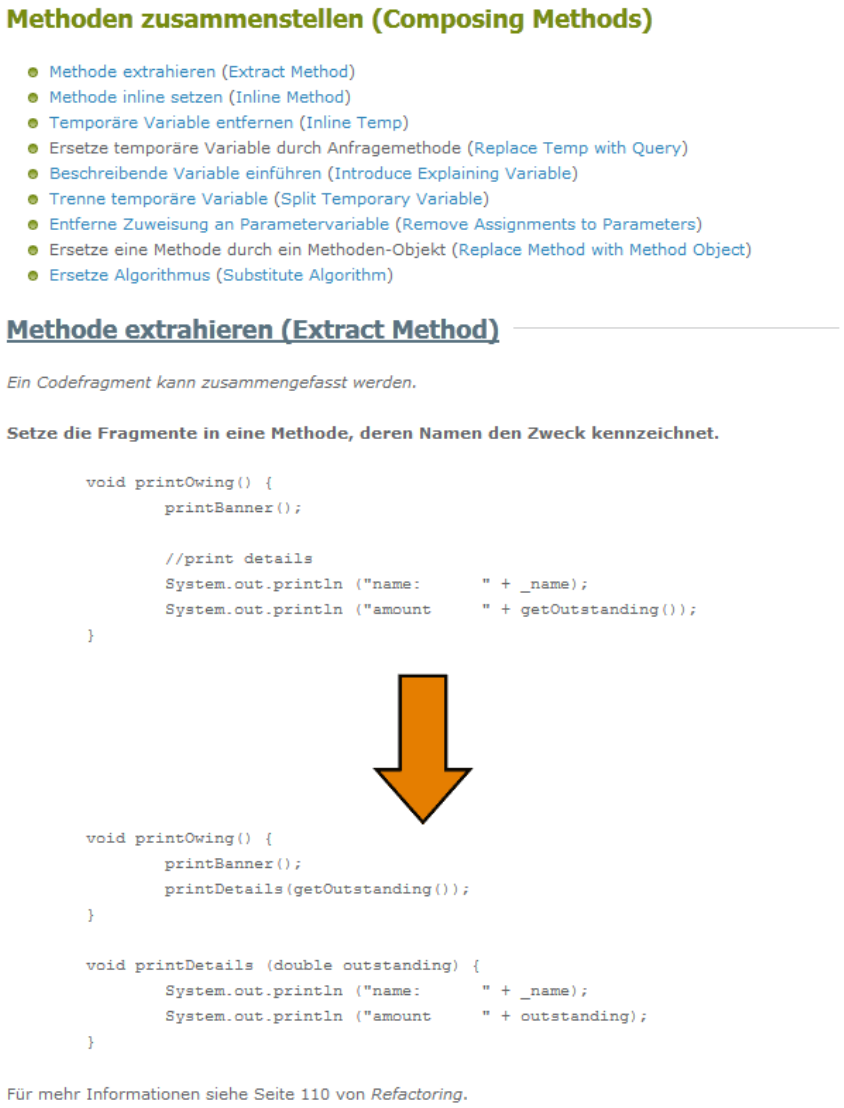
© Olaf Zimmermann, 2014.

What is Refactoring?

Reference: http://refactoring.com/

- **Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior**
  - Goal: improve a certain quality while preserving others
- **Code refactoring is commonly practiced – architectural refactoring is not (yet)**
  - System-level, not program level
  - Hard to formalize

Reference: http://www.tutego.de/java/refactoring/catalog/



Methoden zusammenstellen (Composing Methods)

- Methode extrahieren (Extract Method)
- Methode inline setzen (Inline Method)
- Temporäre Variable entfernen (Inline Temp)
- Ersetze temporäre Variable durch Anfragemethode (Replace Temp with Query)
- Beschreibende Variable einführen (Introduce Explaining Variable)
- Trenne temporäre Variable (Split Temporary Variable)
- Entferne Zuweisung an Parametervariable (Remove Assignments to Parameters)
- Ersetze eine Methode durch ein Methoden-Objekt (Replace Method with Method Object)
- Ersetze Algorithmus (Substitute Algorithm)

Methode extrahieren (Extract Method)

Ein Codefragment kann zusammengefasst werden.

Setze die Fragmente in eine Methode, deren Namen den Zweck kennzeichnet.

```
void printOwing() {
        printBanner();

        //print details
        System.out.println ("name:      " + _name);
        System.out.println ("amount     " + getOutstanding());
}
```

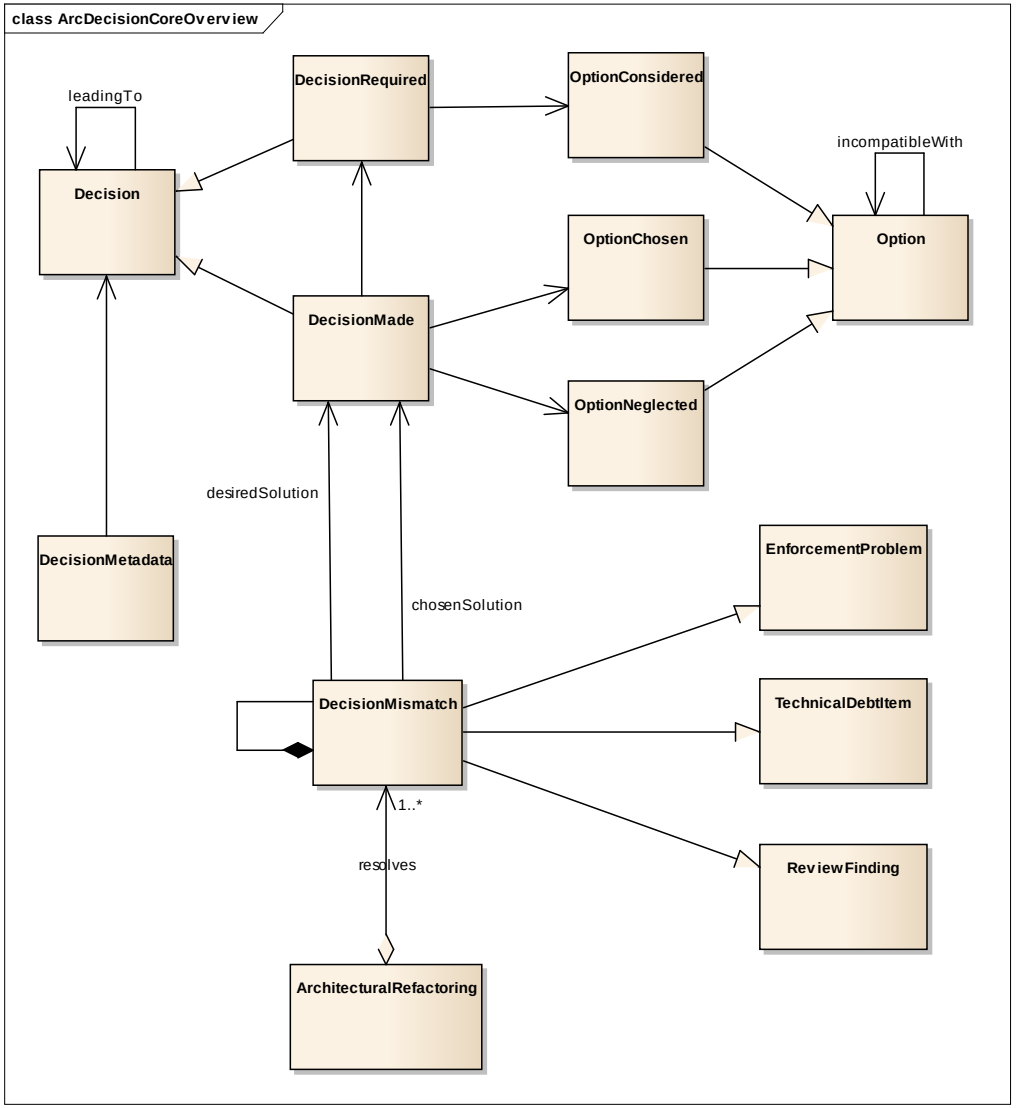
```
void printOwing() {
        printBanner();
        printDetails(getOutstanding());
}

void printDetails (double outstanding) {
        System.out.println ("name:      " + _name);
        System.out.println ("amount     " + outstanding);
}
```

Für mehr Informationen siehe Seite 110 von *Refactoring*.

© Olaf Zimmermann, 2014.

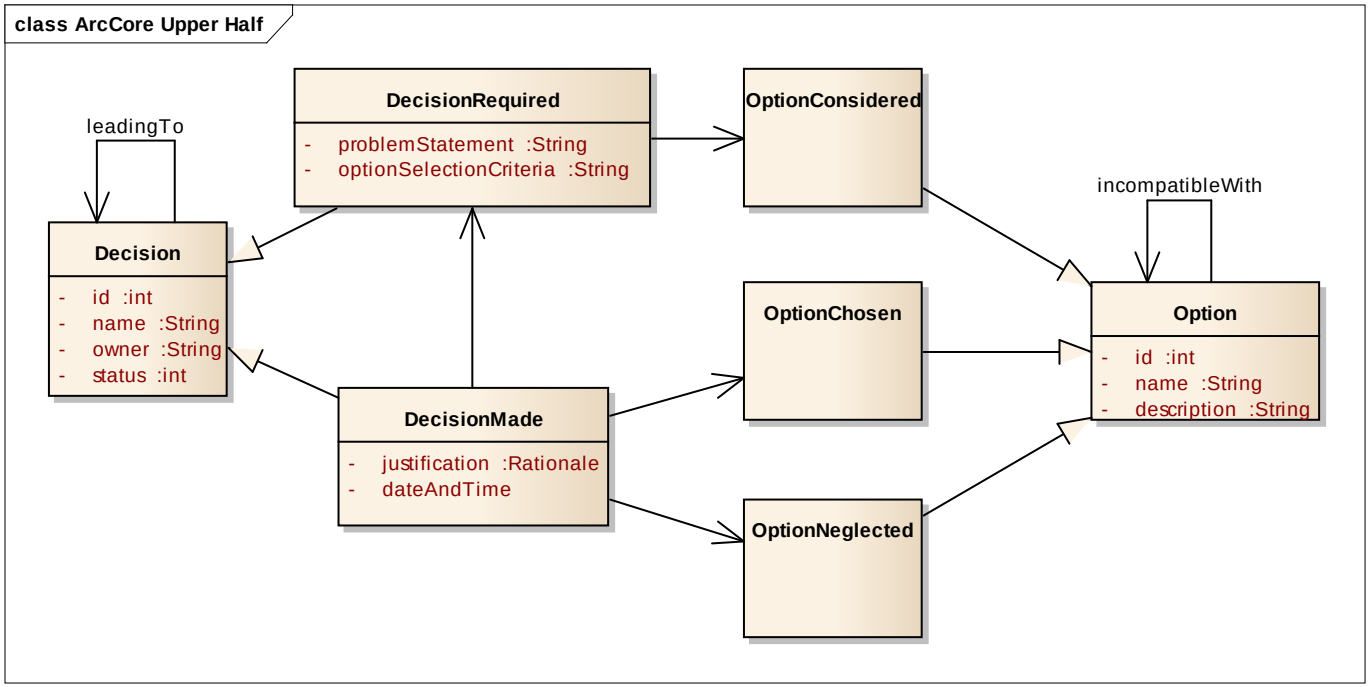
ARC Metamodel (at an Initial State of Elaboration)

- **Decisions required/made**
  - Problem
  - Candidate solutions
  - Chosen solution
- **Refactoring need arises from decision *mismatches***
  - Decision actually made, decision recommendation
  - Same problem (to be) solved differently
- **Mismatch resolution via *pairs of decisions made***
  - Set of undecide-decide activities (actions)
  - Plus decision execution



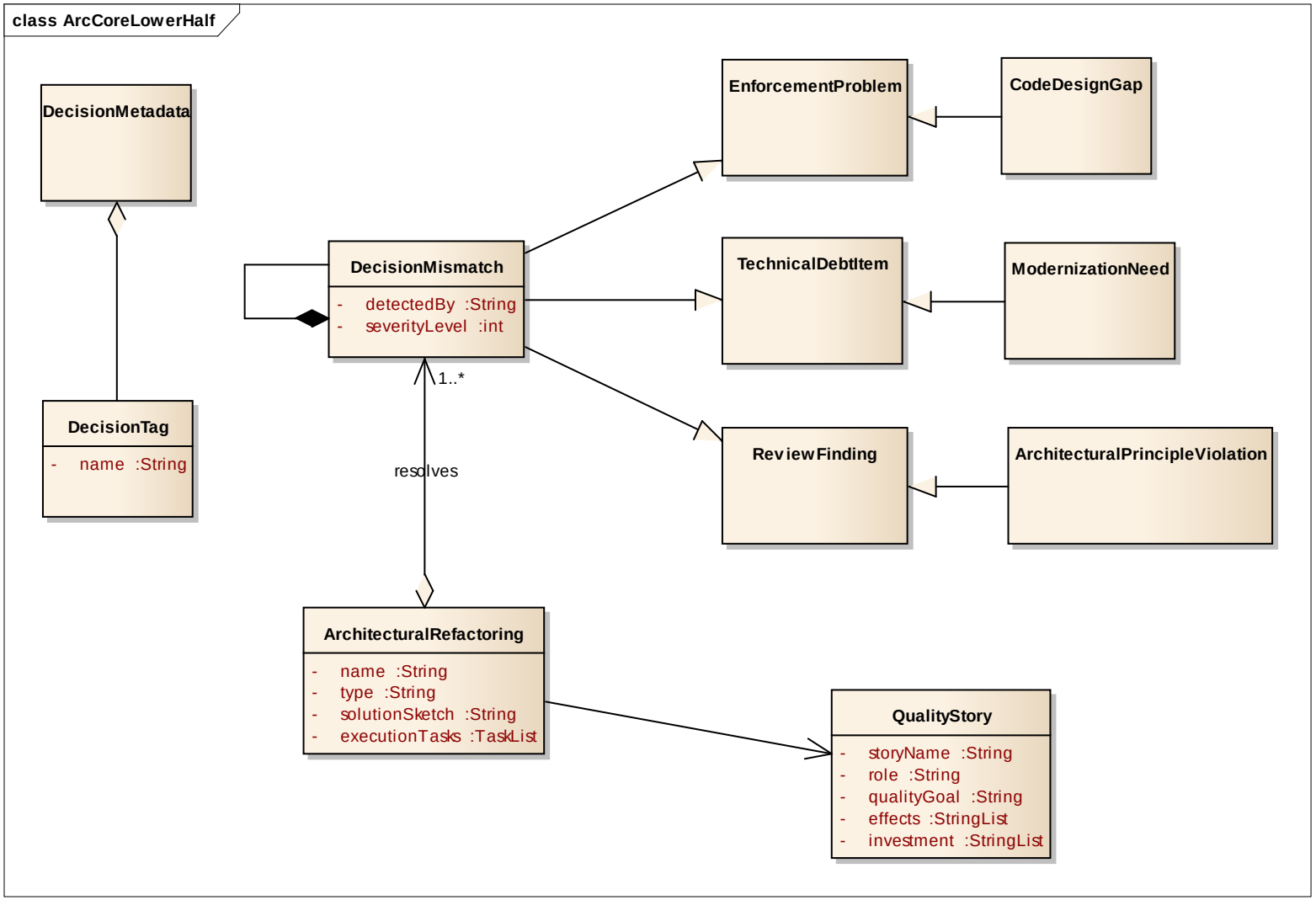
© Olaf Zimmermann, 2014.

ARC Metamodel (at an Initial State of Elaboration) (2/3)



© Olaf Zimmermann, 2014.

ARC Metamodel (at an Initial State of Elaboration) (3/3)



© Olaf Zimmermann, 2014.

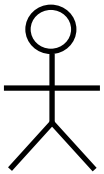
Agenda

1. Motivation
2. Existing work
3. From code refactoring and architectural decisions to architectural refactoring
4. **Refactoring templates and catalog structure**
5. Some basic architectural refactorings
6. Tool support
7. Architectural Refactoring for Cloud (ARC)

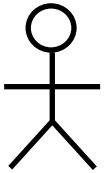
© Olaf Zimmermann, 2014.

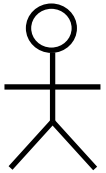
Quality Story Template (Inspired by User Stories)

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],**

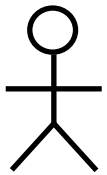
I would like to **[achieve quality goal A]**
– without changing the functional scope of the system –

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, 2014

Quality Story Example (Note: Inspired by User Stories)



DevOps Engineer

***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, 2014

Architectural Refactorings – Template

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, 2014.

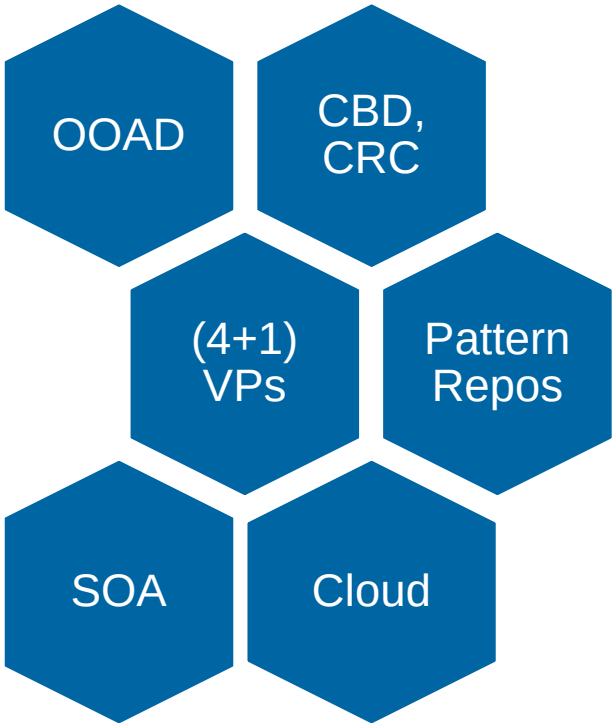
Architectural Refactoring – Example

***Architectural Refactoring:* De-SQL**	
Context (viewpoint, refinement level): • Logical viewpoint, data viewpoint (all levels)	*Quality attributes and stories (forces):* • Flexibility, data integrity
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) • Write new data access layer, implement SQLish query capabilities yourself • Decide on transaction boundaries (if any), document database administration (CRUD, backup)	

© Olaf Zimmermann, 2014.

Towards a Structure for Architectural Refactoring Catalogs

- **OOAD and CBD/CRC refactorings**
  - Component, responsibilities
  - Connector, collborations
  - Boundary classes, services
- **Viewpoints (VPs)**
  - 4+1 VP refactorings (as defined in RUP, OpenUP)
  - alt. Rozanski & Wood's Viewpoints and Perspectives structure
- **Domain-specific refactorings**
  - Service-Oriented Architecture (SOA)
  - Cloud Computing
  - … and many more



© Olaf Zimmermann, 2014.

Agenda

1. Motivation
2. Existing work
3. From code refactoring and architectural decisions to architectural refactoring
4. Catalog structure and refactoring template
5. **Some basic architectural refactorings**
6. Tool support
7. Architectural Refactoring for Cloud (ARC)

© Olaf Zimmermann, 2014.

Fundamental Refactorings (Logical Viewpoint)

- **Add Layer**
- **Collapse Layers (into Single One)**
- **Add Tier**
- **Collapse Tiers (into One)**
- **Split Component**
- **Merge Components**

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): • […]	

- **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

© Olaf Zimmermann, 2014.

Architectural Refactoring – Another Example

***Architectural Refactoring:* Move Responsibility**	
Context (viewpoint, refinement level): • Logical viewpoint, CRC card (see Appendix)	*Quality attributes and stories (forces):* • Cohesion and 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 CRC cards in word processor, drawing tool, documentation wiki • Edit operations in UML or ADL modeling tool • Updates to component realizations in code (note: architecturally evident coding style to be followed)	

© Olaf Zimmermann, 2014.

Fundamental Refactorings (Infrastructure 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, 2014.

Architectural Refactoring – A Third Example

***Architectural Refactoring:* Introduce Cache**	
Context (viewpoint, refinement level): • Logical, platform-specific refinements	*Quality attributes and stories (forces):* • 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	

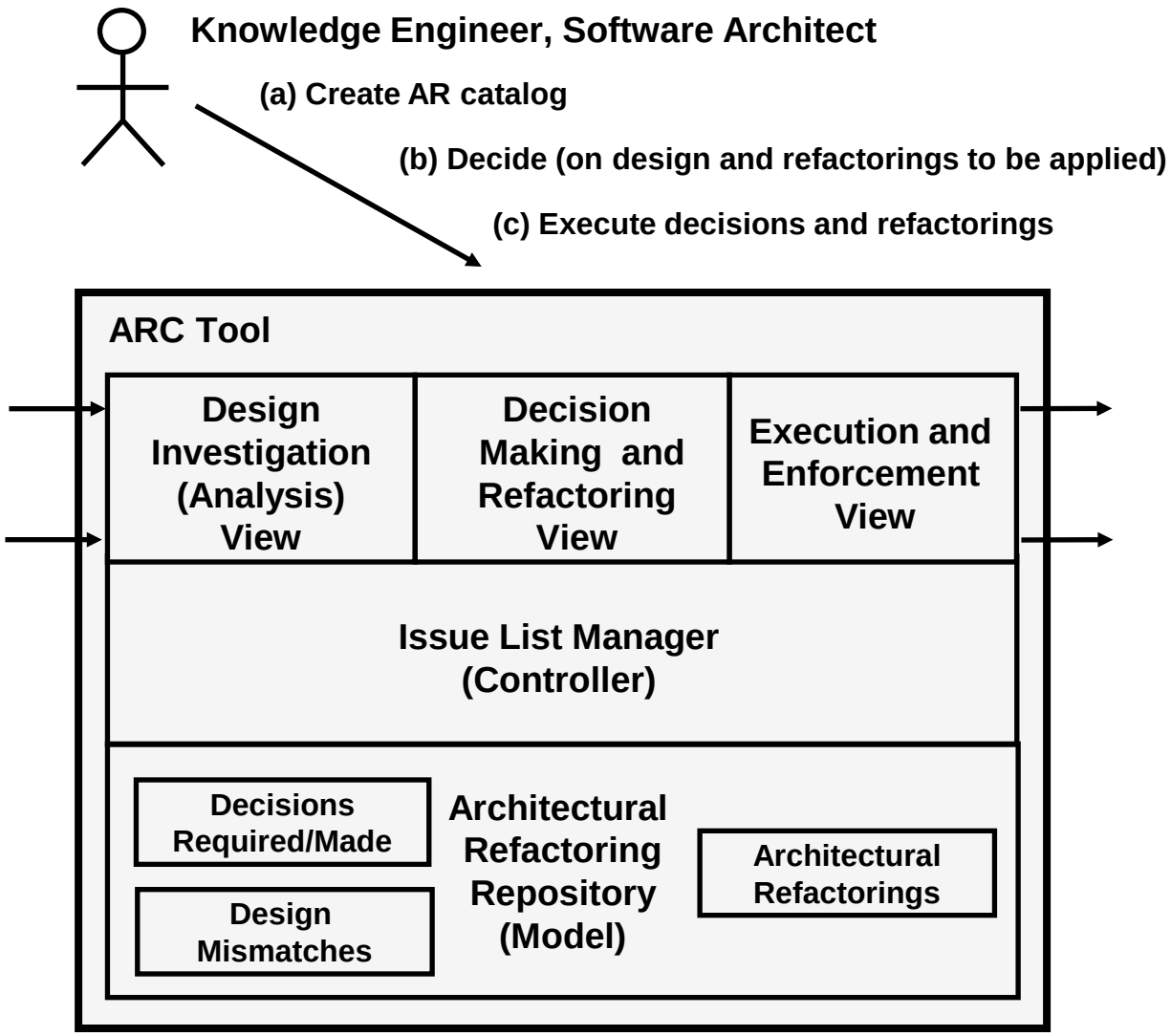
© Olaf Zimmermann, 2014.

Agenda

1. Motivation
2. Existing work
3. From code refactoring and architectural decisions to architectural refactoring
4. Catalog structure and refactoring template
5. Some basic architectural refactorings
6. **Tool support**
7. Architectural Refactoring for Cloud (ARC)

© Olaf Zimmermann, 2014.

Towards Tool Support for Architectural Refactoring



© Olaf Zimmermann, 2014.

Agenda

1. Motivation
2. Existing work
3. From code refactoring and architectural decisions to architectural refactoring
4. Catalog structure and refactoring template
5. Some basic architectural refactorings
6. Tool support
7. **Architectural Refactoring for Cloud (ARC)**

© Olaf Zimmermann, 2014.

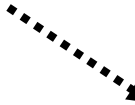
Problem Statement: Migrating to the Cloud is not Straightforward

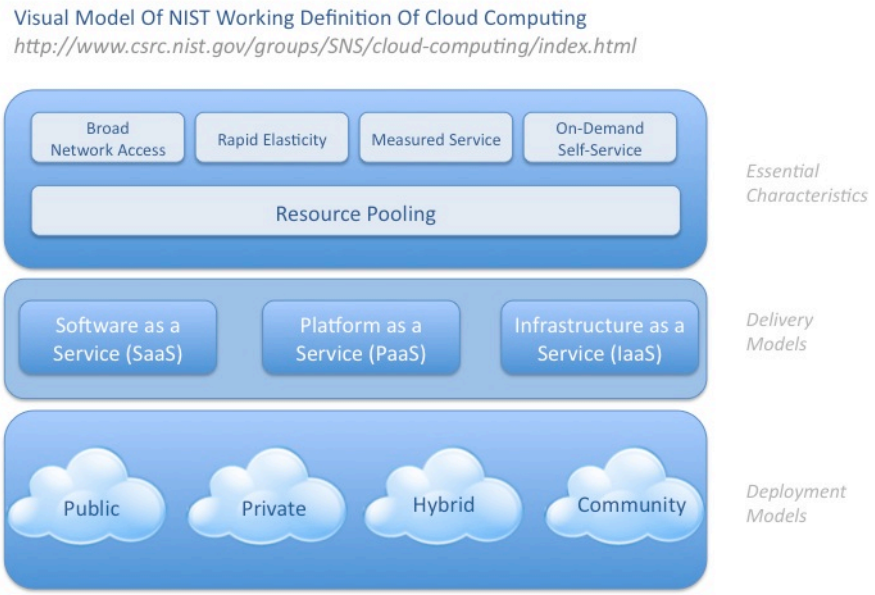
- **Cloud trend ubiquitous, but cloud not mainstream yet (inhibitors!)**
  - Here: PaaS, then IaaS, SaaS only to some extend relevant
- **Many new paradigms and principles**
  - Elasticity, rapid provisioning
  - Radical approach to troubleshooting (restart, throw away hardware)
  - Dynamic billing models (per CPU cycles used, per storage unit used)
- **Impact on (physical) operational modeling is obvious**
- **Impact on component modeling is less obvious, but does exist**
- **No standards, no design methods established yet**

- ***See separate OOP 2014 session***

© Olaf Zimmermann, 2014.

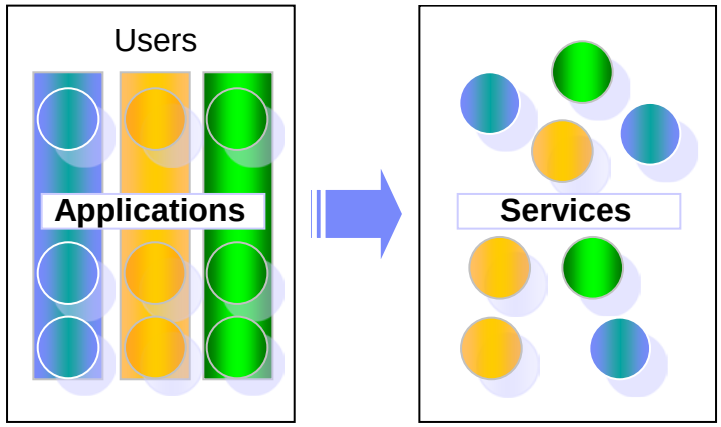
ARC: From Traditional Layer-Tier Architectures to Cloud Services





Traditional

Service-Oriented Architecture (SOA)



Discrete Applications (Two or Three Tiers)

Basket of Services

© Olaf Zimmermann, 2014.

Initial Ideas for Content of Arch. Ref. (AR) Catalog for Cloud

- **Change cloud computing pattern**
  - E.g. from server state to database 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, 2014.

Towards a Cloud Domain Refactoring Catalog (Preview)

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, 2014.

Summary and Discussion

- **Architectural decision making and architectural refactoring a key responsibilities of IT architects which are often underestimated and underrepresented in existing methods and tools.**
  - New task-centric templates: *quality story*, *architectural refactoring*
- **In cloud design and other domains, many decisions and refactorings recur. This makes it possible to reduce the documentation effort and to share architectural decision knowledge including best practices (design acceleration and quality assurance).**
  - Cloud refactoring catalog under construction
- **Tool support for decision modeling with reuse and for architectural refactoring is emerging (catalog management, execution planning)**
- **We would like to hear from you now…**
  - … are the presented scenarios, templates, and tool vision useful?
  - … would you have additional requirements, e.g. collaboration and integration needs?

© Olaf Zimmermann, 2014.

Additional References

- **Architectural Refactoring**
  - Hardly any peer-reviewed publications yet (one book chapter by M. Stal)
- **Architectural Decision (AD) Capturing and Reuse:**
  - J. Tyree/A. Akerman, Architecture Decisions: Demystifying Architecture. IEEE Software, 22/2, March/April 2005
  - O. Zimmermann, Architectural Decisions as Reusable Design Assets. IEEE Software, 28/1, Jan./Feb. 2011, http://soadecisions.org/soad.htm
  - Uwe Zdun, Rafael Capilla, Huy Tran, Olaf Zimmermann: Sustainable Architectural Design Decisions. IEEE Software, 30/6, Nov./Dez. 2013
- **Cloud Reengineering Knowledge**
  - IAAS (University of Stuttgart), http://www.cloud-data-migration.com/
  - T. Höllwarth, Cloud Migration, http://www.cloud-migration.eu/
  - CBDI-SAE, Cloud Migration Patterns, http://everware-cbdi.com/index.php?cID=pattern-index&tab=520

© Olaf Zimmermann, 2014.

Institut für Software

OOP 2014
SOFTWARE MEETS BUSINESS

HINTERGRUNDINFORMATIONEN ZU CRC-KARTEN UND ARCHITEKTURENTSCHEIDUNGEN

IFS INSTITUTE FOR SOFTWARE

Prof. Dr. Olaf Zimmermann
Distinguished (Chief/Lead) IT Architect, The Open Group
ozimmerm@hsr.ch

München, 5. Februar 2014

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Components, Responsibilities, Collaborations (CRC) Card

Component: **[Name of Component]**	
Responsibilities: • [What is this component capable of doing (services provided)?] • [Which data does it deal with?] • [How does it do this in terms of key system qualities?]	*Collaborations (Interfaces):* • [Who invokes this component (service consumers)?] • [Whom does this component call to fulfill its responsibilities (service providers)?] • [Any external connections (both active and passive)?]
Known uses (implementations): • [Which technologies, products (commercial, open source) and internal assets realize the outlined component functionality (responsibilities)?]	

- **Traditional usage of the format (in software architecture context):**
  - http://stal.blogspot.ch/2006/12/architects-toolset-crc-cards.html

© Olaf Zimmermann, 2014.

Recurring Issues (1/2)

Artifact	Decision Topic	Recurring Issues (Decisions Required)
Enterprise architecture documentation	IT strategy	Buy vs. build strategy, open source policy
	Governance	Methods (processes, notations), tools, reference architectures, coding guidelines, naming standards, asset ownership
System context	Project scope	External interfaces, incoming and outgoing calls (protocols, formats, identifiers), service level agreements, billing
Other viewpoints	Development process	Configuration management, test cases, build/test/production environment staging
	Physical tiers	Locations, security zones, nodes, load balancing, failover, storage placement
	Data management	Data model reach (enterprise-wide?), synchronization/replication, backup strategy
Architecture overview diagram	Logical layers	Coupling and cohesion principles, functional decomposition (partitioning)
	Physical tiers	Locations, security zones, nodes, load balancing, failover, storage placement
	Data management	Data model reach (enterprise-wide?), synchronization/replication, backup strategy
Architecture overview diagram	Presentation layer	Rich vs. thin client, multi-channel design, client conversations, session management
	Domain layer (process control flow)	How to ensure process and resource integrity, business and system transactionality
	Domain layer (remote interfaces)	Remote contract design (interfaces, protocols, formats, timeout management)
	Domain layer (component-based development)	Interface contract language, parameter validation, Application Programming Interface (API) design, domain model
	Resource (data) access layer	Connection pooling, concurrency (auto commit?), information integration, caching
	Integration	Hub-and-spoke vs. direct, synchrony, message queuing, data formats, registration

Source: O. Zimmermann, Architectural Decision Identification in Architectural Patterns. WICSA/ECSA Companion Volume 2012, Pages 96-103.

© Olaf Zimmermann, 2014.

Recurring Issues (2/2)

Artifact	Decision Topic	Recurring Issues (Decisions Required)
Logical component	Security	Authentication, authorization, confidentiality, integrity, non-repudiation, tenancy
	Systems management	Fault, configuration, accounting, performance, and security management
	Lifecycle management	Lookup, creation, static vs. dynamic activation, instance pooling, housekeeping
	Logging	Log source and sink, protocol, format, level of detail (verbosity levels)
	Error handling	Error logging, reporting, propagation, display, analysis, recovery
Components and connectors	Implementation technology	Technology standard version and profile to use, deployment descriptor settings (QoS)
	Deployment	Collocation, standalone vs. clustered
Physical node	Capacity planning	Hardware and software sizing, topologies
	Systems management	Monitoring concept, backup procedures, update management, disaster recovery

Source: O. Zimmermann, Architectural Decision Identification in Architectural Patterns. WICSA/ECSA Companion Volume 2012, Pages 96-103.

© Olaf Zimmermann, 2014.

Good and Bad Justifications, Part 1

Decision driver type	Valid justification	Counter example
Wants and needs of external stakeholders	Alternative A best meets user expectations and functional requirements as documented in user stories, use cases, and business process model.	End users want it, but no evidence for a pressing business need. Technical project team never challenged the need for this feature. Technical design is prescribed in the requirements documents.
Architecturally significant requirements	Nonfunctional requirement XYZ has higher weight than any other requirement and must be addressed; only alternative A meets it.	Do not have any strong requirements that would favor one of the design options, but alternative B is the market trend. Using it will reflect well on the team.
Conflicting decision drivers and alternatives	Performed a trade-off analysis, and alternative A scored best. Prototype showed that it's good enough to solve the given design problem and has acceptable negative consequences.	Only had time to review two design options and did not conduct any hands-on experiments. Alternative B does not seem to perform well, according to information online. Let's try alternative A.

Source: Zimmermann O., Schuster N., Eeles P., Modeling and Sharing Architectural Decisions, Part 1: Concepts. IBM developerWorks, 2008

© Olaf Zimmermann, 2014.

Good and Bad Justifications, Part 2

Decision driver type	Valid justification	Counter example
Reuse of an earlier design	Facing the same or very similar NFRs as successfully completed project XYZ. Alternative A worked well there. A reusable asset of high quality is available to the team.	We've always done it like that. Everybody seems to go this way these days; there's a lot of momentum for this technology.
Prefer do-it-yourself over commercial off-the-shelf (build over buy)	Two cornerstones of our IT strategy are to differentiate ourselves in selected application areas, and remain master of our destiny by avoiding vendor lock-in. None of the evaluated software both meets our functional requirements and fits into our application landscape. We analyzed customization and maintenance efforts and concluded that related cost will be in the same range as custom development.	Price of software package seems high, though we did not investigate total cost of ownership (TCO) in detail. Prefer to build our own middleware so we can use our existing application development resources.
Anticipation of future needs	Change case XYZ describes a feature we don't need in the first release but is in plan for next release. Predict that concurrent requests will be x per second shortly after global rollout of the solution, planned for Q1/2009.	Have to be ready for any future change in technology standards and in data models. All quality attributes matter, and quality attribute XYZ is always the most important for any software-intensive system.

Source: Zimmermann O., Schuster N., Eeles P., Modeling and Sharing Architectural Decisions, Part 1: Concepts. IBM developerWorks, 2008

© Olaf Zimmermann, 2014.

Modeling Principles and Practices (1/2)

1. **Q+A principle:** *Characterize problems in question form to whet consumers' appetite (and don't forget to answer the question in the solution description).*
2. **Diversity principle:** *Explain solution both to senior and to junior audiences to ensure broad applicability.*
3. **Decisiveness principle:** *Be assertive to make the given advice (guidance) actionable.*
4. **Objectivity principle:** *Separate facts from opinions to ensure accuracy and acceptance.*
5. **Annotation and structuring principle:** *Provide context information and organize the model coherently to support decision ordering and clustering.*
6. **Mentoring and management principle:** *Facilitate knowledge exchange to support collaboration between knowledge engineers and to stimulate discussions between decision makers.*

© Olaf Zimmermann, 2014.

Modeling Principles and Practices (2/2)

7. **Provenance principle:** *Acknowledge sources and status of knowledge to ensure authenticity and actuality.*
8. **Consistency principle:** *Establish naming conventions and structuring heuristics to ensure readability (model orientation) and repeatability (during model updates).*
9. **Rigor and accuracy principle:** *Perform peer reviews and professional editing steps (e.g., copy editing) to achieve publication quality.*
10. **Dependency management principle:** *Be careful and diligent when modeling and modifying a network of decision points and options.*
11. **Rationale principle:** *As a decision maker, justify decisions properly.*

Reference: O. Zimmermann, C. Miksovic, J. Küster, Reference Architecture, Metamodel and Modeling Principles for Architectural Knowledge Management in Information Technology Services. Journal of Systems and Software, Elsevier. Volume 85, Issue 9, Pages 2014-2033, Sept. 2012.

© Olaf Zimmermann, 2014.