Designing Software Architectures Part 2/2

Dr. Martin Girschick

Software Engineering in Industrial Practice

WS2015/2016

Exkursion zu Capgemini

Technology Services persönlich kennenlernen

Wann? **Dienstag, 26. Januar 2016**

Oder **Freitag, 5. Februar 2016**

Wo? **Capgemini-Niederlassung in Offenbach**

Was? **Projektvorträge und Smalltalk**

Anmeldung **http://tiny.cc/exkursion-capgemini**

Fragen? **Mail an martin.girschick@capgemini.com**

Wir freuen uns auf Ihren Besuch!

Capgemini

CONSULTING.TECHNOLOGY.OUTSOURCING

Summary of Part I

- What is the purpose of software design?
- How to get from specification to construction?
- How do you produce good designs?
  - The architectural principles
  - The idea of the software component

What were the 10 architecture principles introduced last week?

Post on twitter under the hashtag #seiipTUD

We need a component definition that supports our architectural principles and an efficient project implementation.

Component – definition objectives

Objectives

Architectural principles

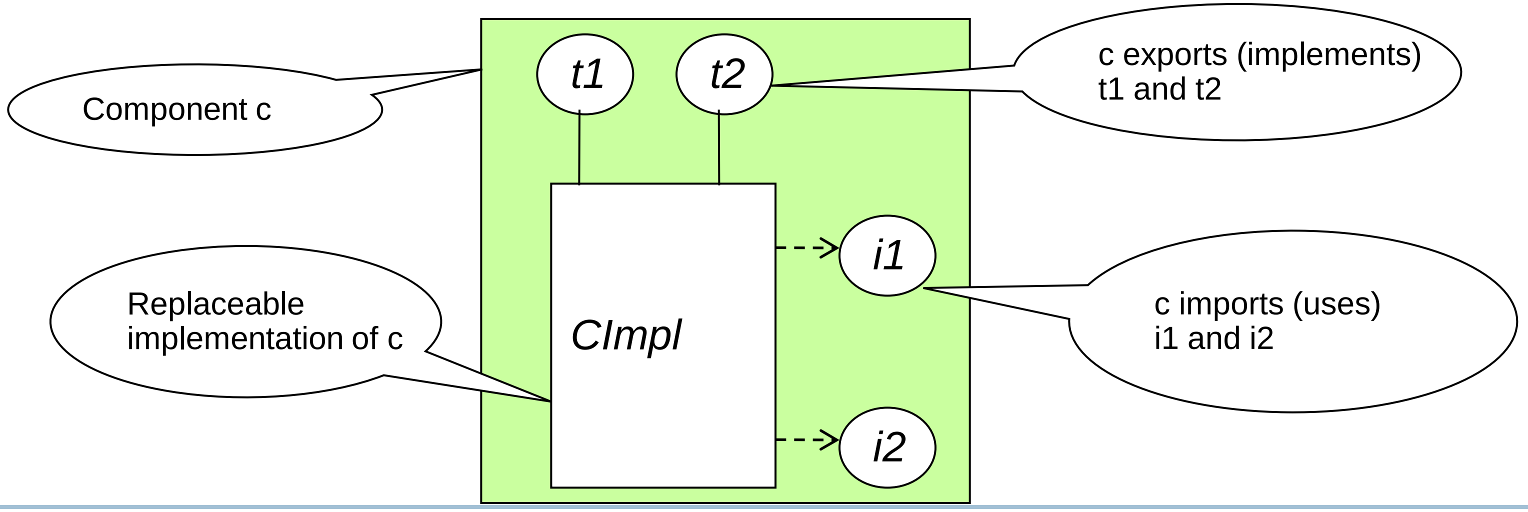
1. Separation of concerns
2. Minimization of dependencies
3. Information hiding
4. Homogeneity
5. Zero redundancy
6. Software categories
7. Layering
8. Design-by-contract
9. Data sovereignty
10. Reuse

Efficient project implementation

- Staffing
- Planning
- Controlling
- Responsibility

Quasar component definition – six features of a component

- A component
  1. **Exports** one or more **interfaces**
  2. **Imports** other **interfaces**
  3. **Hides the implementation** and can therefore be replaced by another component that exports the same interface
  4. Is suitable as a unit that can be **reused**.
  5. Can **contain other components**.
  6. Is, together with the interface, the **key unit in terms of design, implementation and planning**.
     - Hint for granularity: It can be built by **one or a few persons** within **reasonable time**.



Six features of a component

HANDOUT

- For the export interfaces there is a clear definition of the services offered, in particular the exact semantics of the interfaces.
- The import of an interface means that the component uses the services of this interface. It cannot run until all of the imported interfaces are available. This is the task of configuration.
- Each component hides the implementation and can therefore be replaced by another component that exports the same interface
- It is a unit that is suitable for reuse because it does not know the environment in which it runs. It only makes minimal assumptions about it.
- Components can contain other components or, to put it another way: You can put together (or compose) new components from existing components over an unlimited number of stages.
- Together with the interface, the component is the key unit in terms of design, implementation and thus planning.

The Quasar component definition supports our objectives.

Quasar component definition

HANDOUT

A component…

- …provides its functions in the form of interfaces,
- …uses interfaces of other components (must be configured),
- …has a replaceable implementation,
- …is a suitable unit for reuse,
- …may contain other components (composition),
- …is a meaningful unit for construction, implementation and planning.

design-by-contract
minimization of dependencies
data sovereignty
software categories

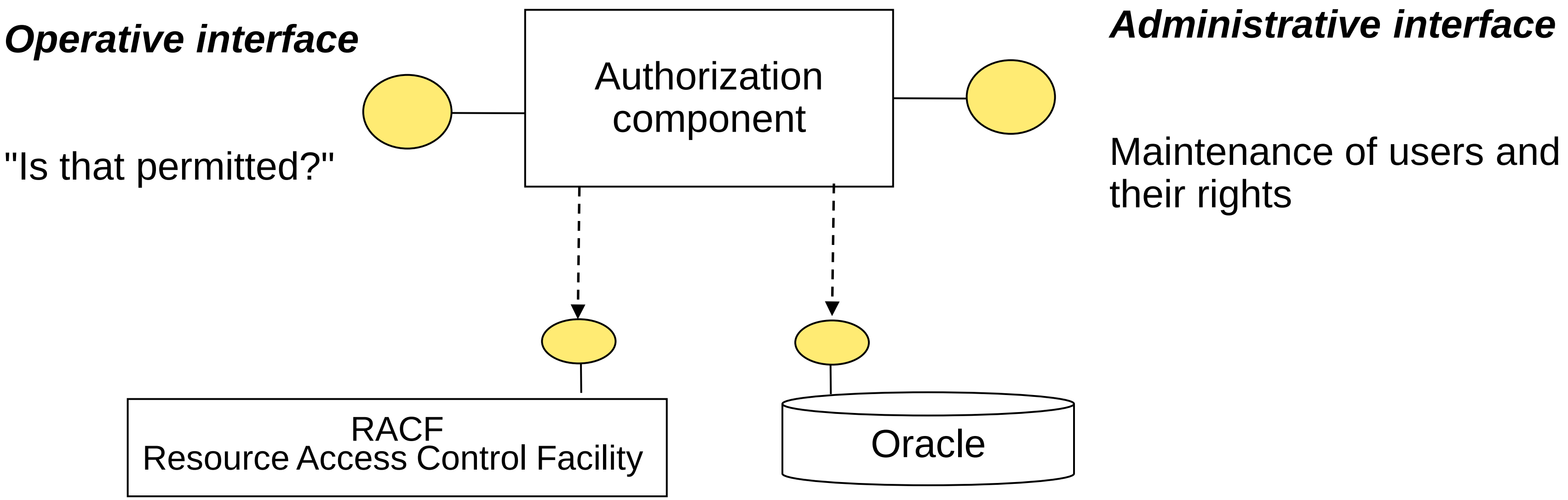
information hiding

zero redundancy
reuse

separation of concerns,
homogeneity,
software categories

planning,
staffing,
controlling

Example: different views of an authorization component



A component is only used through its interfaces

⇩

The interfaces of a component define the component

The external (“interface”) view of a component comprises more than the interface class

Constituent parts of an interface

Interfaces
- Services offered
- Call-backs required

Types
- Transfer objects
- (Business) data types
- Entity interfaces

Results
- Result types
- Error / exception classes

Contract
- Semantics of methods, parameters
- Pre- and post-conditions, invariants

Everything required to implement against the interface!

Components are used to construct subsystems, which are the largest building blocks of an application.

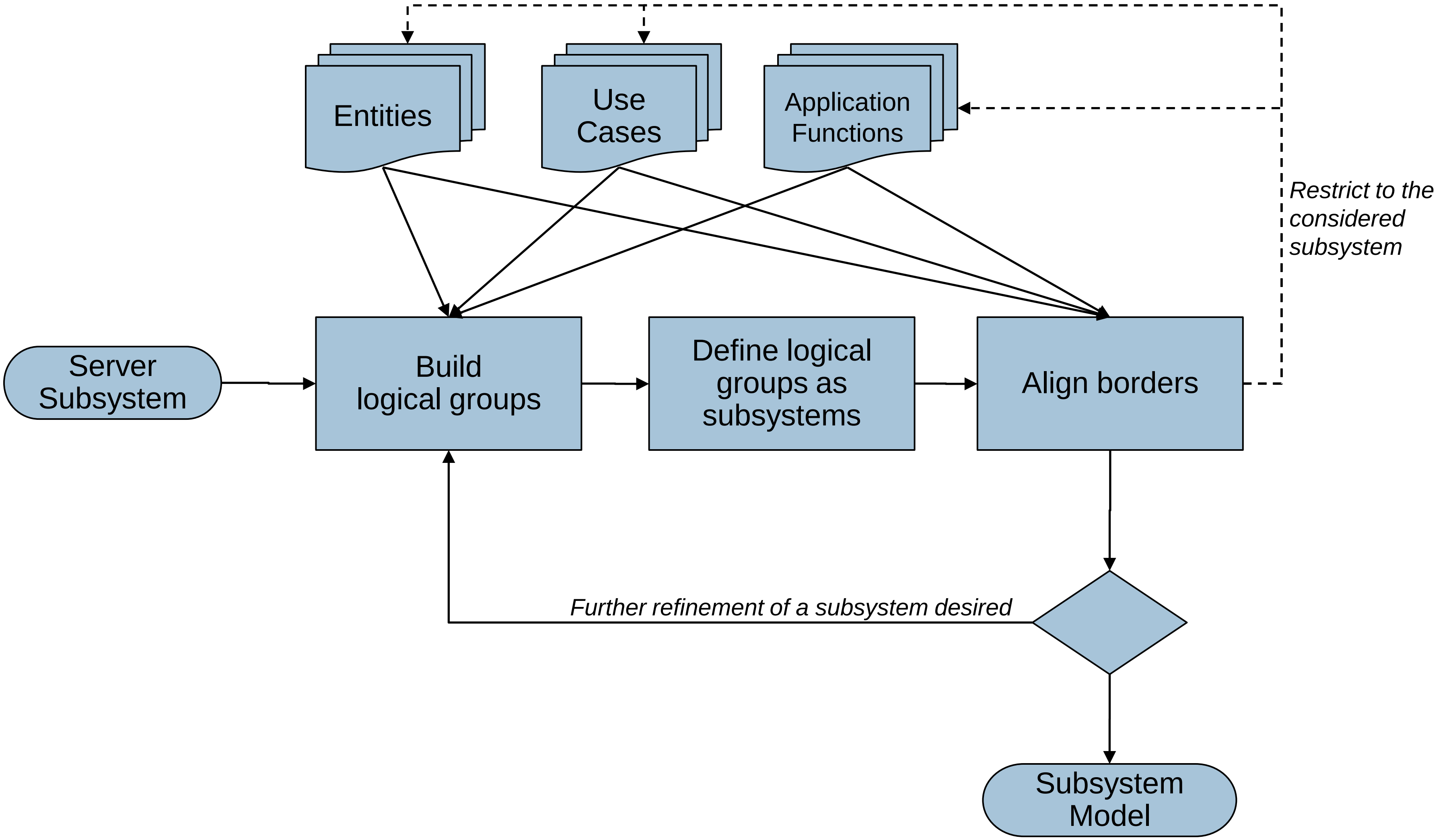
Subsystems – definition and properties

Subsystem

Definition	***A set of components or other subsystems which are grouped together to reflect logical criteria.***
Properties	• Contains no code, only components. • Serves as coupling area (more later).
Purpose	• Structuring the implementation (packages, projects,…). • Unit of planning, team building, and x-shoring. • Possible unit of deployment.

Use logical groups of entities, use cases and application functions to identify server side subsystems.

Identification server subsystems



Use logical groups of entities, use cases and application functions to identify server side subsystems.

Identification server subsystems

HANDOUT

1. Identify “the server” as one subsystem.
2. Ideally, a grouping of the business logic into something similar to subsystems should already have been done in the specification phase. If so, use this grouping as a starting point for the subsystems. If not, discuss the business logic with the business analyst to identify the first draft subsystems.
3. Assign each entity to a subsystem. If it cannot be meaningfully assigned to an existing subsystem, define a new subsystem.
4. Assign each use case and application function to a subsystem. If it cannot be meaningfully assigned to an existing subsystem, define a new subsystem. *Although a use case / application function may in the end be implemented by several components, there should be one clear subsystem that serves as its “anchor” or entry point.*
5. Evaluate the borders between subsystem – identify all entities whose assignment is disputable. Check if an additional subsystem introduced to contain them would contract other entities or use cases. If so, consider the introduction of the additional subsystem.
6. Evaluate each subsystem, if a further logical subdivision would be possible. If so, consider doing so (considering the size of the resulting subsystems). Continue with step 3 for each new (finer) subsystem.
7. In the end, the finest subsystems (leafs) should be of comparable complexity (as determined by the amount of entities and amount and complexity of use cases/application functions they contain). A subsystem should usually consist of several components. However, exceptions, when well motivated, are permitted.

How to identify the server components

Identification server components

Input	Component building
Entities	• Build logical groups • Assign group ownership to **data component** • Granularity: aim for ~ 5-20 entities / component
Use Cases **Application Functions**	• Assign to **data components** (if scope restricted to data component) • Assign to **logic components** (if functionality spanning entities of several data components)
Third-Party Interfaces	• **Adapter component** per third-party system • **Adapter component** per transformation / passed data structure (parameterized)
Reports	• **Reporting component** per subsystem (subsystems entities + public dimensions only) • **Reporting subsystem** (reports spanning operative entities from several subsystems)

How to identify the server components

Identification server components

HANDOUT

Per server subsystem:

1. Group the entities contained in the subsystem according to their logical relationship.
2. Identify loosely coupled or independent groups and assign their ownership to data components.
3. Assign use cases / application functions, where possible, to the data components. Such use cases / application functions should not modify any entity outside their assigned data component!
4. Assign use cases / application functions modifying entities from multiple groups to logic components.
5. Assign one adapter component per third-party system or unique transformation needed. If a single transformation can serve multiple third-party systems, make the adapter component configurable (third-party system as parameter).
6. Assign the report generation to reporting components:
    1. As reporting components per subsystem, if their reporting requirements can be fulfilled solely based on the entities of this subsystem (and possibly public dimensional entities from a master data subsystem).
    2. Consider to extract all reporting requirements into a separate reporting subsystem, if the reports combine operative entities from several subsystems.

How to identify server component interfaces

Identification server component interfaces

HANDOUT

Step	Action
Use Case Interface	• Assign a component interface to each use case which was assigned to that component
Interface Methods	• Declare method per application function of the use case (name, signature, types or abstraction, errors)
Auxiliary Interfaces & Methods	• Study use case / application function flow • Identify functionalities to be supplied by other components (not the one assigned to the use case) • Declare auxiliary interfaces and methods providing these functionalities
Verification	• Clean up interface structure, remove redundancy (e.g. use case, read-only, read-write interfaces) • Verify completeness using sequence diagram per use case / application function

How to identify server component interfaces

Identification server component interfaces

HANDOUT

Per server component:

1. Assign an interface to each use case that was previously assigned to the component.
2. Assign one method in the interface identified in step 1 per application function implementing the use case. Identify the method signature necessary to implement the logic.
3. Identify auxiliary interfaces and methods:
    1. Per method identified in step 2, identify which calls to *other* components are necessary for its implementation.
    2. Identify the method signature for these methods.
    3. Assign the methods to auxiliary function interfaces of the components.

Verification of the results:

1. It is recommended to achieve a structure distinguishing the component interfaces according to their scope:
   interfaces assigned to a specified use case – auxiliary read-only interface – auxiliary interface with write access
2. Per use case / application function, verify the implementation of the use case flow in terms of the component interfaces by showing this flow in a sequence diagram (where the swim lanes are component interfaces).

Software categories help with component breakdown

Idea

Areas of expertise

- Subdivide software according to “area of expertise”
- “Expertise” = an aspect (technical, business) of the problem
- Area of expertise = software category

Refinement

- Coarse-to-fine specialization of categories
- Finer level = narrower scope

Structuring criterion

- Refined categories serve as a structuring criterion to components
- Components should not mix categories

Procedure: First analyze the software categories and then break the system down into components on this basis

Basic software categories: software blood groups

Combinations:

A + 0 = A
T + 0 = T
A + T = AT

Categories	That means...
0 software	Independent of application and technology; ideal for reuse; example: class library for strings and containers
A software	Determined by the logical application; independent of technology; generally the largest part of the system; example: employees, booking
T software	Independent of the logical application; Reusable if the same technical component is used; example: database access layer
AT software	Concerned with technology and application; difficult to maintain; resists changes; reuse as likely as winning the lottery!
R software	Pure transformation; example: screen format in XML
C software	Configuration: brings the different categories together (main)

Break a system down into components in such a way as to achieve strong cohesion and loose coupling

Ed Yourdon

Component break-down

Strong cohesion

Within a component (cohesion = holding together)

- Put elements with related content into a **single** component
- Create a clearly defined, precise area of responsibility
- Simple components should belong to a **single software category**

Loose coupling

Between components

- Minimize dependencies (narrow interfaces, few imports, etc.)
- Minimize assumptions about other components (e.g. special data formats)

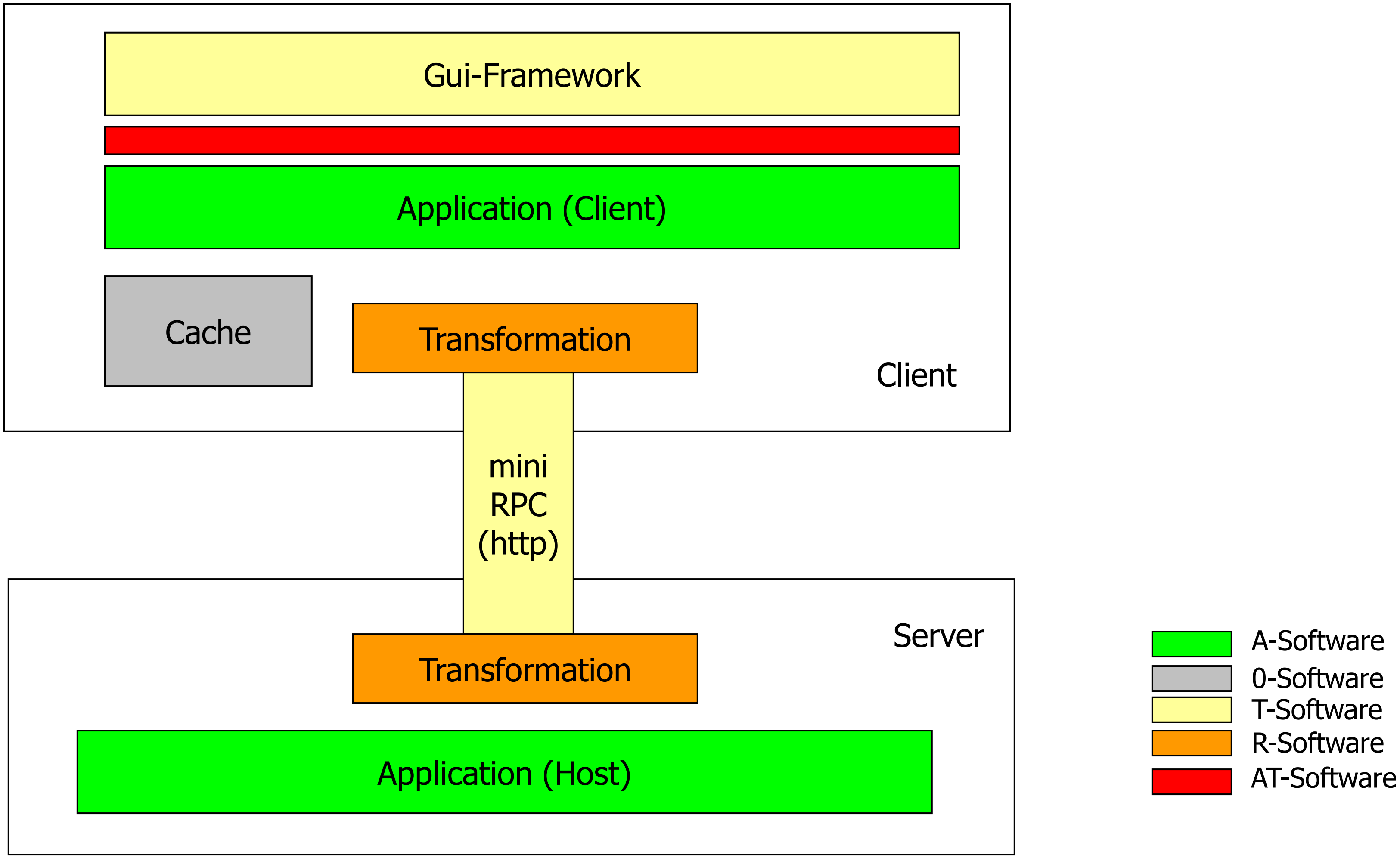
Typical errors when breaking down into components

Error manifestations

Error	Manifestations
Mixing of A and T	• Components depend on T software and implement business logic • The components need to be adapted when either changes
God components	• Components are responsible for more than one thing • Components belong to more than one software category • The purpose of a component cannot be described in a short, memorable text
Distributed responsibilities	• A responsibility is distributed across multiple components • Components cannot be developed separately from each other
Spaghetti design	• Dependencies have not been reduced

Software categories, example for a 3-Tier native client

HANDOUT



Using Transport Objects to communicate with the application core

Definition:
A **Transport object** (**TO**) is a container which transports data across layers and processes. This includes physical layers (communication between systems). This concept reduces method calls and decouples components.

Referenzen

- J2EE pattern: "transfer object"
 http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html
- "Data transfer object": Fowler
 (Fowler's "value objects" equate to our "A data types")
- DTOs are special types of the value object pattern (often used synonymously)
 (Pattern comes from EJB: The structure of a value object is based on an entity bean)

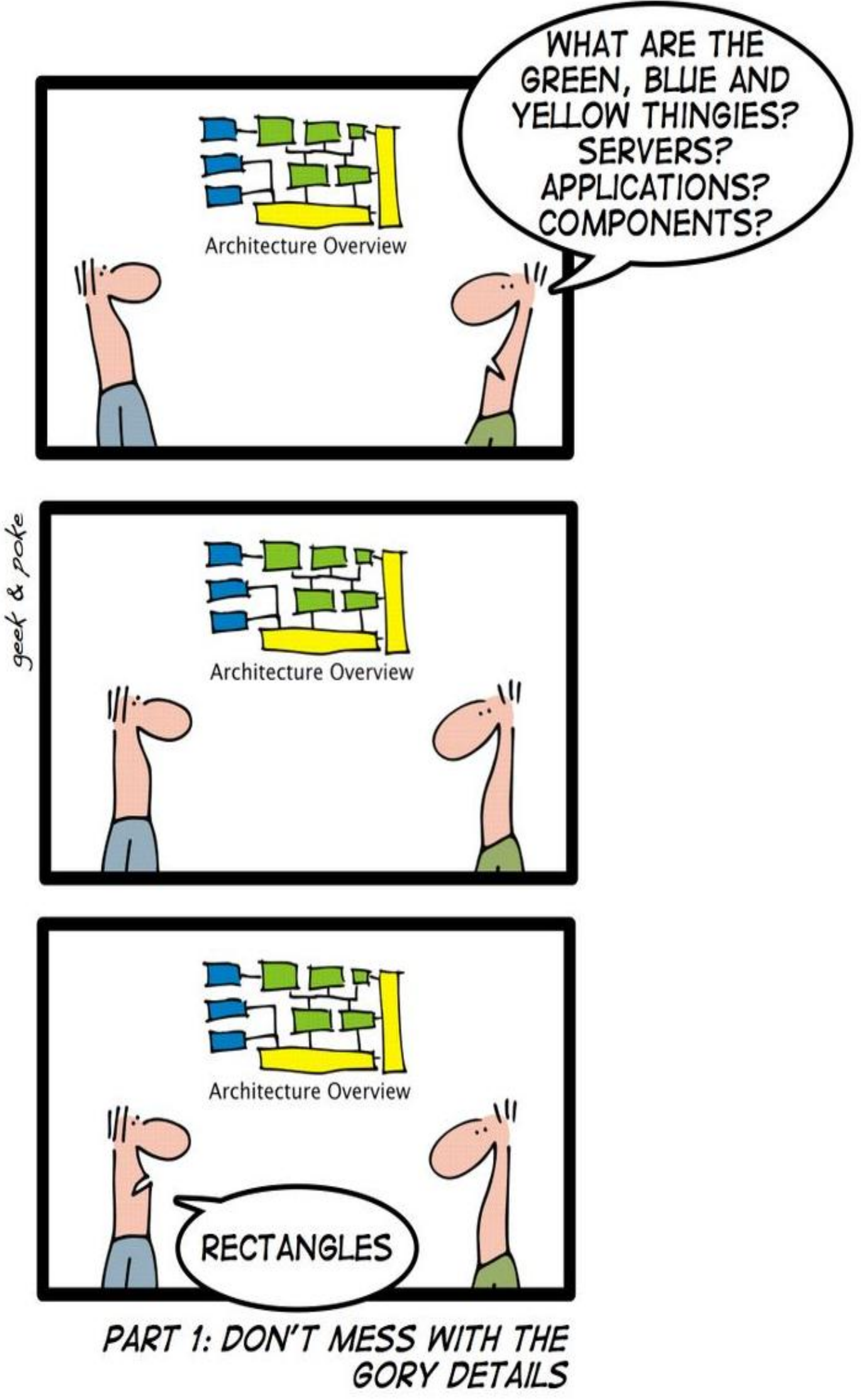
Why transport objects instead of entities?

Using TOs ⟷ Using entity types

separation through transport objects	only entity types
⊕ decouples components. Changes in the model do not affect the interface.	⊕ Easier, less effort.
⊕ Higher control, which data is transferred to the caller.	⊕ Can be generated from the model
⊕ additional aspects (metadata) can be added to the TO.	⊖ Higher coupling
⊖ Higher effort for creating TOs and mapping data.	⊖ eventually, entities will be decorated with TOs anyway.
⊖ Lower performance (because data is copied)	⊖ less control of which data is transferred
	⊖ less secrecy for the internal structures of the component. E.g. relationships can be misused.

AGENDA

- What is the purpose of software design?
- How to get from specification to construction?
- How do you produce good designs?
- The architectural principles
- Thinking in terms of components and interfaces
- **Architectural views and layers**
- Design best practices



via http://geekandpoke.typepad.com/geekandpoke/

Quasar architectural views:
A/T separation at the architectural level

Software architecture: components and interfaces

A architecture (application architecture)

- Free of technical, product-related practical constraints
- Is developed anew for each project
- Structures the software from the perspective of the application
- Contains logical classes such as "Employee" or "Account"

A components

T architecture (technical architecture)

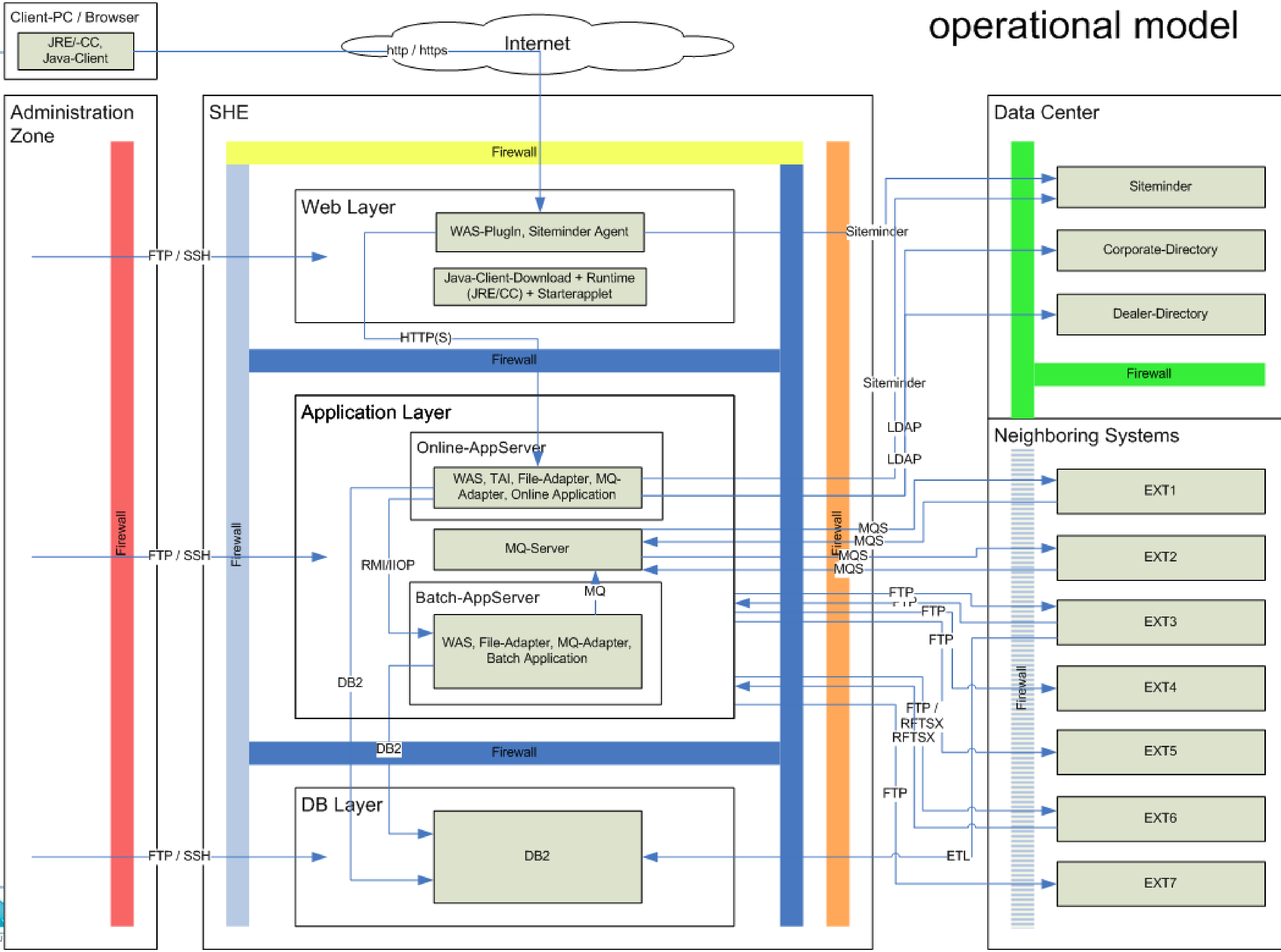
- Describes the "virtual machine" on which the software designed with the A architecture runs (container for A components)
- Template for A code
- Combines A and TI architecture
- Structure of the 0/T code
- Reuse possible

T components

TI architecture (architecture of the technical infrastructure)

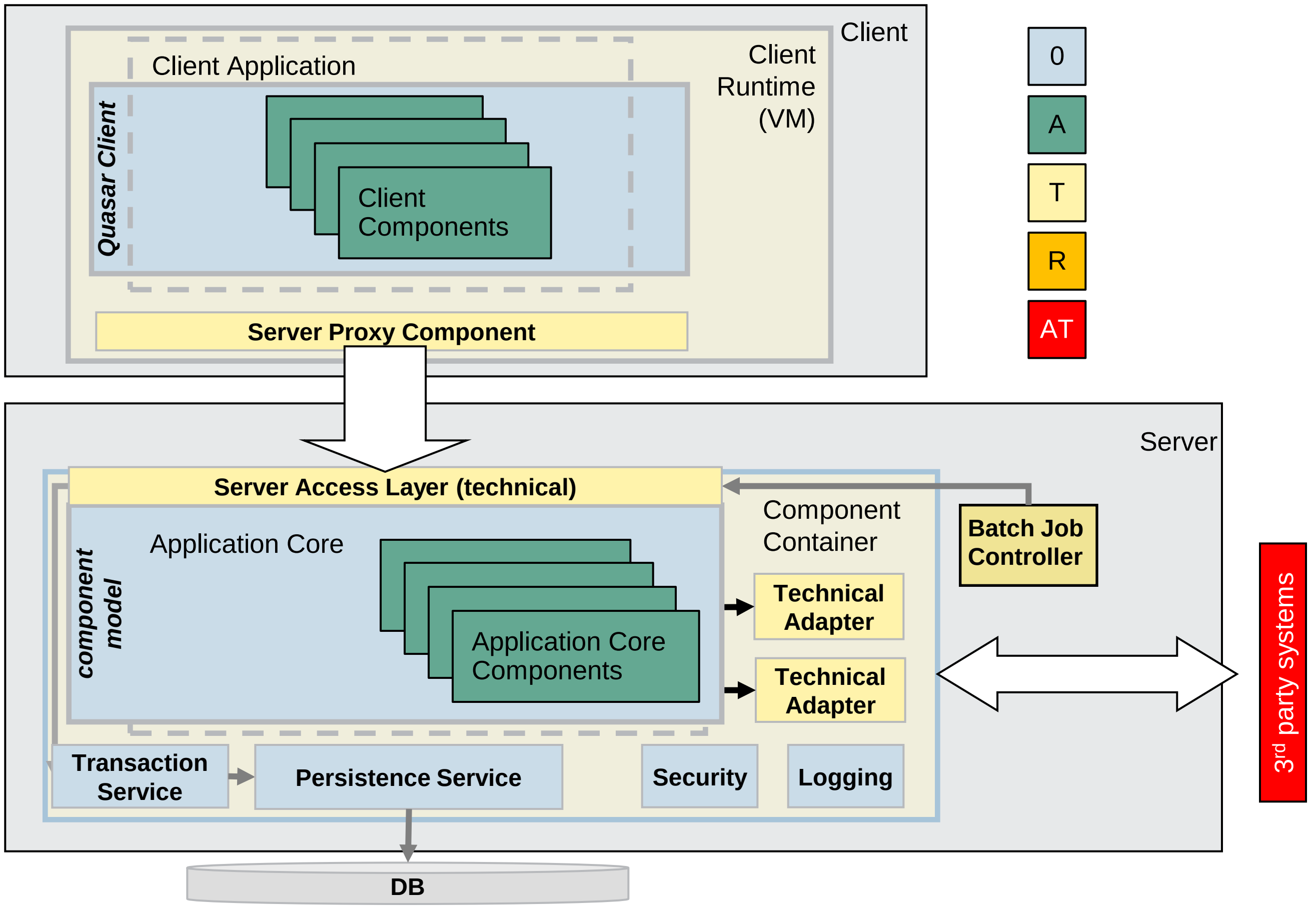
- Physical devices (computers, network cables, etc.)
- System software (operating system, DBMS, application server, etc.)
- Interaction of the hardware with the system software installed on it
- Programming languages used
- Products with versions

Example for TI architecture



The T architecture defines how we structure an application and its A components from a technical point of view

Standard Quasar T architecture (high level view)*



The usual three layers

- Dialog
- Application core
- Persistence management

T components

- Batch Job Controller
- Server Access Service (Server Proxy Component and Server Access Layer)
- Authorization
- Transaction management
- Logging
- Technical adapters for 3rd party system integration

The A architecture defines how we structure an application from a business point of view, how we subdivide it in order to handle complexity

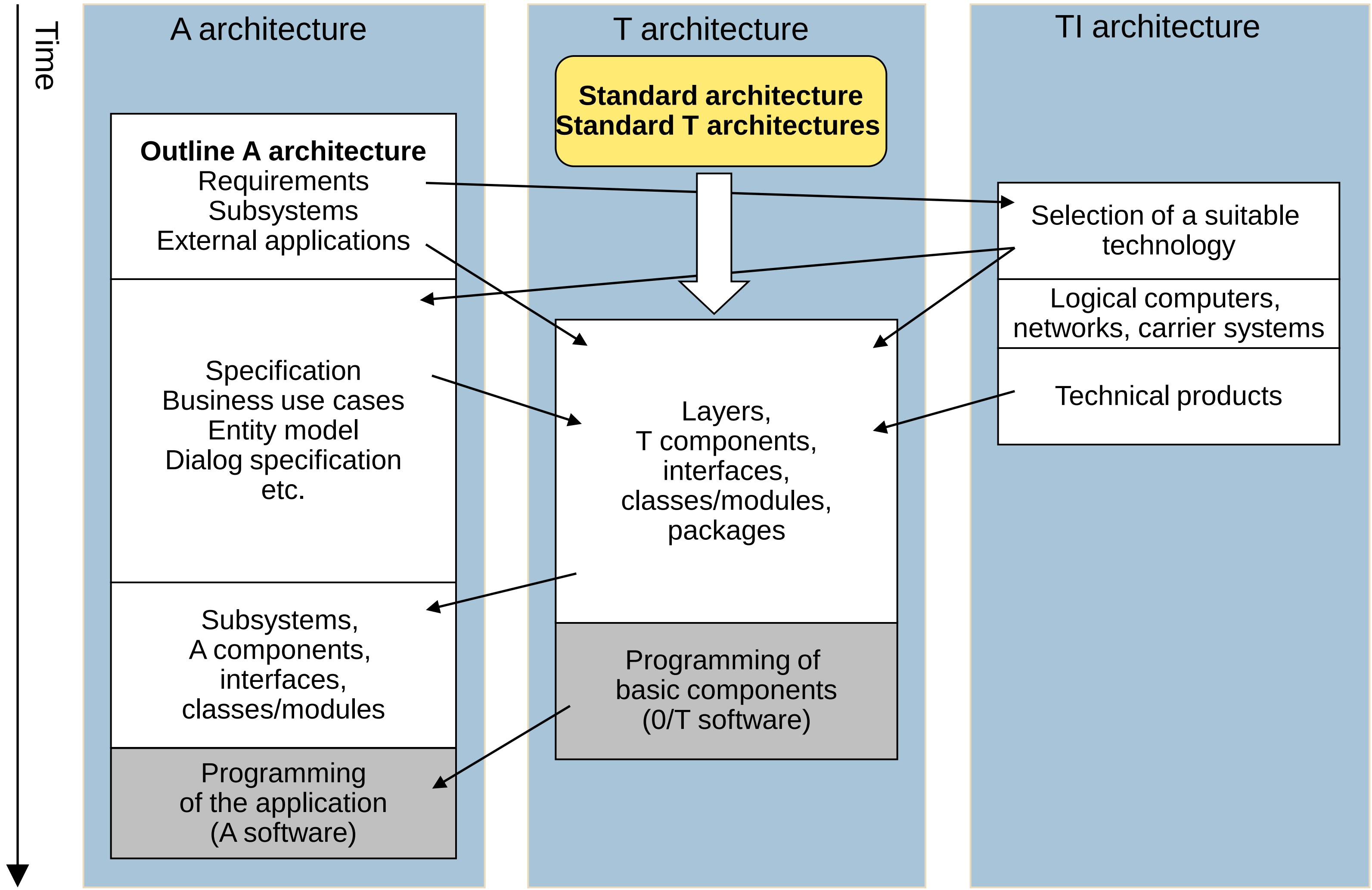
A architecture

- The A components are the central entities for planning and implementing software systems
- A components are implementations of the “templates” found in the T architecture

Use of the view

- The construction plan of the system
- Understand the system
- Control dependencies
- Decide where to put new functionality
- Define data ownership

Architectures in the project timeline: Parallelization of development



Architectures in the project procedure: Parallelization of development

HANDOUT

- The three views of the architecture are developed in parallel.
- There is no simple assignment of an architectural view to a project phase:
  - The outline A architecture and TI architecture emerge during specification.
  - The CON phase begins with the T architecture. The A and TI architectures are further refined in this phase.
  - The A and T architectures lead finally to the implementation of the technical artifacts.
- The various architectural views influence each other:
  - The logical requirements determine the hardware and software described in the TI architecture.
  - However, within certain limits the TI architecture can also influence the A architecture (for example, when the customer has stipulated a certain technology: A dialog for a web client, for example, must be different from a dialog for a native client, etc.)
  - The T architecture is based on the specifications of the TI architecture.
  - The T architecture must provide the framework in which the elements of the A architecture can be built. The two views therefore influence each other:
    - A -> T architecture: requirements to be met by the T architecture framework
    - T -> A architecture: Requirements of the T architecture with regard to the elements to be created within this framework

Summary for architecture views

A architecture (application architecture)	T architecture (technical architecture)	TI architecture (architecture of the technical infrastructure)

• Overview of the business aspects without technical detail.	• Generic view of technical architecture.
	• Serves as a bridge from application architecture to technical infrastructure
		• Focuses on operational aspects
		  • products
		  • distribution
		  • communication

AGENDA

- What is the purpose of software design?
- How to get from specification to construction?
- How do you produce good designs?
- The architectural principles
- Thinking in terms of components and interfaces
- Architectural views and layers
- **Design best practices**

Design best practices

Strive for simplicity

- **K**eep **I**t **S**mall and **S**imple
- **M**ake it **E**asy

Base yourself on the requirements

Work iteratively

Best practice: Strive for simplicity

HANDOUT

- "If you can't explain it in five minutes, either you don't understand it or it doesn't work." (Mark W. Maier & Eberhardt Rechtin, The Art of Systems Architecting)
- "You ain't gonna need it." (YAGNI approach) - no abstraction in advance
- "Everything should be made as simple as possible, but not simpler" (Albert Einstein)
- "Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away." (Antoine de Saint-Exupéry)

KISS ME

- **K**eep **I**t **S**mall and **S**imple
- **M**ake it **E**asy

Select a design that is as simple as possible, but just complex enough to solve the problem.

Best practice: Base yourself on the requirements

HANDOUT

- Know the business logic
- Know the prioritization of the requirements
- Base every design decision on concrete requirements
- Call difficult requirements into question sometimes
- Make the design objectives explicit - and evaluate trade-offs between project and architectural objectives (e.g. development costs versus maintainability)

Best practice: Work iteratively

HANDOUT

- Nobody comes up with the right design immediately 100% of the time
- Work in small steps
  - "From coarse- to fine-grained"
  - "From outside to inside"
- Schedule time for final polishing and refactoring
- <u>Actively</u> seek feedback

Design best practices

Avoid redundancy

Maintain a uniform style

Document your decisions

Further best practices...

HANDOUT

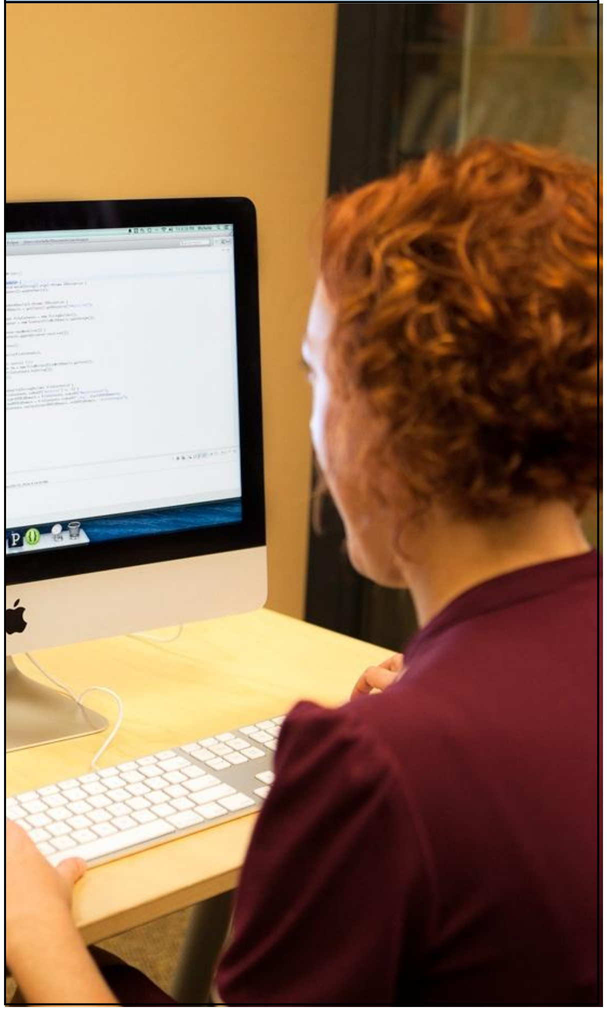
- Avoid redundancy
  - DRY – Don't repeat yourself
- Avoid dependencies of details
  - Encapsulate behaviour in interfaces
  - Do not access the implementations directly
- Ensure lean interfaces
  - Dependencies only on the things that are really used

- Maintain a uniform style
  - Solve similar problems in the same way as far as possible

- Document your design decisions
  - Document what you decide, and why
  - Also document which alternatives where not chosen, and why

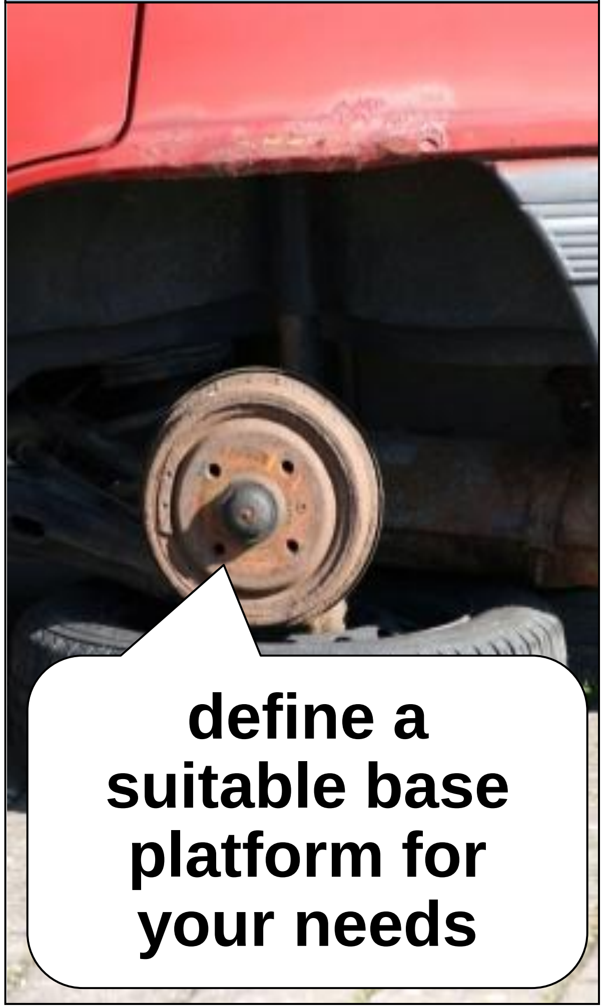
Design best practices

Make your design testable



Allow yourself to be helped

Don’t reinvent the wheel



Further best practices...

HANDOUT

- Make your design testable and test-driven
  - Think about how you can test the different parts of the system well in advance

- Allow yourself to be helped
  - Nobody knows everything
  - Make use of your colleagues' experience
  - Communities within company
  - Ask a knowledge specialist
  - Use the internet

- Don't reinvent the wheel
  - Use documented knowledge and proven solutions
  - Use design and architecture patterns
  - Use standard software architectures and interfaces

Software complexity has increased drastically in the last decade

Mainframe
Transaction monitor
Database

Technology stacks 80er/90er

- homogenious, primarily driven by a few software vendors

Mainframe
Unix/Linux
Process Server
Applikations-Server
Enterprise Service Bus
Portal Server
Persistenz-Framework
WebServices
Database
Frameworks

Technology stacks today

- complex, distributed, many vendors

Software complexity has increased drastically in the last decade

Mainframe

Mainframe

Unix/Linux

more on Capgemini's
Open Application Standard Platform
in the upcoming lectures

Overview of lecture topics

No.	Date	Subject
1	16.10.2015	Introduction
2	23.10.2015	Different Application Lifecycles for different needs
3	30.10.2015	Estimation and Project Management
4	06.11.2015	Requirements Management
5	13.11.2015	Specification
6	20.11.2015	Quality Management
7	27.11.2015	— TechnoVision Workshops —
8	04.12.2015	Designing Software Architectures Part 1/2
9	11.12.2015	Designing Software Architectures Part 2/2
10	18.12.2015	Modern Technology Stacks
11	15.01.2016	Creating and Maintaining Open Platforms
12	22.01.2016	— Workshops —
13	29.01.2016	Model Driven Development
14	05.02.2016	Performance, Operations and Continuous Integration
15	12.02.2016	Enterprise Architecture

Christmas break

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Appendix: Developer Manual and Code Review

Developer Manual – Table of Contents

- Installation of developer tools (IDE, build management, repository, etc.)
  - configuration (plugins, etc.)
  - use of prebuilt/preconfigured tools
- Installation of runtime environment (Application Server, Database, etc.)
  - configuration
- Checkout of sources
- building
- initial installation in runtime environment
- typical use cases during development
- coding conventions
  - use of auto formatting
  - how to write tests (coverage, setup, etc.)
  - naming conventions

Code reviews workflow

- preview: architect gives in introduction to the work package
- development: developer writes code and tests
- review: architect (or experienced developer) checks the code
  - Does it match the specification, is anything missing?
  - Are the important parts covered by tests?
  - Are the tests checking relevant edge cases and/or is the code handling them correctly?
  - Are code conventions followed?
  - Is there any code smell (e.g. checked by findbugs, PMD, etc.)
  - Was or is refactoring of existing code necessary?
  - Are architectural conventions followed (software categories, layers, components, etc.)
  - comments, logging, performance, other non functional requirements
  - are other artefacts updated correctly (documentation, database scripts, etc.)
- review comments are then addressed by the developer

Capgemini

CONSULTING.TECHNOLOGY.OUTSOURCING

Together. Free your Energies.

TOGETHER. FREE YOUR ENERGIES.

COLLABORATIVE BUSINESS EXPERIENCE

Research and science live on the exchange of ideas, the clear arrangements are thereby useful:

Disclaimer:
Although this presentation and the related results were created carefully and to the best of author's knowledge, neither Capgemini nor the author will accept any liability for it's usage.

If you have any questions, please contact:
Capgemini | Offenbach
Dr. Martin Girschick
Berliner Straße 76, 63065 Offenbach, Germany

Telephone +49 69 9515-2376
Email: martin.girschick@capgemini.com