Summary of course Software architecture

http://kmi.tugraz.at/staff/denis/courses/sa/index.html

Lukas Prokop

8th of Dec 2011

Contents

Chapter 1

Software architecture

The following summary shall give you a general framework how to progress from a software analysis process over a design phase to a final implementation goal. However, in difference to courses like OAD, we will focus on the various architectures which can be chosen for the implementation.

1.1 Requirements

In the first phase we will receive customer wishes which are informal and unprecise. It's our task to formalize them and define corresponding requirements.

1.1.1 Requirements

There are three types of requirements:

- non-functional (quality attributes, runtime-related or not)
- functional (structured language, use cases, description by formal methods like state diagrams)
- contextual (technology to be used, law / expertise / customer needs / customer experience)

Each of the non-functional requirements addresses one of the quality attributes, which are described in two different lists: MeTRiCS and PURS. Please remember that we are not talking about components, classes and design decisions yet. Specifically quality attributes are not component specific.

Using those requirements addressing quality attributes, we can identify the key concepts of the application domain. Identifying it will help us to keep attention to the customer's wishes and understanding his requirements.

Those requirements can be defined in multiple iterations.

1.1.2 PURS

- Performance
  - Database design

  - Choice of algorithms
  - Communication
  - Resource management
- Usability
- Reliability
  - MTTF: mean time to failure
  - MTBF: mean time between failures
  - MTTR: mean time to repair
  - For web systems: MTTF $\rightarrow \infty$
  - Acceptability of bugs depends
- security
  - Authentication and Authorization
  - ACL (Access Control List)
  - RBAC (Role-Based Access Control)
    - resolves many-to-many relationship between user & person

1.1.3 MeTRiCS

- Maintainability
  - Code comments
  - OOP design principles
  - consistency
  - documentation
  - Don't be afraid of refactoring, redesigning and rewriting!
  - throw-away prototypes
  - keep innovative
- Evolvability
- Testability
- Reusability
- Integratability
- Configurability
- Scalability

1.2 Analysis

Requirements $\Rightarrow$ Key concepts $\Rightarrow$ application domain.

1.3 Design

Design consists of creating models. There 2 kinds of models:

- Structural model
  is a static model of the system (ie. division of the system into components and connectors).
- Behavior model
  is a dynamic model of the system (ie. component interactions)

Functionality is what the system can do and behavior is the activity sequence. Each component has a set of responsibilities. Behavior is the way how these responsibilities are exercised to respond to some event. An event may be an action of the user or an event from an external system. A particular behavior is an event plus a response in the form of a sequence of component responsibilities.

Design can be codified by box-and-lines diagrams and comparing the model to OOP classes.

There are two major approaches how to identify components:

noun method underline keywords (ie. key concepts)

categorization of possible concepts in { Data[1], Function[2], Stakeholder[3], system[4], hardware[5], abstract concept[6] }

1.4 Architectural views

Conceptual Components are sets of responsibilities and connectors are flows of informations.

Executional Components are execution units (processes) and connectors are messages between processes.

Implementation Components are libraries, source code, ... and connectors are protocols & APIs

1.5 Methodology

A methodology for software development shall be chosen after the definition of requirements but before the software design process.

- rational
- spiral
- agile
- evolutionary rapid

[1]one component
[2]many components
[3]no components
[4]external systems; are probably 1 component
[5]one physical component
[6]rarely components

Chapter 2

Execution architecture

Many quality attributes are explicitly addressed by the execution architecture. Execution architecture focuses on the system structure (eg. performance, security, reliability) and addresses concurrent components, which are always simply summed up to components up to now. In a single-computer, single-process and single-thread environment, the execution architecture is very simple.

Executation architecture becomes very important whenever you design network-based systems, multi-core or multiprocessing units, multi-threaded systems or event-based GUIs. Components are considered to be either hardware, concurrent subsystems, processes or threads. Connectors indicate either synchronous, asynchronous calls or callback mechanisms.

Each model in Executation architecture is a model at a specific level of granularity. Subsystems can be quite complex.

2.1 Concurrent subsystems

- Always long-lived
- Creation means start of the service, Closing means shutdown of the service
- Operates through lifetime

2.2 Process model

- Restricting a concurrency model to processes
- depicts the execution structure
- Do not go into details on external subsystems

2.3 Execution stereotypes

user initiated This stereotype describe systems where the events are controlled by the user. All UI components are always of this kind of stereotype. There might be a separate even thread listening to user input events.

active components generate activity internally. This might be eg. a crawler of a search engine.

services wait for requests of other components. They typically perform complex tasks and have a clear communication protocol. Example includes databases and the web.

2.4 Comparison to CA

There are *no rules* how to decide where conceptual components reside in the execution architecture (described by term "Binding execution and conceptual models").

Element	Conceptual architecture	Execution architecture
Components	Domain-level responsibilities	Unit of concurrent activity
Connectors	Information flow	Invocation
Views	Single	Multiple

Chapter 3

Implementation architecture

IA focuses on *how* the system is built. It defines a set of technological elements for the implementation. This includes software packages, libraries, frameworks and classes. IA is a number of implementation models. It addresses non-runtime quality attributes.

Components and connectors reflect software entities and relationships (connectors describe a "uses" relationship). Each IA focuses on one of the concurrent subsystems or processes from the execution view.

3.1 Application components

An application component is implementing domain-level responsibilities, which can be found in the conceptual architecture. But also a number of conceptual components can be mapped to a single application component. Realization takes place by source code, files or binaries. UI components typically map onto a single application component.

3.2 Infrastructure components

Infrastructure components are necessary for the running system, but do not relate to the application functionality. Sometimes they act as an container for several application components.

A container component provides an execution environment for the contained components. So typical container components include frameworks, servers, generic clients and off-the-shelf components.

"Impact Maps" might help to illustrate dependencies between components.

3.3 Connectors

Connectors in an IA are API calls, callbacks, network protocols and signals.

3.4 Infrastructure selection

The selection of infrastructure is heavily influenced by the different philosophies of the development team members.

- Conceptual issues: MVC, component-based, service-based
- Executational issues: Spawning of external components
- Implementation issues: programming languages
- Contextual issues: Commercial products, OpenSource
- Organizational issues: Know-How, Team

3.4.1 Algorithm (”Weighted Scoring method”)

For each of those issues you have to identify and weigh criteria. There are n alternatives $(A_1 \dots A_n)$ and m different criteria $(C_1 \dots C_m)$. Each alternative is for each criterion with score S_{ij}. Each criterion has a weight relative to its importance $W_1 \dots W_m$. The final score for A_i is:

$$S(A_i) = \sum_{j=1}^{m} S_{ij} \cdot W_j$$

3.5 IA definition procedure

1. Find application components
2. Find infrastructure components
3. Design interfaces
4. Define behavior design and verification

We need a clear interface between all components. In best case, those interface are already standardized (eg. HTTP). In behavior design we have to go into detail for IA and use case maps not precise enough for it; sequence diagrams are recommended.

Prototypes will help us to verify our final IA.

Element	CA	EA	IA
Components	Domain-level resp.	Unit of concurrent activity	Impl. module
Connectors	Information flow	Invocation	”Uses” relationship
Views	Single	Multiple	Split

Chapter 4

OO Design principles

Design is the process of modeling new software entities to satisfy requirements. Software entities might be different things depending on the level of granularity and programming paradigm. Design is not an exact entity and therefore not measureable.

Procedural	OOP
producedures	classes
variables	objects
	relationships

We have discuss 4 OO design principles:

1. Open-Closed
2. "Design by contract" or "Liskov substitution principle"
3. Single responsibility principle
4. Law of Demeter

4.1 Open-Closed

"Software should be open for extension but closed for modification"

- You should be able to extend a class without modifying it
- Reason for this principle: abstraction (fixed design but limitless behaviors)
- Early design decision: Fixed vs expandable system parts (clear interfaces)
- We can test components finally and extend them without modifications
- Example violation of this rule: usage of `instanceof` operator (destroys polymorphism)

4.2 Design by Contract

"Define a contract between a class and its users"

- Preconditions and postconditions for methods (you might want to insert tests for those conditions using `assert`)
  Invariants for the class (technical view of pre- and postconditions)
- Liskow Substitution Principle (related to "Design by Contract")
  - "If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T."
  - Polymorphism does it anyway, but we are talking about implemented methods in the sub- and superclass
  - Example violation of this rule: return value is null object and no real object.
  - Allowed within this rule: Weaken preconditions, strengthen postconditions
  - Cannot satisfy this rule? Refactor class hierarchy!

4.3 Single responsibility principle

"One entity (class, object), one responsibility" (separation of concerns)

- A responsibility is in fact a reason to change
- Example violation of this rule: Student is reading from DB *and* printing HTML

4.4 Law of Demeter

"A method of an object should only invoke methods of itself, its parameters, objects it creates and its members"

- Chaining of method calls across various objects is evil.

Chapter 5

Architectural styles

An architectural style is a particular pattern that focuses on the large-scale structure of a system typically described by the common vocabulary.

Variants of architectural style:

- Data-centered architecture
- Data-flow architecture
- Abstraction-layer architecture
- N-tier architecture
- Notification architecture
- Remote invocation and service architecture
- Heterogeneous architecture

5.1 Data-centered architecture (DCA)

Systems in which the access and update of a widely accessed data store is the primary goal. Basically it is just a centralized data storage with a number of clients. This architecture requires three protocols: communication, data definition and data manipulation. This architecture has two subtypes:

Repository client sends a request, system executes

Blackboard system spread notifications / data to subscribers

- ensures data integrity
- reliable, secure
- testability guaranteed
- Clients are independent → performance and usability at clientside good
- problems with scalability

  - shared repositories
  - replication; but increases complexity

Example implementations:

1. RDBMS (repository with database schemas and SQL for communication)
2. Web architecture (hypermedia data model, pages with links, HTTP for communication, integrity not guaranteed [404 error], extremely scalable)

See also: resource oriented architectures

5.2 Data-flow architectures (DFA)

An architectural style to improve the quality of reuse and modifiability. We are viewing the system as a sequence of transformations on successive pieces of input data. Finally this data is assigned to its destination. There are two variations:

structural variation more complex topologies like loops, branches, more than 1 input

communicational variation synchronous behavior

- Maintainability
- Modularity

Example implementations:

1. UNIX pipes (pipes for communication, probably concurrent processing of incremental data)
2. any kind of filter

5.3 Abstract layer architectures

Abstract layer architectures introduce a layering system where each layer is on top of another one and there are well-defined interfaces between them.

- reduces complexity
- improves modularity
- reusability
- maintainability

Example implementations:

1. Operating systems
2. Virtual machine (interface between compiler & real machine, improves portability)
3. Network protocol stack

5.4 N-tier architectures

N-tier architectures evolved from business applications and can nowadays be widely seen in client-server models for the web. They try to separate presentation, application and data storage components.

2-tier Rich clients run application and server stores data

3-tier Rich clients, application server and data server

Rich clients Clients which have full knowledge of application. Either they implement a standard application and/or protocol or a custom one.

Thin clients Small clients like in X-server architecture of UNIX-based operating systems

5.5 Notifications architecture

Information and activity gets propagated by a notification mechanism (listener and callbacks). It's basically a more abstract view of the blackboard data-centric architectures. Clients can "register" to receive notifications and receive data either when data is relevant for them ("interested user") or always ("broadcast").

Example implementations:

1. GUI event handling

5.6 Remote invocation and service architectures

This architecture involves distributed processing components and a client invokes a function on a remote component.

- Increased performance through distributed computation
- Tightly coupled components

Service architectures store where services are registered.

5.7 Heterogeneous architectures

No real system follows strictly only a single style. Therefore architectures might be structurally heterogeneous:

- Overall architecture follows one style
- Single components follows other style

Example implementations:

- Web has 2-tier architecture (browser and server) and browser has notification architecture (user events)
- Web-based search engine
  Conceptually: data-centric, layered, 3-tier
  Structurally: layered (network), 3-tier, notification
  Execution: distributed, service-oriented, ...

Chapter 6

Web architecture

Two views: Is the web an application platform or a huge distributed database?

The Web started as a static information system. *Hypermedia* are documents linked into a web. The user is able to retrieve documents by following unidirectional links. It's major advantage is that it only requires simple components and everything is based on HTTP, HTML and URL standards (cross-platform, global scope).

> Web's major goal was to be a shared information space through which people and machines could communicate
> —Tim Berners Lee

- Usability: easy to create, structure and reference information. voluntary participation, very error forgiving
- Simple: easy implementation, text-based components, only a "simple" HTML parser required
- Extensibility: requirements changed, forms were introduced
- Scalability: anarchic, dezentralized

Constraints were introduced on the web architecture to obtain an optimal solution to the requirements and quality attributes.

- Client-server model (separation of concerns)
- stateless architecture (client-side sessions)
  - Improves visibility, reliability, scalability
- information can be labeled as cacheable
  - Improves efficiency, scalability, user-perceived performance
- Uniform interface
  - Identification of resources (URL)
  - manipulation of resources through representations (HTML/XML)

- self-descriptive message (GET, POST, PUT, DELETE)
  - layered system (scalability by proxies, shared-caches and gateways; also load-balancing)
  - Code on demand (Java applets, flash, Javascript)

Web's specifics:

1. User requirements
2. User interface and usability
3. Application state and hypertext
4. Addressability
5. Architecture

6.1 Application state on the Web: Hypertext

- HTTP is stateless (connection-less)
- Application logic manages sessions
- AJAX problems: recovery of states in new session impossible, browser back button problem
- server state vs client state = scalability problems vs recovery problems
- Eg. Google Maps: permalinks for states (second addressbar)

6.2 Addressability

- URLs: good usability and meaninful links
- bookmarks
- linking of services is possible

6.3 User-oriented DB applications

- traditionally built with N-tier architecture (started with $N = 2$)
- DBMS layer (Relational Database) and app/presentation (scripts) layer
  Major issue: Changes in application logic leads to changes in presentation functionality
- Solution is a 3-layer system: Separation between Application and Presentation
  introduction of AJAX $\rightarrow$ application logic either on client or server
- MVC models since GUIs got introduced
  (especially popular in web applications)

6.4 Data management

- Data Access Object
- ORM (Object Relational Mapper)
- Information retrieval by fulltext search and/or link analysis
- Metadata (taxonomy/folksonomy)
- Data in Web is organized in simple node-link model
- Each node is a data item with a unique address and representation
- Programmers combine a number of services to achieve a desired functionality and create a distributed system (eg. mashups)

6.4.1 User-centric view

What is Google? A *service* to query a massive database (Web search index).
What is web application? A *service* offering a specific functionality (remotely)
What is a web site? A *service* offering specific human consumable information.

6.4.2 The Programmable Web

> "Programmers follow the architectural style to integrate and combine services to achieve a desired functionality"

HTTP is used for data transportation, XML for data representation, but services also offer HTML, JSON, plain text and binary data.

Classification

How to transmit method information (what to do):

- HTTP methods (standardized, not extensible)
- URL/GET parameter (not standardized, extensible, eg. Flickr)
- SOAP and WSDL

How to transmit scoping information (where to do):

- URL/GET parameter
- SOAP

Competing architectures in practice

- RESTful, Resource-Oriented architectures
- RPC-style architectures
- REST-RPC hybrid architectures

6.5 Remote Procedure Call (RPC)

Server receives a request envelope with all necessary data to perform a certain task. Server sends back a similar envelope.

Protocols: SOAP, XML-RPC

Problems:

- RPC implies an API
- APIs tend to enforce a tight coupling of modules and systems
- Where is all the nice Web architecture gone?

6.6 Resource Oriented Architectures

- method information in HTTP method
- scoping information in URL

4 features of ROA (resource oriented architectures):

1. addressability
2. statelessness
3. uniform interface
4. connectedness

Idempotence: same operation has same effect however often it is applied.

6.7 Designing RESTful services

REST originated from a PhD thesis by Roy Fielding.

1. Figure out data set
2. Split data set into resources
3. For each resource: define URL
4. Expose a subset of the uniform interface
5. Design representations accepted from the client
6. Design representations served to the client
7. Integrate this resource into other resources using links
8. Consider possible application states
9. Consider possible error states

Software packages: Java Restlet, Plugin for ruby on rails, Django in python