A Pattern Language for a Semantics-Driven Software Architecture

Stefan Holtel
Vodafone Group R&D .DE
Chiemgaustraße 116
81549 Munich
Germany
+49 89 9 54 10-0
stefan.holtel@vodafone.com

6th June 2007

"We only paint the panorama picture from what is happening anyway",
Manuel De Landa

Abstract

Domain-Specific Software Architectures (DSSA) have been proposed as a way to build service development platforms better fitting in with developer needs. Thus, Architecture Description Languages (ADL) became an approach to create technology-independent software architectures. Numerous ADLs like ACME, AESOP, or Rapide were specified to address special features of a problem domain or a product line. But they still reflect a technical approach. Their grammar is based on technical concepts like 'packaging', 'interfaces', or 'components'. Software architectures build up from ADLs still foster a technical mind-set. Lately, semantic approaches emerged in different ICT realms, namely the "semantic web" or "semantic desktops". It is most likely to even see a rise of semantic concepts in future software architectures. We therefore suggest a pattern language to create Semantics-Driven Software Architectures (SDSA), which are driven by the semantic concepts of a problem - not by technology to resolve them. Such an SDSA might serve as a blueprint for a post-technology era of software architecture platforms (SDP).

Keywords

Domain-Specific Software Architecture (DSSA), Architecture Description Language (ADL), Semantics-Driven Software Architecture (SDSA), Pattern Languages

Copyright © 2007, Vodafone. Other than as permitted by law, no part of this document may be reproduced, adapted, or distributed, in any form or by any means, without the prior written consent of Vodafone Group Plc. Permission is granted to copy for the EuroPloP 2007 conference. All other rights reserved.

1. Introduction

"The Map is not the Territory.",
Alfred Krzybski

A Domain-Specific Software Architecture (DSSA) is defined as

> "an assemblage of software components, specialized for a particular type of task (domain), generalized for the effective use across that domain, composed in a standardized structure (topology) effective for building successful applications". (Hayes-Roth 1994)

Another definition (Hayes-Roth and Tracz 1994) claims, that DSSA is

> "a context for patterns of problem elements, solution elements, and situations that define mapping between them."

Both definitions utilize abstractions of technical terms like "components", "standardized structure" or "elements". They miss the point of focusing on the expert language of a problem domain. A DSSA does not consider the wider context of a possible problem domain, it still stems from a technical understanding of service specification – and will motivate technology-driven implementations.

Although no standards exist (may be UML a minor exception), numerous Architecture Descriptoin Languages (ADL) have been provided to create DSSA blueprints. They are implemented with a variety of guiding principles and emphasis on different styles, e.g.

- Aesop deals with the specification and analysis of architectural styles (Garlan, Allan et al. 1994)
- Rapide focuses on specifying component interfaces and component interaction (Luckham and al. 1995)
- Wright is based on communication protocols (Garlan and Allen 1994)
- MetaH was created for the real-time domain like avionics (Vestal 1996)
- UniCon addresses packaging and functional issues (Shaw and al. 1995)

Additional languages address further specifics (Medvidovic and Taylor 2000), but all lack in making semantics of a problem domain a relevant element of their design. All ADLs can be considered as double-edged: they reflect knowledge which was abstracted from the problem domain (e.g. MetaH focusing on real-time problems in avionics or Rapide outlining interface- and interaction issues), but they do not present the knowledge domain on a deeper level of understanding.

2. From Domain-Specific to Semantics-Driven Software Architectures

"We become what we behold. We shape our tools and then our tools shape us.",
Marshall McLuhan

2.1. The Missing Link of Domain-Specific Software Architectures

The general idea for specifying a Semantics-Driven Software Architecture (SDSA) is based on the idea of push back technical thinking when building a software architecture. We envision a software architecture blueprint, which emerges from principle of a given problem domain: The wise selection of only a couple of relevant semantic modules (which fundamentally reflect a problem domain) should become the foundation of a software architecture.

The availability of an SDSA-based software architecture would help us to get new meaning and insight in the very nature of service creation and the necessary development processes. SDSA could become an early sign for an ICT industry undergoing a transformation of seismic proportion from technology- to user-centric development. Today, we still miss the tools, processes, and methods respectively.

SDSA is even poised for managing classes of services which today might be difficult to to predict precisely. Services might be unavailable from a technical point for a while and a practical point of view, but they should be envisoned early on a platform which should best serve as a stable and well-understood tool for a couple of years (take e.g. telepathic interfaces or neuro-implants).

But changes for software architectures in this regard could not come incremental: a shift from DSSA to SDSA requires a phase shift that appears problematic and challenging. Once made, it might reveal a new simplicity without the formulaic solutions of to much technology thinking in software architectures today.

Model-Driven Architecture (MDA) e.g. is poised to leverage DSSA and can be seen as a typical standard approach in this regard. MDA claims to split abstraction layers which help for reusing models. That reuse might be easy, if considered from a developer’s point of view. However, it is rather heavier to reuse, when you want to start from an appropriate understanding of the problem domain and want to utilize the MDA for different purposes.

2.2. How to lose Semantics: “Take a seat …”

Take the following example from Tracz (Tracz 1995): He defines three different problem domains: theater, airline, and libraries. Any domain comprises specific notions, terms, and definitions, which reflect a deep understanding of the knowledge domain respectively. But following the reference design-principle of DSSA, Tracz identified generalized notions (see last column in Table 1). Thus, e.g. “seat”, ”seat” and ”book” become “item”.

From a DSSA understanding, this was abstraction for a reference architecture from three separated knowledge domains (Tracz 1995). Unfortunately, additional information of the knowledge domain, which resides in the given notions of “seat” and “book”, unnoticeably vanishes and is not available any more.

Theater Domain	**Airline Domain**	**Library Domain**	**Inventory Generalization**
Seat	Seat	Book	Item
Row	Row	Shelf	Room/Shelf/Bin
Section	Ticket Category	Section	Aisle or Building
Performacne	Flight Number	Title	Description

Seating Arrangement	Seating Arrangement	Floor Plan	Warehouse
Tickets Sold	Tickets Sold	Books on loan	Items sold
Tickets Remaining	Tickets Remaining	Books available	Current inventory
Price	Price	Penality for lateness	Cost/Item
Performance time	Flight Departure	n/a	n/a
Performance date	Flight Date	Due date	Expiration date?
Ticket Agent	Ticket Agent	Librarian	Clerk

Table 1: Comparison of Theater, Airline, Library, and Inventory Domain (Tracz 1995)

"Seat" and "book" reflect similar concepts with regard to their respective domains. A DSSA will generalize those terms with "item". But what is meant by "item" in a performance context (chair in hall), in a transportation context (airplane seat), and in a library context (a book, somehow a kind of "seat")? It would be hard to derive the fitting definition of "item" backwards without additional information. The conclusion for typical DSSA: The more complex a knowledge domain from its very nature, the more challenging it will become to build a software architecture which really reflects different problem domains without loosing important information.

The challenge of generalization is: information about a domain vanishes slowly and unnoticed. It will complicated for non-experts to get a clear understanding about underlying concepts. A DSSA drops terms and notions ahead a technical dimension – not a semantic one. A DSSA is designed around technical concepts like 'interface', 'component', or 'workflow'. Those concepts are often not helpful to understand a complex problem domain.

Why is this fact such important to be considered? From psychology research we know that each single word is strongly related to the personal and cultural background of a person. Any single world can create confusion and misinterpretation (Farb 1993). Sometimes there is even no chance to directly translate a word between two different languages: One word in one language simply misses a similar or equal concept in another language. Requirements engineering often deals with these and similar problems (Gause and Weinberg 1989).

2.3. The New Era of Semantics

We call those service architectures Semantics-Driven We recognized, that semantics are becoming a basic concept in numerous realms of IT. Tim Berners-Lee created some groundbreaking papers on the future of the internet, which he claimed would become a "semantic web" (Berners-Lee 1990; Berners-Lee, Hendler et al. 2001). Real-life implementations are going to debut on the market in the near future (Spivack 2006). Several teams are currently working on the idea of assembling semantic desktops like Nepomuk[1], Gnowsis[2], or Beagle++ (Chirita, Costache et al. 2005), which will bring semantics ideas to the everyday desktop workplace.

A possible way to overcome those and other limitations of DSSA might be, to create software architectures which would offer service models from the scratch, reflecting the archetypical flows of

---

[1] http://nepomuk.semanticdesktop.org/xwiki/

[2] http://www.gnowsis.org/

information and communication in a problem-domain vocabulary, and making the explanation of the wider context of a service an integral explained part of the software architecture.

Software Architectures (SDSA). It seems to be the right time to reflect on the impact of semantics for future software architectures. In analogy to Boehm (Boehm 1994), who coined the “iconic turn” for the usage of media, we claim, that development of software architectures needs a “semantic turn”. We state, that a “semantic turn” of dealing with services will replace the former principle of exploiting technologies as its best. New and promising software services will less and less dependent on technical enablers. Instead, the groundbreaking idea behind a service will become paramount. It will be the crucial factor of success or failure. Technical problems will diminish in the future, usability topics will emerge as the top challenge of creating valuable services.

Domain-Specific Software Architecture	**Semantics-Driven Software Architecture**
is dealing with abstractions of technical concepts	Takes into account notions, concepts, and terms given by the problem domain
Wants to be unambiguous	Defines contradictions and ambiguitiy as a valuable and important design principle which must be reflected by the architecture
Usually stucks in a bias on e.g. device form factors or interaction paradigms	Is not limited to prerequisites, hidden agendas, or assumptions about possible technical implementations
Just “refers” to an unambigious terminology of the problem domain	Makes the “usage” of the problem domain terminology a central feature of the software architecture
Tries to generalize for the purpose of a reference design	Sticks to a specification as far as it better reflects the understanding of the given problem domain (no generalization, if clarity lacks)

Table 2: Comparing Domain-Specific and Semantics-Driven Software Architectures

Web 2.0 lately presented the idea, that there are more than technologies like AJAX, RSS, or microformats necessary to create new business value not yet seen before (O'Reilly 2004).

A software development platform based on a technology-agnostic approach would make it easier to address the paramount goal of the „disappearance of technology” posted by Marc Weiser more than a decade ago (Weiser 1991; Weiser 1994). From a software design point of view, his fundamental idea address important usability and design issues like outline e.g. by Donald Norman (Norman 1998).

2.4. The Workflow of Software Service Creation

We can distinguish three specific realms within an archetypical software development process to build a service upon a software architecture:

- Start with a general vision for a service: Is there an exciting idea for a service which can be outlined (beyond one or more technical enablers)?

- Specify the service idea: outlining the service itsself and adding a business model and other organizational requirements; understand the broader environment of a service, which will have any impact of its introduction, acceptance, usage, etc.

- Implement the service: Specific description of the service, which will become the blueprint of its technical implementation

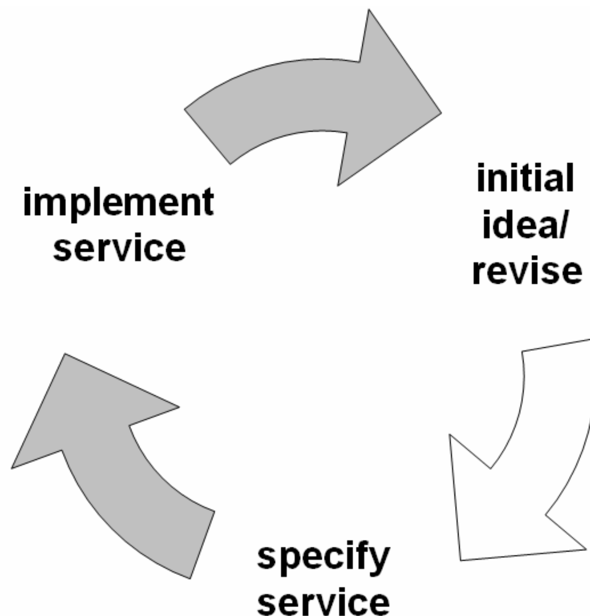

Figure 1: Archetypical Steps of Service Development

An SDSA should leverage this basic workflow. An SDSA pattern language should reflect the rules of thumb within any of these stages and could become a guiding principle to create numerous implementations of such a software architecture platform.

3. Creating an SDSA Pattern Language

“Tools lack ethics”,
Paul Saffo

A blueprint for an SDSA should avoid to address technical requirements for a software development platform. It should abstain from building remedies for technical problems (e.g. relevant interfaces, sufficient APIs, performance and scalability issues). Instead, it should deliver a guiding principle on how to create a software architecture which reflects the archetypical process of outlining a service idea from the scratch.

Architect Christopher Alexander in the 1970s created a collection of pattern for architectural purposes (Alexander, Ishikawa et al. 1977; Alexander 1979). Ken Beck and Ward Cunningham took those ideas from Alexander in 1987 and wrote design pattern for the creation of graphical user interfaces in Smalltalk3. Finally, Eric Gamma carried over the methodology and provided a general pattern methodology for software engineering (Gamma 1995).

[3] http://c2.com/ppr/best.html

With an SDSA pattern language under construction, we see the pattern methodology serves as integrating tool to match semantic concepts with software engineering concepts.

Based on our findings in relevant models and methodologies from different disciplines like psychology, sociology, biology, etc., we will present an SDSA pattern language, that should present a guiding principle of understanding the design of software architectures not starting from a technical point of view. Rather, SDSA is based on the idea, that technical metaphors, analogies and experiences (which usually drive design of a traditional DSDA) is not a sufficient guiding principle for the future.

The creation of a software architecture should instead totally reflect the understanding of the problem domain. Thus technical concepts like workflow, interfaces, or components etc. should not be the guiding principles for an SDSA pattern language. It should rather be driven and based on models which interpret and explain human activities and motives for their requirement for e.g. communication, information gathering, etc.

The workload of building an SDSA pattern language can be considered as identifying patterns for three generic areas:

1. One pattern domain (SERVICE FRAMEWORK) deals with how to build a general framework (e.g. clarify terms, notions, and concepts of the problem domain; get a technology-independent service vision, etc.). Possible methods in use for this purpose could be templates, checklists, taxonomies, or glossaries.

2. Another pattern domain (SERVICE CONTEXT) deals with how to create, explain, and document a service's context (e.g. the business modeling, clarifying stakeholder intentions and motives, etc.). Possible basic methodologies supporting this domain are use case modeling (Schneider and Winters 1998; Cockburn 2000; Schneider and Winters 2001), and especially essential use cases, utilized on a business workflow level (Constantine and Lockwood 1999).

3. The last pattern domain (SERVICE SPECIFICATION) deals with how to specify the generic service in a technology-agnostic manner (e.g. general communication model instead of an inseparable mix of communication types and formats, separating content from service flow, etc.). Typical methods introduced here again are use case modeling, but with a focus on possible service implementions in mind.

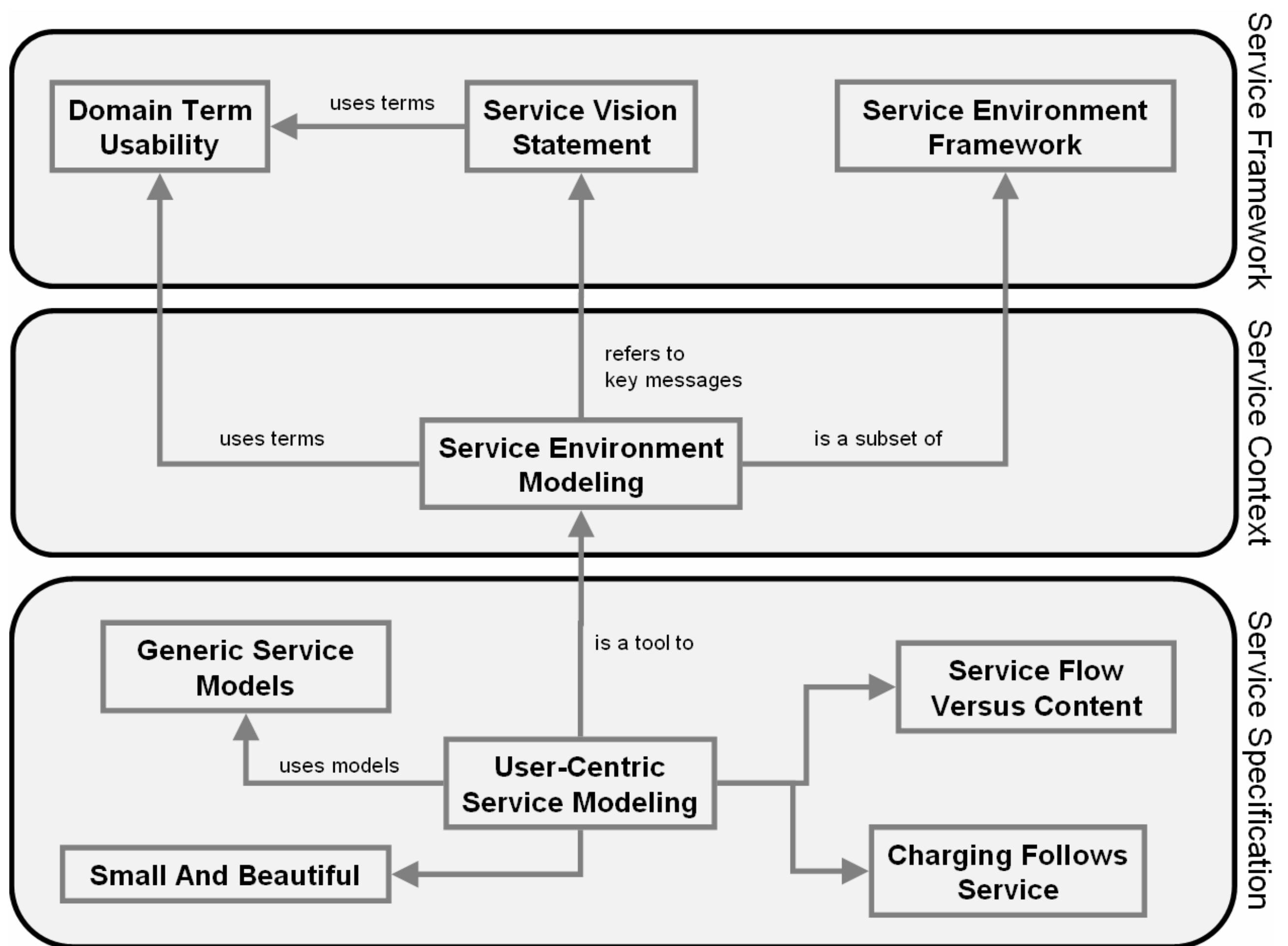

Figure 2: SDSA Pattern Language Overview

The names of the pattern domains reflect, that a developer will usually adjust to a natural sequence of utilizing them in a service framework. This reflects the way of specifying an idea in going from general to specifics (i.e. from blurring idea to a technical implementation).

Explaining some used terms in (Figure 2: SDSA Pattern Language Overview) might be helpful to further increase its understanding:

Domain (Knowledge Domain, Problem Domain)	Every information which specifies a realm for which a service is supposed to be implemented
Environment	Anything what might influence the technical implementation (= service), e.g. users, stakeholders, rules, communities, etc., but will not be part of the service implementation itsself
Modeling	The action of creating a timeline-based specification of distinct acts which the service or service environment flow of actions
Charging	The flow of actions to bill a service

Service	Anything, which serves as the technical implementation of system under development
Service Flow	The sequence of actions taking place to form a full-fledged service
Vision Statement	A easy-to-read and interpretable statement of the overarching objectives of a service; usually comprises a problem statement and a product/service positioning statement

Table 3: Definitions of the SDSA Domain Overview

Thus, a valuable SDSA should start on a general understanding of communication actions in services. Paramount is not to implement a specific problem domain to serve as a reference model later on (as requested by a DSSA definition). Every SDSA should instead set up a toolset, which clearly reflect a problem domain from a technology-agnostic point of view.

4. The SDSA Pattern Library

“The trick is to introduce already existing ideas into the mainstream without excessive use of authority. Why use a sledgehammer when a feather will do?”,
Richard Tanner

The SDSA Pattern Library should serve software architects in designing software platform architectures, e.g. by defining an ADL based on SDSA principles. It will separate between three specific realms of a software development process, i.e.

- the codification and description of the knowledge domain
- the specification of the broader service’s context and its environment (people involved, stakeholder affected, rules to be incorporated or newly challenged, division of effort, etc.)
- the specification of the service from a stern user-driven point of view

We suppose, that building software architectures based on these principles could reduce the risks and enhance the opportunities to deliver services, which will bring value add to users, stakeholders, and the organizations which use services developed by these guiding principles.

4.1. Service Environment Framework

“Awareness is only one piece of an effective technology adoption plan;
even the best technology is useless unless applied.”,
Jim Skinner

Context:

- You might have started a solution for **Domain Term Usability**

- You want to consider circumstances for the software architecture, which might not be an integral part of the software development process, but rather will influence its final outcome and objective (e.g. stakeholders motives, hidden rules, organizational constraints)

- You need an idea on how to systematically aufbereiten 'soft' information around the development of services

Problem:

How can you encourage software developers keeping an eye on the impact of the service on users, stakeholders, promoters, or the organization during the software development process? Is it possible to bear in mind consequences and challenges of the service's environment during the developing process?

Driving Forces:

- Service specification often is done without considering the broader impact of its creation, launch, or even its life cycle

- Developers are often led by leveraging capabilities of available technology; they sometimes forget the primary goal of implementing a service useful to customers or are not interested in it

- Numerous stakeholders often have their own views and opinions about a service idea, its creation process, and its launch; a software architecture does not feature anything, to consider its impact on the software development process

- There is value in looking at things in a different way by enhancing the view and getting beyond "gut fellings" of assumptions to understanding hard facts

Solution:

Offer possibilities to define a service as an embedded part of its broader environment.

The software architecture should offer to make the service's usage scenario a part of the definition on a high level. It should help to document the service's environment according to a simple and easy-to-understand method.

Examples:

The principles of Activity Theory (Leontiev 1978; Engeström and Middleton 1996) provides such a framework. Activity Theory offers a framework, ranging from explaining personal and collaboration aspects as well as business and technical aspects of developing services. Activity Theory looks "from outside", describing the broader environment of creating a service.

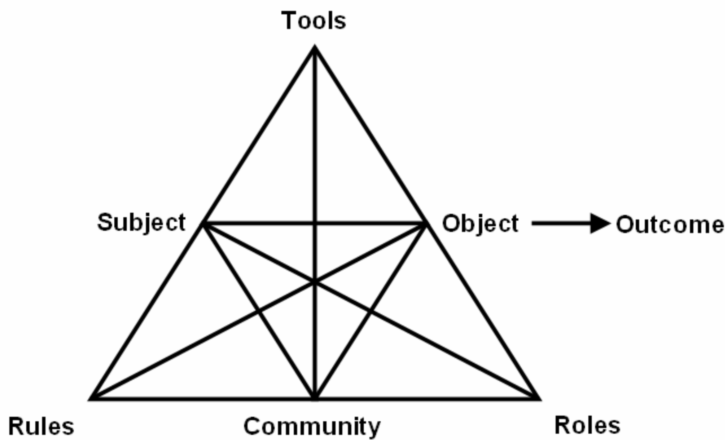

Figure 3: Activity System Overview

Activity Theory claims, that human deeds can be seen as a process, which can be modelled in an Activity System: The subject (= carpenter) changes an object (= wood, timber) to create a wishful or expected outcome (= rooftop, house). Most likely, he will use tools (= hammer, nails) to make it easier to achieve his goal. He even has to stick to rules (= static calulcations, house map) and has to arrange his actions within a community of others (= architect, landlord). The more complex an activity is, the more roles will emerge to achieve the ultimate goal (=carpenter, roofer, helping staff).

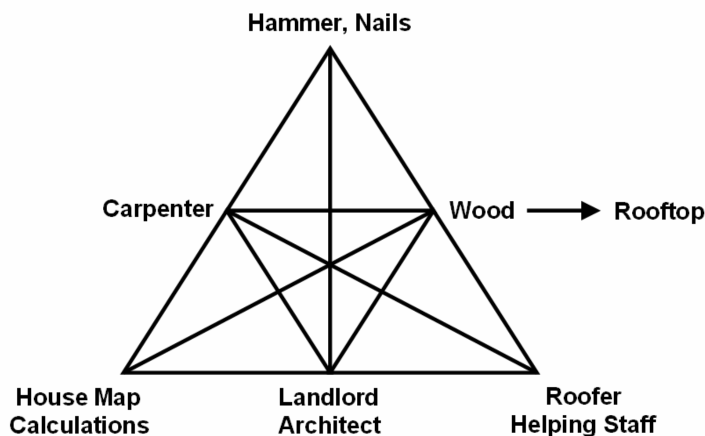

Figure 4: Activity System of a Carpenter's Work

Matching this model upon software architectures, the developed service is just the ‘tool’ of a larger activity system. The software architecture should offer opportunities to describe, outline, and refer even to all other components of an activity system such as the subject, the object to be changed, the expected outcome, the open and hidden rules, the involved community, and roles in the division of effort.

4.2. Domain Term Usability

“The limit of my language means the limits of my world”,
Ludwig Wittgenstein

Context:

- You might have implemented **Service Vision Statement**
- You need a base for **Service Environment Modeling**
- You want to create a software architecture from the scratch
- You would like to create a software architecture which reflects the creation of services for a specific knowledge domain or a product line

Problem:

How can you make it easy to refer to the knowledge of a problem domain during the archetypical software development process? How can you guarantee whether problem domain knowldege is fully be considered by what will be implemented?

Driving Forces:

- Developer's capabilities to put aside technology during the software development process is sometimes week; they usually share little interest to cope with knowledge domain concepts on a deeper level then necessary
- Problem domain expert's knowledge is deeply contextual and triggered by circumstance and year-long experience. In understanding what they know about the problem domain, developers would have to recreate the context of their knowing if they ask a meaningful question or enable knowledge use
- Developers and problem domain experts often do not understand each other; they use different words, different phrases, and different languages
- Inside-the-box definitions of problems guarantee inside-the-box-solutions
- Restating the problem shifts attention to fertile new ground and opens minds to new possibilities
- Reverting to tired clichés and pseudo-challenges
- Casting a problem in different light and using hard data to confront orthodoxies can be utilized to discover whether there are exceptions to the status quo
- Reframing a problem helps to focus attention on numerous possible solutions

Solution:

Create a neutral problem domain reference of terms and notions and make it visible all the time during the development.

Making it even simpler for everyone involved in the software development (e.g. product managers, engineers, users), create a reference, which can be understood by common sense. It should reflect all definitions, terms, notions and models (e.g. taxonomies, ontologies) necessary to understand the service from a problem domain point of view.

The software architecture should oblige developers to check up on a regular basis, whether they are really on track in understanding the service. Definitions should be easy accessible at any stage of the software development cycle in glossary, ontologies, and/or taxonomies. The software

architecture shold force developers to avoid abstract terms and notions because those will hide specific knowledge of the problem domain.

Examples:

1. Create a glossary listing all the crucial terms describing the problem domain and force utilizing those terms and their concepts within the software architecture. A simple glossary could look like this

Term	Short Definition	Long Definition	Synonyms	Related to	Translations	See Also

Table 4: Sample Glossary Template

2. Force the use of glossary definitions during the process of creating the services (e.g. by direct access to definitions while coding software with mouseover-effect).

Figure 5: Sample for Mouseover Effect

4.3. Service Vision Statement

"Users had been climbing in through the window for years before Apple finally moved the door.",
Paul Graham

Context:

- You already have implemented **Domain Term Usability**
- You have done a **Service Environment Framework**
- You like to create a possibility to grasp the general idea of a service and make it available within the software architecture at any level and detail

Problem:

How can you support developers keeping an eye on central relevant ideas of a service without getting sidetracked and distracted by a multitude of possible technical challenges and implementations?

Driving Forces:

- Capability of remembering the generic idea of a service during software development at any time
- Like-minded group of engineers are more safe in resolving technical problems, not knowledge domain problems which they are not experts
- Developers utilize concepts from their given technical domain and prefer to remain with their own concepts, even when explaining the knowledge domain

Solution:

Clarify the central idea of a service in a vision statement and foster developers to refer to it during the whole development process.

This vision should persist through all stages of the development process. It should outline in an easy manner the most important features, the service is expected to deliver. Any preliminary development results should automatically be matched against the vision statement.

Examples:

1. According Moore (Moore 1991), passing the 'elevator test' is crucial for any new product or service. He recommends a product position statement, which in easy words says what the product is about, who will be the customer, and what will be the unique selling proposition.

For an SGI workstation the product position statement could look like this:

For	Post-production film engineers
Who are	Dissatisfied with the limitations of traditional film editors
Our workstation is	A digital film editor
That lets you	Modify film images any way you choose.
Unlike workstations from	Sun, HP, or IBM
We	Have assembled all the interfaces needed for post-production film

	editing.

Table 5: Product Position Statement (Moore 1991)

The availability of such a template could become an integral part of a software architecture for a way to remember the overall service vision at any step in the software development process. Developers should be forced to match preliminary results against the written product position statement.

2. Consider, how the format of Powerpoint – although supposed to be easy-to-read and understand - can blur or hide hard facts: before NASA's devastating loss of the Columbia space shuttle, engineers from Martin Marietta and Boing buried imminent risks to the spacecraft's protective ceramic tiles within the complicated, nested, ten-point-font bullet point of their Powerpoint presentation (Tufte 2006). Thus, even a simple vision statement must be considered serious.

4.4. Service Environment Modeling

"A game is the total of rules describing it.",
John von Neumann

Context:

- You have created a **Service Environment Framework**
- You can guarantee **Domain Term Usability**
- You support to create a **Service Vision Statement**

Problem:

Software architectures usually do not force to document, explain, or reflect on the service environment. Forgetting about the stakeholder issues, or limitations and constraints for the service deployment, often leads to misinterpretation, frustration, and anger after its deployment.

Driving Forces:

- Stakeholder issues are usually documented (if at all) at the beginning of the software development process and are seldom referred to during the development
- Developers sometimes ignore facts, which are not service-specific, but will be part of the service environment after launch and deployment
- Important information and knowledge about the service's environment for deployment are hard to get – and even harder to codify for later use
- A service might be complicated (like an aircraft, that can be fully explained to the last screw), rather a service's environment will be complex (like any human system with interacting agents, which can not simply explained). Here, cause and effect can not be separated because they are intimatley intertwined (Juarrero 1999).
- Properly understood, a service paradoxically is both a "thing" (implementation) and a "flow" (e.g. stakeholder motives, user behaviour)

Solution:

Separate modeling the service from modeling is environment.

Differentiate service specification from service environment specification. The context of a service might be relevant for the business case modeling (and must be therefore made clear and documented), but it is supposed to be not part of the actual specification of the service. Separating those aspects from the service specification makes this clear.

Examples:

While the concept of Activity Theory (Figure 3: Activity System Overview) refers to the general idea to establish a framework where to integrate any kind of service in a wider context, now you would like create a possibility to model the workflow within a specific Activity System.

Basically, an Activity System can reflect four distinguished activities which are responsible for any action taking part within: Production, consumption, distribution, and exchange (=communication) (Engeström 1987). A service under development usually will have to support one of those archetypical service realms, or a mix of some.

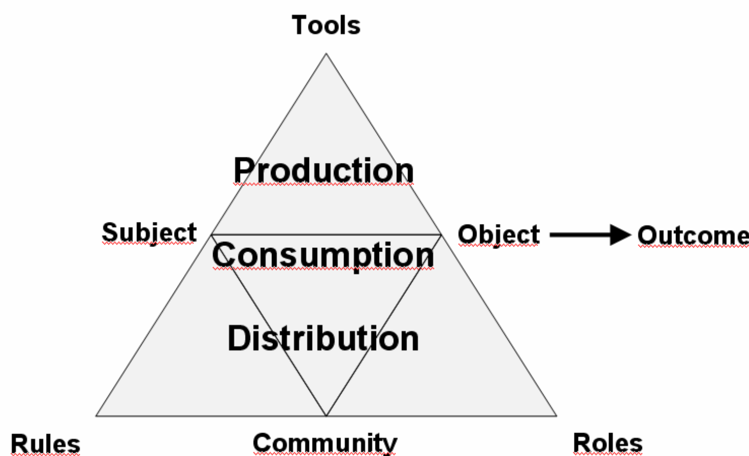

Figure 6: Possible Actions within an Activity System adjusted to (Engeström 1987)

Any category stands for a specific type of service. The following table presents some typical services for each category:

Production	Write a text Record a speech Draw an image
Consumption	Receive a text Listen to music Watch TV show
Distribution	Up- and download files

	Broadcast a movie Send a mail

Table 6: Typical services based on Archetype Activities

The software architecture should support to model those four service workflows, which can take place in an Activity System. The modeling itsself could be based on creating essential use cases (Constantine and Lockwood 1999). Essential use cases abstract from sticking to technical implementations, they should rather outline the general idea of what will happen within the service environment, if the service will be launched.

4.5. User-Centric Service Modeling

"Users had been climbing in through the windows for years before Apple finally moved the door",
Paul Graham

Context:

- You have done **Service Environment Modeling**
- You have created a way for simple reuase of service templates from **Generic Service Models**

Problem:

How can you support developers in sticking to the service vision statement? How can you enforce developers that they have to consider/beachten the service environment?

Driving Forces:

- Developers often forget the importance of the paramount service goal, when they are eager to dive into technical details
- The more complex a service becomes, the more complicated it is, to overlook the whole project and its possible ramifications
- In a complex software development project, often technical challenges become the driving topics of project management

Solution:

Strictly support specifying a service from a user's point of view.

Force developers to keep in mind that services must strongly be modelled from a problem point point of view. They should bear in mind the limitations, challenges, and opportunities, which arise from the service's environment at any time of the software development process. Encourage developers, that they have to develop services along a set of fundamental design principles by automatically confront them with those during the service implementation process, at any milestone, or with the arrival of any preliminary result. Force developers they must directly refer to terms and notions imported from the knowledge domain glossary in their e.g. service models or database designs. Any translated term, translated from the knowledge domain to a technical term, becomes a potential point of failure in understanding the knowledge domain.

Examples:

1. Messages in Office software often warn user's of possible mistakes like storing data before leaving the programme. Those warnings help the user not do lose consistency of the task she wants to finalize.

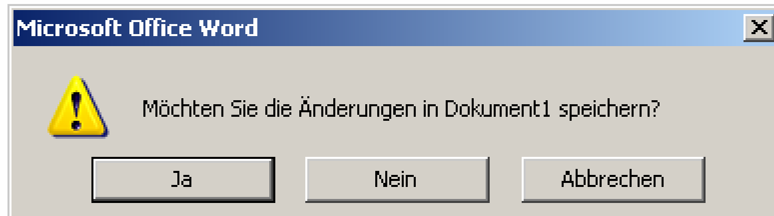

Figure 7: Winword Warning Message

2. Provide guidelines which list e.g. universal design principles (The Center for Universal Design 1997; The Center for Universal Design 1997; Lidwell, Holden et al. 2003), which are

1. Equitable Use: The design is useful and marketable to people with diverse abilities
2. Flexibility in Use: The design accommodates a wide range of individual preferences and abilities.
3. Simple and intuitive: Use of the design is easy to understand, regardless of the user's experience, knowledge, language skills, or current concentration level.
4. Perceptible Information: The design communicates necessary information effectively to the user, regardless of ambient conditions or the user's sensory abilities.
5. Tolerance for Error; The design minimizes hazards and the adverse consequences of accidental or unintended actions.
6. Low Physical Effort: The design can be used efficiently and comfortably and with a minimum of fatigue.
7. Size and Space for Approach and Use: Appropriate size and space is provided for approach, reach, manipulation, and use regardless of user's body size, posture, or mobility.

Table 7: Principles of Universal Service Design (The Center for Universal Design 1997)

3. The IBM Institute for Business Value analysis argues, to follow seven dimensions of user experience, which could serve as a guiding principle for the general design of a service (IBM Institute for Business Value 2006):

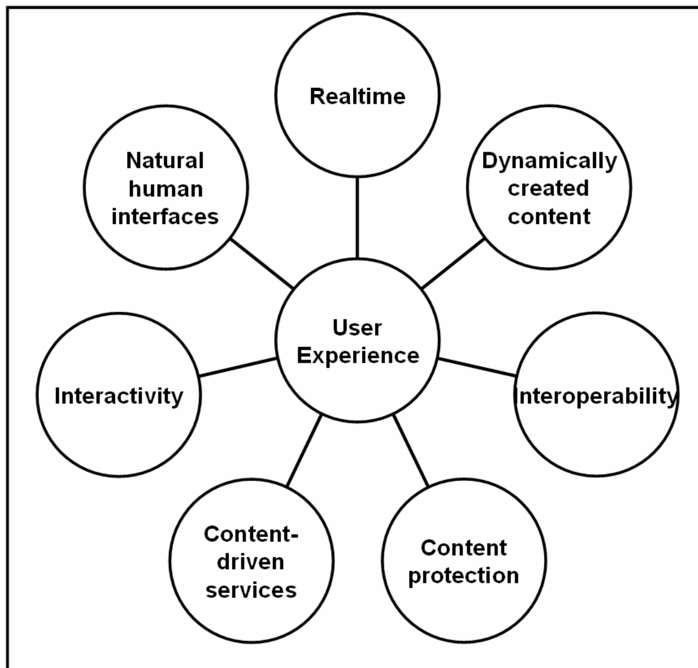

Figure 8: The Seven dimensions of user experience, all built on open standards (IBM Institute for Business Value 2006:7)

4. ‘Bumptop’ (Roush 2007) is a 3D environment for desktops, which make use of effects already known from gaming engines like light effects, shadows, or animations. The simulation of physical laws on a desktop is suppposed to increase the perception of the user and makes it easier for him to cope with files and folders.

4.6. Generic Service Models

“We're not here to invent something if it already exists.”,
Jane Marquart

Context:

- The overall volume of possible service workflows must be reduced for creating a specific service implementation.

- Developers have to decide which technologies to put in place for implementating a given service idea.

Problem:

How can you guarantee, that your software architecture will be capable of not only creating past or existing service ideas, but even new and unprecedented ones? How can you minimize or eliminate the risk of a major software architecture overhaul due to the introduction of technologies for new types of services, which would not fit in within the existing framework?

Driving Forces:

- A general definition of the term ‘service’ (beyond technology) is not available or seems to be unnecessary for developers to create their services

- Developers usually have no interest of understanding the common principles of services, if they do not recognize any technical challenge or constraint

- Service ideas are often directly linked to technologies, they are, actually, reflecting technical capabilities, not service ideas

- New technologies are being added to a software architecture, which force to change the mind-set of the developer by its impact on service creation

- Software architectures are generally driven by the integration or migration of technologies, not by new ideas of service capabilities beyond technical issues

- People never start from a zero base when designing a new service, all players during development come with the baggage, positive and negative derived from multiple histories

Solution:

Create a collection of fundamental service primitives.

With a general model about the nature of human deeds and actions, developers will get a guiding principle on how to create services from a principles point of view. Identify a set of meaningful, archetypical service primitives, which you could be offered as templates supposed to be put together for specific service idea.

Examples:

1. Any service can be traced back to simple acts of communication. Human communication is a directed exchange of information between a sender and a receiver (human to human, human to machine, machine to machine). Thus, refer to a model of basic steps of archetypical types of communication which are not related to a specific technology but to the context and sense of communication like given in (Shannon and Weaver 1949).

2. Vilem Flusser (Flusser 1998) distinguishes three general steps to compose communication services: creation, distribution, and storage:

Creation	**Distribution**	**Storage**
Create unique new data	Send or broadcast data	Put data to place for later use
Combine given data	Share with others	Organize data for later retrieval

Table 8: Acts of Communication according to (Flusser 1998)

Flusser even differentiates two principle types of communication. According to Flusser all (sic!) knowable forms of communication can be split into ‘dialogical’ (face to face, phone calls, direct mails) and ‘discoursive’ (music broadcasting, newsletters, blogs) acts of communication. Actually, Flusser states, that any service just seems to be a mix of these communication ‘primitives’.

Dialogues	**Discourses**
Recombine data by exchange of given	Convey information from sender to receiver

information (2-way)	(only 1-way)
Divergent, “open form”	Convergent, “closed form”
Compute new information	Distribute information

Table 9: Flusserian Communication Act are Sequences of Dialogues and Discourses

We suppose, that Flusser’s view on communication could become a valuable guideline for a general understanding of services and, therefore, could form the basis of a library of generic communication services.

4. IBM’s Activity Explorer (Geyer, Muller et al. 2006) is a platform supporting activity-centric collaboration. It integrates resources, tools, and people around the computational concept of a work activity and can be considered as abstracting from the technical level of a working tool to the level of underlying activities that the service is supposed to support.

5. The Media Richness-Theory (Dennis and Valacich 1999) goes back the theory of social presence and enhances the concept of “media richness”. Media Richness claims, that selection of media directly correlates with the activity behind it. Thus, according to Dennis and Valacich, enhanced technical capabilities will not improve the service itsself, it just depends on the underlying activity whether the “richness” of a given service is valuable to the user. “The ‘richest’ medium is that which best provides the set of capabilities needed by the situation: the individuals, task, and social context within which they interact.” (Dennis and Valacich 1999:3)

6. Amy Jo Kim outlines “five key mechanics of game design” which are: collecting things, earning points, providing feedback, exchanges, customization[4]. He argues for an application- instead of platform approach to building social media. Application foucs makes a game “easier to learn” because the cues all point in one direction.

7. “Buddy-centricity” as a paradigm of mobile communication services[5].

4.7. Service Flow Versus Content

“The most merciful thing in the world, I think,
is the inability of the human mind to correlate all its contents.”,
H.P. Lovecraft

Context:

- You have done **Generic Service Models**
- You want to implement a guideline on how to model the specific flow of a service
- You want to make it easy to create a bunch of services on the software architecture which is not limited to existing combinations of communication channels and specified content

Problem:

---

[4] http://www.oreillynet.com/conferences/blog/2006/03/how_game_mechanics_can_make_yo.html

[5] http://blog.wirelesswanders.com/2007/04/04/is-mobile-web-20-more-hype/

How can you create a software architecture, which is not stuck to already existing service categories? Is it possible to create software architectures which are beyond distinguishing services with regard to technical enablers (e.g. content services = streaming media, communication services = voice call, SMS, Email, etc.)?

Driving Forces:

- History of service architectures shows, that new service categories usually arise from their technical enablers

- Software architectures are created around ‘black boxes’ of specific services, which link a specific communication channel and its preferred content

- Bundled solutions often encourage to think in boxes, which might not be the best when thinking about the service development process from the end

- Fixed bundles reduce complexity. People like to reduce complexity.

- Developers usually develop services along existing technologies, not with a mind-set of separating communication channels and content

Solution:

Separate content from acts of communication.

Any communication service produces content, any content zwingend necessary for a communication service. Thus, separate specification of communication service specification from content handling. Separate service content from service orchestration. Both are part of *any* service.

Examples:

1. Web designers usually face the problem of separating presentation from content.[6] The major reason to separate presentation from the rest of the page is to simplify any change from a slight design adjustment to a full-fledged redesign. To achieve complete separation of the presentation, they isolate everything specifically and solely geared towards style.

2. Yahoo! introduced the “Pipes” concept[7] that strictly distinguishes between the process of service building and content collection.

4.8. Charging Follows Service

“Pecunia non olet”,
Latin saying

Context:

- You have an understanding for a **Service Modeling Framework**

- You created **User-Centric Modeling**

---

[6] http://www.alistapart.com/articles/separationdilemma/

[7] http://popoes.yahoo.com

- You need to integrate provisioning upon a software architecture

Problem:

How can you be sure, that a service is not uniquely driven by its provisioning model? How can you avoid, that a unique service idea will not be blurred by the fact, that you have to bring value-add to the service user before he will be provisioned?

Driving Forces:

- Specification of a service is a mix of service idea and service provisioning
- Provisioning types are preliminary endless and can be listed easier than possible service types
- Service types are supposed much more than types of provisioning

Solution:

Separate entities for modeling the service and modelling the provisioning.

Create a library of all principle types of provisioning a given service. Build a workflow, which explicitly forces to start creating a service, and then binding a specific provisioning to it. Implement two entities in the software architectures which separate service modeling and provisioning modeling. Create a 'provisioning library', which lists all possible forms of provisioning models and which can be enhanced any time a new provisioning scheme might come up.

	Direct Income Revenues	**Indirect Income Revenues**
Transaction-based	Transaction fees Connection fees Usage fees	Provisions
Non-transaction-based	Installation fees Basic fees	Banner ads Data mining revenues Sponsorship

Table 10: Principle Forms of Provisioning according to (Wirtz 2001:215)

Examples:

1. …

2. …

4.9. Small and Beautiful

"A car is more complex than a loaf of bread",
August-Wilhelm Scheer

Context:

- You want to support developers in dealing with the general structure of a software architecture
- You are looking for an idea on how to arrange software architecture modules in a simple and easy-to-grasp way
- A new module must be integrated in an existing software architecture

Problem:

How can you support developers coping with the complexity of software architectures which derives from the number of semantical concepts a software architecture would have to reflect?

Driving Forces:

- The more modules a software architecture consists of, the more complicated it is to deal with the way to combine the pieces
- Any additional module will enhance complexity of the software architecture
- Modules, which are specified on different levels of understanding, make it hard to understand their hierarchical interdependencies
- Each level of a specification for a system operates with different levels of abstraction, both implicit and explicit. The higher the level of abstraction, the higher will be the costs of disembodiment
- The best level of abstraction for a software architecture is a small zone, where all parties can agree on, that it reflects there personal understanding and background. The software architecture, thus, will become a “codification of the shared context and knowledge”.
- The more sophisticated an abstraction level is, the more complicated to to cope with it
- Form and function do not follow one another anyway
- People cannot communicate their needs completely to an architect, so the form that follows does not function well. On the other hand, a software architecture cannot entirely mold behaviour, so its function only partly follows its forms.
- Good software architecture design comes from dialogue between construction and use. If something is not quite right, fix the architecture a little bit (based on the general models), and allow user’s behaviour to adjust a little bit.

Solution:

Modules should be comprehensible without further knowledge.

When founding software architectures based on “simple ideas”, mistakes to make use of the platform will become easier to detect and to resolve. Mental concepts based on intuitive ideas of the nature of e.g. communication are more appropriate for people, because they are closer to their day-to-day experience. Thus, make every single concept very clear to be understood.

There is no such thing as a perfect fix, thus, make it easy to add or remove meaningful modules which might become more ore less important during the lifecycle of the software architecture. The smaller a possible change, the more likely it can be utilized in the right way: form and function come together.

Examples:

1. For the fact, that people can best deal with up to 7+-2 chunks of information (Miller 1956), the software architecture should contain the minimum on logical modules necessary to cope with on any level.

2. From research of biological systems, a fundamental rule can be derived, that the density of living beings is crucial for the likelihood of their surviving. If the density of living beings is low enough, the system is in a stable status, if it becomes to high (i.e. elements to close together), there are only two possibilities left: more-sophisticated organization on a higher level or degradation to an earlier grade of density (Vester 1999:72-74)

3. "The Oregon Experiment After Twenty Years" (Bryant, in: Rain Magazine, Vol. XIV, No. 1, Spring 1991:3pp) …

4. John Maeda introduced "The Laws of Simplicity" (Maeda 2006). It lists 10 principles to design better products.

5. Conclusion

"The future has already happened, it's just not very well distributed.",
William Gibson

We identified shortcomings of DSSA, namely being still technology-centric in their outline and proposed a Semantic-Driven Software Architecture (SDSA). An SDSA can be regarded as a special flavour of a DSSA, namely build along a line of creating software architectures from a semantic point of view, not technical concepts. Next, initial pattern for an SDSA pattern language were outlined which especially are based on concepts of Activity Theory according to Leontiev (Leontiev 1978) and Engeström (Engeström 1987) and findings on generic communication models (Flusser 1998).

A number of questions arose while creating the SDSA pattern language, presenting issues for further research and forthcoming efforts to enrich the initial pattern language:

- Will an SDSA-based software architecture be more complex or simpler than today, based on non-technical principles?

- What is a meaningful packaging or a modularization for an SDSA-enabled making it easy to utilize?

- What does the "end-to-end" argument according to Saltzer et al. (Saltzer, Reed et al. 1988) mean for an SDSA-based software architecture?

- How would an organizational change adapt to the introduction of an SDSA-related software architecture?

We see some interesting direction for research with regard to the future of software architectures. We e.g. assume, that one way to build future software architectures will be the introduction of bionic principles (Vester and Hesler 1982). May be, software architectures in the future can even

learn and evolve from their having done their individual experiences. The question could arise whether we could discover an ultimate “services DNA” which would give us a kind of ultimate toolbox to create services.

6. Acknowledgements

The author would like to thank Michael Weiss for his advice and valuable contributions as shepherd for EuroPloP 2007.

7. Appendix

7.1. Pattern Overview

Nr.	Name	Problem	Solution
1	Service Environment Framework	How can you encourage software developers keeping an eye on the impact of the service on users, stakeholders, promoters, or the organization during the software development process? Is it possible to bear in mind consequences and challenges of the service’s environment during the developing process?	*Offer possibilities to define a service as an embedded part of its broader environment.*
2	Domain Term Usability	How can you make it easy to refer to the knowledge of a problem domain during the archetypical software development process? How can you guarantee whether problem domain knowldege is fully be considered by what will be implemented?	*Create a neutral problem domain reference of terms and notions and make it visible all the time during the development.*
3	Domain Term Usability	How can you support developers keeping an eye on central relevant ideas of a service without getting sidetracked and distracted by a multitude of possible technical challenges and implementations?	*Clarify the central idea of a service in a vision statement and foster developers to refer to it during the whole development process.*
4	Service Environment Modeling	Software architectures usually do not force to document, explain, or reflect on the service environment. Forgetting about the stakeholder issues, or limitations and constraints for the service deployment, often leads to misinterpretation, frustration, and anger after its deployment.	*Separate modeling the service from modeling is environment.*

Nr.	Name	Problem	Solution
5	User-Centric Service Modeling	How can you support developers in sticking to the service vision statement? How can you enforce developers that they have to consider/beachten the service environment?	*Strictly support specifying a service from a user's point of view.*
6	Generic Service Models	How can you guarantee, that your software architecture will be capable of not only creating past or existing service ideas, but even new and unprecedented ones? How can you minimize or eliminate the risk of a major software architecture overhaul due to the introduction of technologies for new types of services, which would not fit in within the existing framework?	*Create a collection of fundamental service primitives.*
7	Service Flow Versus Content	How can you create a software architecture, which is not stuck to already existing service categories? Is it possible to create software architectures which are beyond distinguishing services with regard to technical enablers (e.g. content services = streaming media, communication services = voice call, SMS, Email, etc.)?	*Separate content from acts of communication.*
8	Charging Follows Service	How can you be sure, that a service is not uniquely driven by its provisioning model? How can you avoid, that a unique service idea will not be blurred by the fact, that you have to bring value-add to the service user before he will be provisioned?	*Separate entities for modeling the service and modelling the provisioning.*
9	Small and Beautiful	How can you support developers coping with the complexity of software architectures which derives from the number of semantical concepts a software architecture would have to reflect?	*Modules should be comprehensible without further knowledge.*

7.2. Glossary

ADL	Architecture Description Language
DSSA	Domain-Specific Software Architecture
ICT	Information and Communication Technology
SDP	Software Development Platform

SDSA	Semantics-Driven Software Architecture

8. References

Alexander, C. (1978). The Oregon Experiment, Oxfort University Press Inc, USA.

Alexander, C. (1979). The Timeless Way of Building. New York, Oxford University Press.

Alexander, C., S. Ishikawa, et al. (1977). A Pattern Language: Towns, Buildings, Construction. New York, Oxford University Press.

Berners-Lee, T. (1990). Information Management: A Proposal.

Berners-Lee, T., J. Hendler, et al. (2001). "The Semantic Web."In: Scientic American **284**(5): 34.

Boehm, G. (1994). Was ist ein Bild? München, Fink.

Chirita, P.-A., S. Costache, et al. (2005). Beagle++: Semantically Enhanced Searching and Ranking on the Desktop. Hannover, University of Hanover**:** 15.

Cockburn, A. (2000). Writing effective use cases. Boston, Addison-Wesley.

Constantine, L. L. and L. Lockwood (1999). Software for Use. A Practical Guide to the Models and Methods of Usage-centred Desing, Addison Wesley.

Constantine, L. L. and L. A. D. Lockwood (1999). Software for Use: A Practical Guide to the Models and Methods of Usage-centred Design.

Dennis, A. R. and J. S. Valacich (1999). Rethinking Media Richness: Towards a Theory of Media Synchronicity. 32nd Hawaii International Conferences on System Sciences. Los Alamitos, IEEE.

Engeström, Y. (1987). Learning by expanding. Helsinki, Orienta Consultit.

Engeström, Y. and D. Middleton (1996). Cognition and Communication at Work. Cambridge, Cambridge University Press.

Farb, P. (1993). Word Play. What Happens When People Talk.

Flusser, V. (1998). Kommunikologie.

Gamma, E. (1995). Design patterns : elements of reusable object-oriented software. Reading, Mass. ; Wokingham, Addison-Wesley.

Garlan, D., R. Allan, et al. (1994). Exploiting Style in Architectural Design Environments. SIGSOFT Software Engineering Notes. **19:** 175-188.

Garlan, D. and R. Allen (1994). Formalizing Architectural Connection. 16th International Conference on Software Engineering, Sorrento, Italy, IEEE Computer Society Press.

Gause, D. C. and G. M. Weinberg (1989). Exploring requirements : quality before design. New York, NY, Dorset House Pub.

Geyer, W., M. J. Muller, et al. (2006). Activity Explorer: Activity-Centric collaboration from research to product. IBM Systems Journal. **Vol. 45:** 713-738.

Hayes-Roth, R. (1994). Architecture-Based Acquisition and Development of Software Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program, Teknowledge Federal Systems.

Hayes-Roth, R. and W. Tracz (1994). DSSA Tool Requirements for Key Process Functions. Technical Report ADAGE-IBM-93-13B. Owego, Loral Federal Systems.

IBM Institute for Business Value (2006). Who's in charge here? In the world of digital convergence, it's the end user. Somers, IBM Global Services.

Juarrero, A. (1999). Dynamics in Action: Intentional Behavior as a Complex System, MIT Press.

Leontiev, A. N. (1978). Activity, Consciousness and Personality. New York, Prentice Hall, Englewood Cliffs.

Lidwell, W., K. Holden, et al. (2003). Universal Principles of Design: A Cross-Disciplinary Reference: 100 Ways to Enhance Usability, Influence Perception, Increase Appeal, Make Better Design Decisions, and Teach Through Design Rockport Publishers Inc.

Luckham, D. C. and e. al. (1995). "Specification and Analysis of System Architecture Using Rapide."In: IEEE Transactions on Software Engineering **21**: 336-355.

Maeda, J. (2006). The Laws of Simplicity, The MIT Press.

Medvidovic, N. and R. N. Taylor (2000). A Classification and Comparison Framework for Software Architecture Description Languages, University of Southern California**:** 1-24.

Miller, G. A. (1956). The Magical Number Seven, Pluse or Minus Two: Some Limits on Our Capacity for Processing Information.

Moore, G. A. (1991). Crossing the Chasm. New York, HarperCollins.

Norman, D. A. (1998). The invisible computer : why good products can fail, the personal computer is so complex, and information appliances are the solution. Cambridge, Mass., MIT Press.

O'Reilly, T. (2004). "What Is Web 2.0?"In.

Roush, W. (2007). "Göttliche Unordnung auf dem Desktop." Technology Review Retrieved 12.4.07, 2007, from http://www.heise.de/tr/artikel/87987.

Saltzer, J. H., D. P. Reed, et al. (1988) End-to-End Arguments in System Design. **Volume**, DOI:

Schneider, G. and J. P. Winters (1998). Applying use cases : a practical guide. Reading, Ma., Addison-Wesley.

Schneider, G. and J. P. Winters (2001). Applying use cases : a practical guide. Boston, Addison-Wesley.

Shannon, C. F. and W. Weaver (1949). The Mathematical Theory of Communication, Urbana, Ill.: The University of Illinois Press.

Shaw, M. and e. al. (1995). "Abstractions of Software Architecture and Tools to Support Them."In: IEEE Transactions on Software Engineering **21**(6): 314-335.

Spivack, N. (2006). "The Third-Generation Web is Coming." Retrieved 26.3.07, 2007, from http://www.kurzweilai.net/meme/frame.html?main=/articles/art0689.html.

The Center for Universal Design (1997). The Principles of Universal Design. Raleigh, NC: North Carolina State University. **2007**.

The Center for Universal Design (1997). The Principles of Universal Design. Raleigh, NC, North Carolina State University.

Tracz, W. (1995). "DSSA (Domain-Specific Software Architecture) Pedagogical Example."In: ACM SIGSOFT **20**(3): 49-62.

Tufte, E. R. (2006). The Cognitive Style of PowerPoint: Pitching Out Corrupts Within, B&T.

Vestal, S. (1996). "Languages and Tools for Embedded Software Architectures." Retrieved 2.3.07, 2007, from http://www.htc.honeywell.com/projects/dssa/dssa_tools.html.

Vester, F. (1999). Die Kunst, vernetzt zu denken (The Art of Interlinked Thinking). Ideen und Werkzeuge für einen neuen Umgang mit Komplexität. Stuttgart, Deutsche Verlagsanstalt.

Vester, F. and A. v. Hesler (1982). Sensitivity Model. Frankfurt/Main, Umlandverband Frankfurt.

Weiser, M. (1991). "The computer for the 21st century."In: Scientific American **265**(3): 94-104.

Weiser, M. (1994). The world is not a desktop. Interactions**:** 7-8.

Wirtz, B. W. (2001). Electronic Business, Gabler Verlag.