Advanced Topics of Software Engineering (ASE)

Chapter 2.1: Architecture, Modularity, Component-based SE

Prof. Dr. Florian Matthes, Prof. Dr. Alexander Pretschner

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

From requirements to system design

Software architecture

There are many definitions of "software architecture", see, e.g.
http://www.sei.cmu.edu/architecture/start/definitions.cfm

"The fundamental organization of a system embodied in its ***components***, their ***relationships*** to each other, and to the ***environment***, and the ***principles guiding its design*** and ***evolution***."

[ISO/IEC 42010]

"The structure of the ***components*** of a program/system, their ***interrelationships***, and ***principles and guidelines*** governing their ***design*** and ***evolution*** over time."

["Introduction to the special issue on software architecture." Garlan D., and Perry D.E. (1995)]

"Software architecture is a set of ***functional decompositions*** of a software system according to ***non-functional requirements*****."**

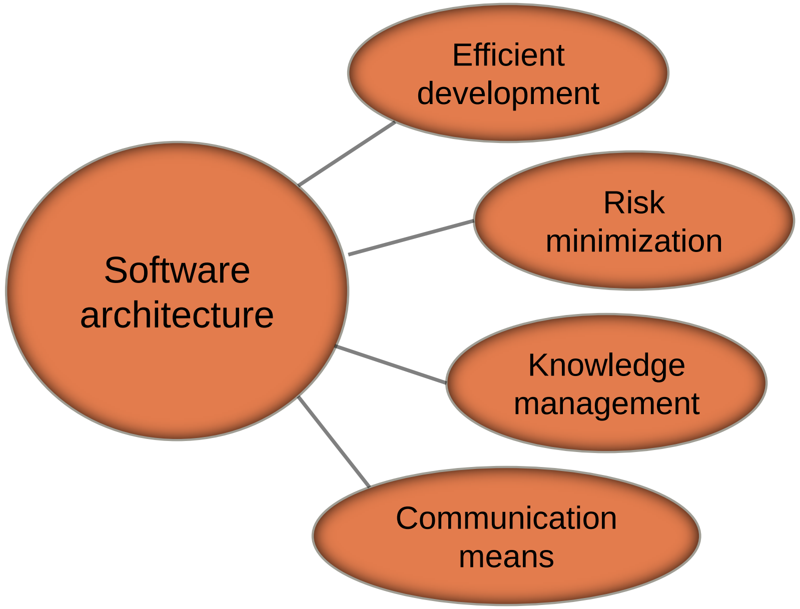
Newer perspective

Architecture is a set of significant ***design decisions*** that shape a system.

[Kruchten P. (2004), Bosch J. (2004), Booch G. (2006)]

Software architecture as a planning document

- Software architecture influences the success of a software project heavily
- It enables a systematic planning of **large** software systems to satisfy defined quality requirements
- The main influence of a software architecture concerns project management



> "Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled."

[Eoin Woods (SEI 2010)]

The purpose of software architecture

Quality of the resulting system!

Efficiency of the development process

- Decoupling of tasks enables separation of labor and a flexible organization of the project.
- Software architecture defines a structure which enables iterative and incremental development.

Risk Minimization

- Risks are described and resolved early.

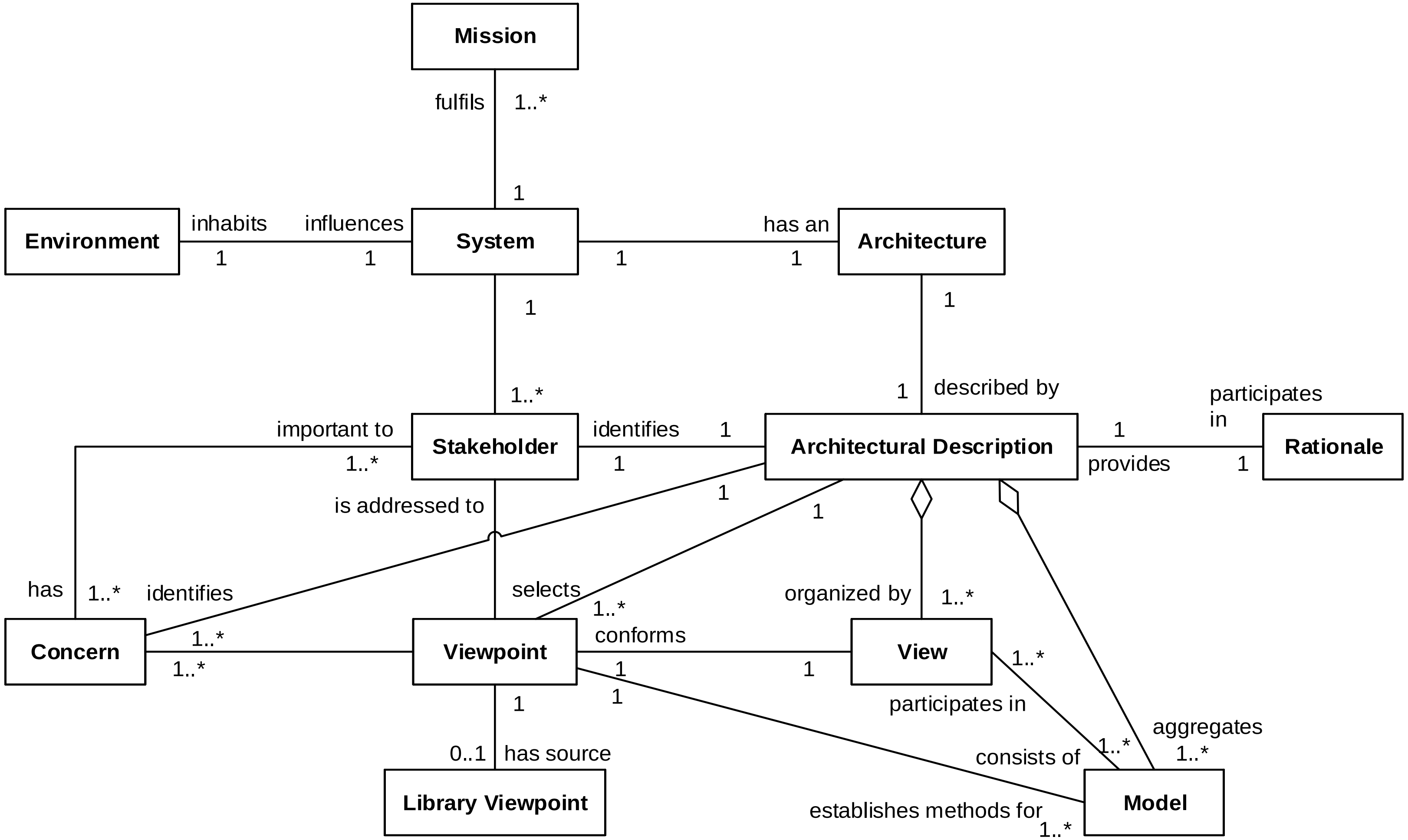
Communication Means

- Transfer of requirements and expectations between customers, project leaders, developers, testers, users, maintenance staff.
- Things can be checked early on the architecture, misunderstandings are mitigated.

Knowledge Management

- Explicit representation as documents
- Planning of evolution
- Planning of reuse
- Helps to incorporate new staff quickly

Architecture in software intensive systems (1)



http://www.iso-architecture.org/ieee-1471/cm/

[ISO/IEC/IEEE 42010]

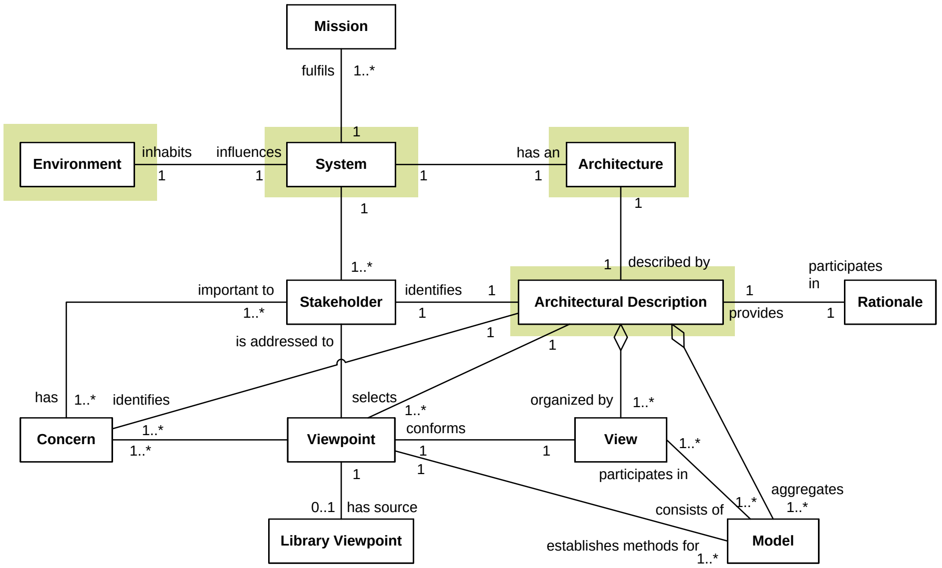
Architecture in software intensive systems (2)

System

- A **collection of components** organized to accomplish a specific function or set of functions.
- The term system encompasses individual applications, systems in the traditional sense, subsystems, systems of systems, product lines, product families, whole enterprises, and other aggregations of interest.
- A system (a) has defined **boundaries**, (b) consists of **components** and **interfaces**, (c) interacts with its **environment** through these interfaces, (d) is defined by its **static structure** and its **dynamic behavior**.

Environment

- Developmental, operational, political, and other influences upon that system.



Every system has an **architecture**, which can be recorded by an **architectural description**.

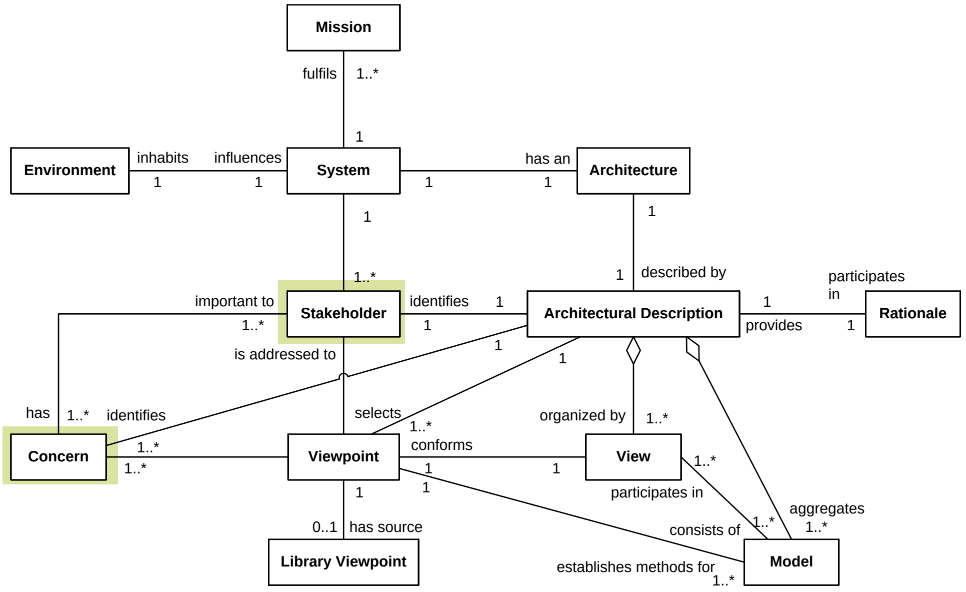
Architecture in software intensive systems (3)

Stakeholder

- Each stakeholder typically has interests in, or concerns relative to, that system.
- Have various roles with regard to the creation and use of architectural descriptions.
- Two key roles among stakeholders are the principal and the architect.

Concern

- An interest which pertains to the system's development, its operation, or any other aspect that is critical or otherwise important to one or more stakeholder
- Includes system considerations such as performance, reliability, security, distribution, and evolvability



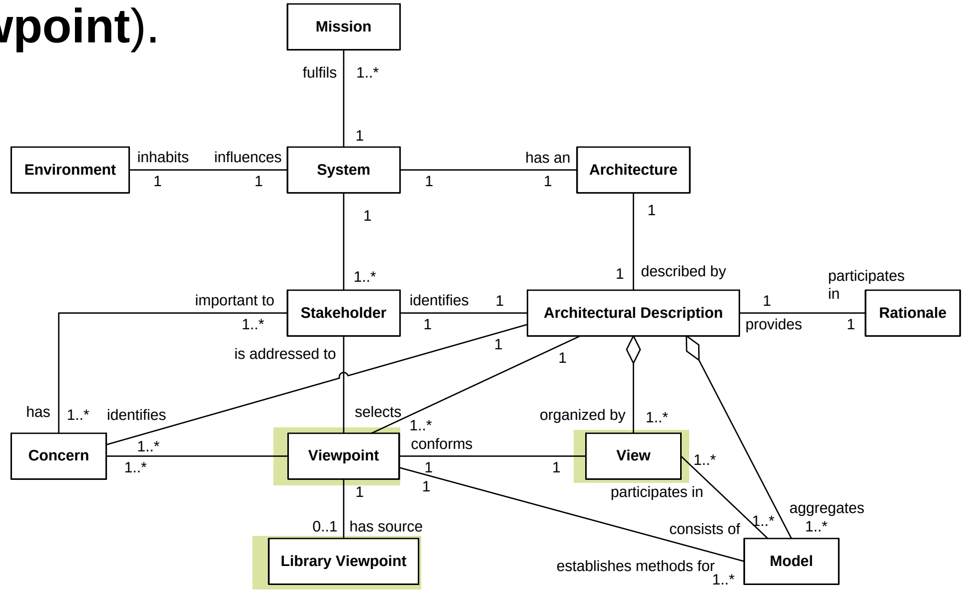
Architecture in software intensive systems (4)

View

- Addresses one or more of the concerns of the system stakeholders.
- The term view is used to refer to the expression of a system’s architecture with respect to a particular *viewpoint*. A view conforms to a viewpoint.

Viewpoint

- Determines the languages (including notations, model, or product types) to be used to describe the view, and any associated modeling methods or analysis techniques.
- A viewpoint may use viewpoints defined elsewhere (**library viewpoint**).



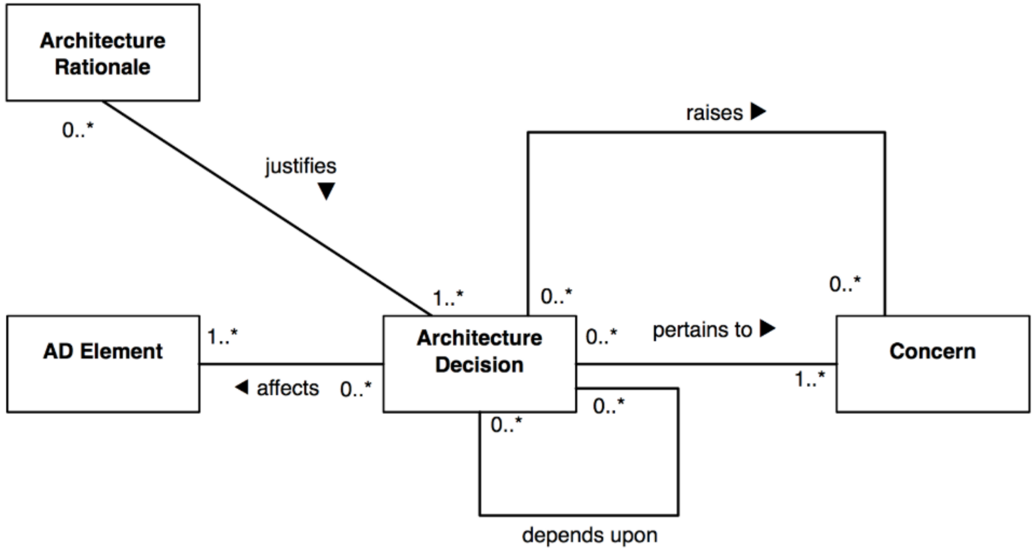
Architecture in software intensive systems (5)

Architectural decisions and rationale

"A description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture."

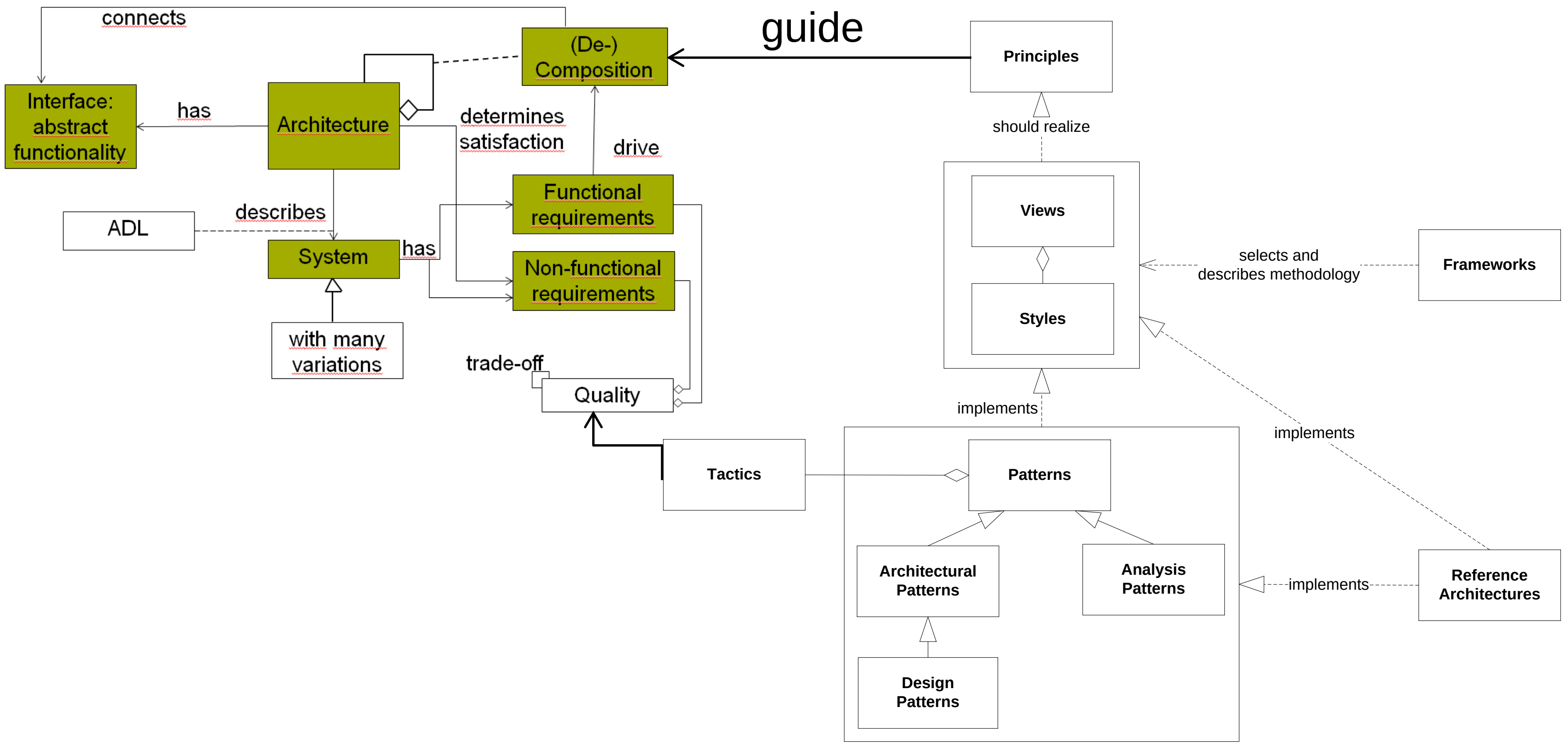
["Software architecture as a set of architectural design decisions. ", Jansen, A., & Bosch, J. (2005)]

- **Architecture rationale** records the explanation or justification about architecture decisions that have been made and architectural alternatives chosen or not chosen.
- Architecture Descriptions (AD) element – any item in AD (stakeholder, view, viewpoint, model…)



An architecture decision affects AD elements and pertains to one or more concerns.
By making a decision, new concerns may be raised.

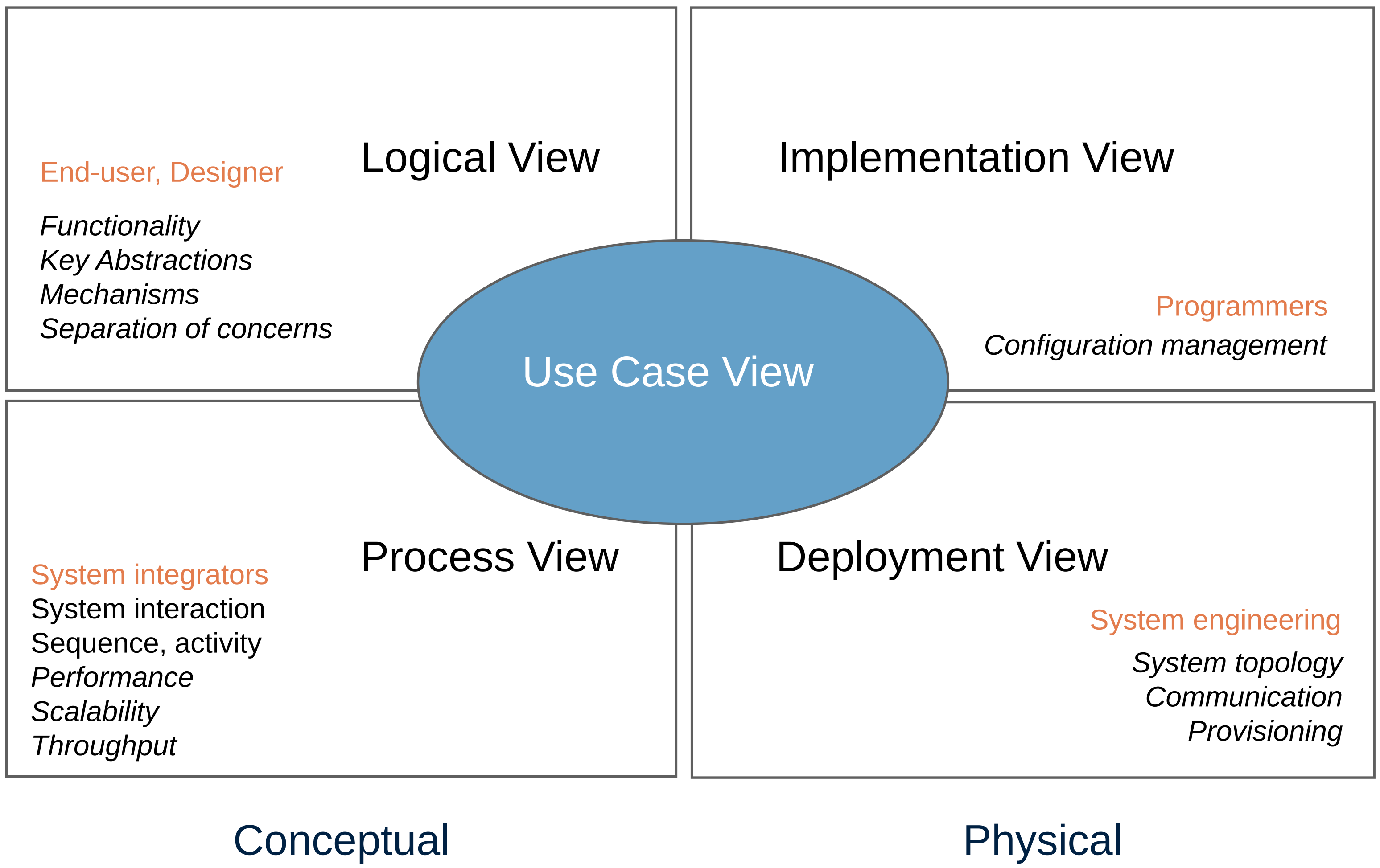
Another perspective



[“Software architecture” Pretschner A.]

An example of architecture viewpoints

4+1 view model



["The 4+1 View Model of Architecture." Kruchten, P. (1995)]

An example of architecture viewpoints

4+1 view model

4 Describes software architecture using five viewpoints

- The **logical viewpoint** describes the design's functionality from an end user perspective
- The **process viewpoint** describes the design's dynamic communication, concurrency and synchronization aspects
- The **deployment viewpoint** describes the mapping of the software onto the hardware and reflects its distributed aspects
- The **implementation viewpoint** describes the software's static organization in its development environment

+1

- **Scenarios** are used to show that the elements of the four views work seamlessly together. The scenarios are in some sense an abstraction of the most important requirements

["The 4+1 View Model of Architecture." Kruchten, P. (1995)]

Adapting viewpoints

Not all systems require all viewpoints

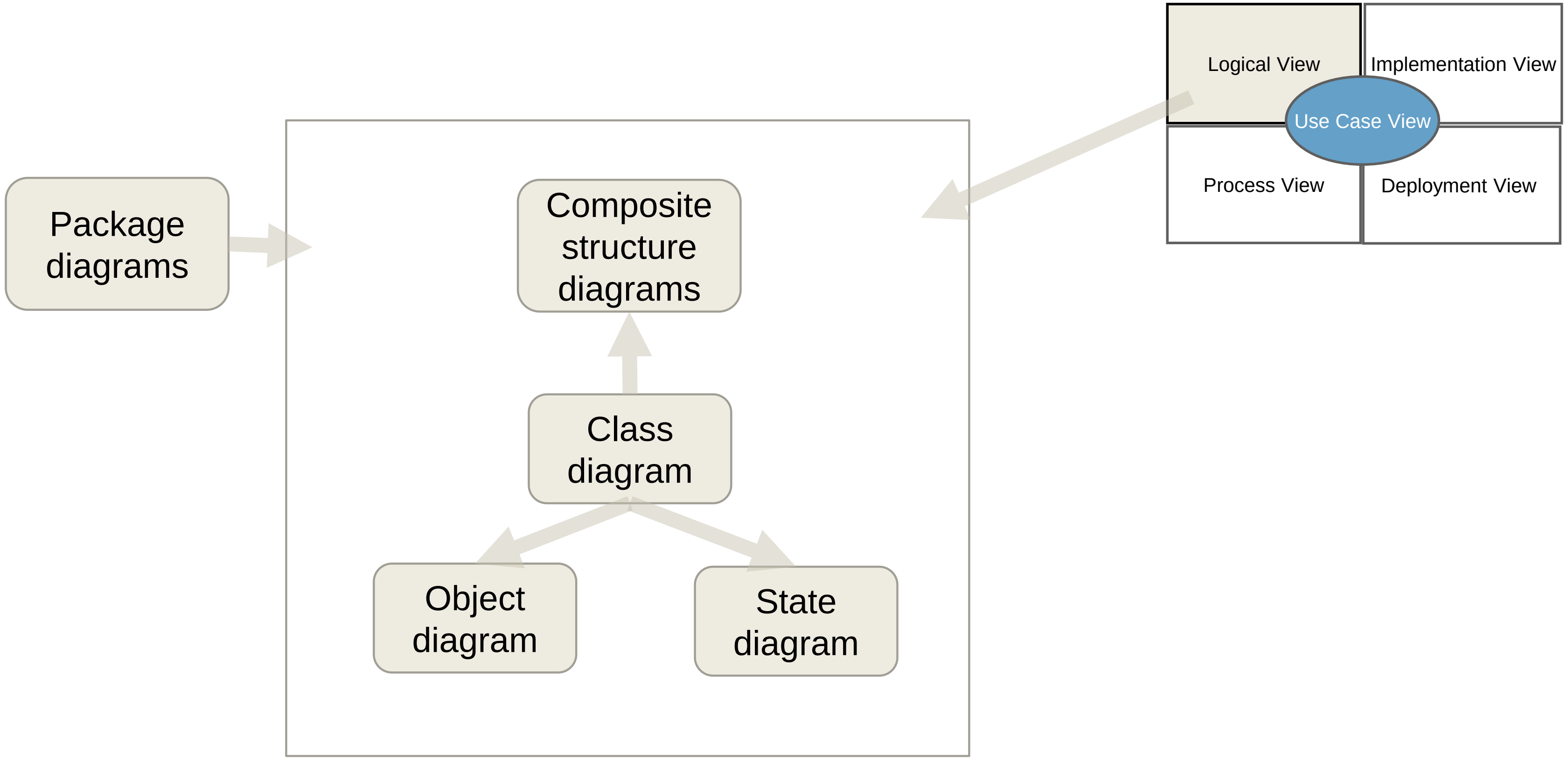
- Single process (ignore process viewpoint)
- Small program (ignore implementation viewpoint)
- Single processor (ignore deployment viewpoint)

Some systems require additional viewpoints (not very common):

- Data viewpoint
- Security viewpoint
- Other aspects

Modeling 4+1 views with UML 2

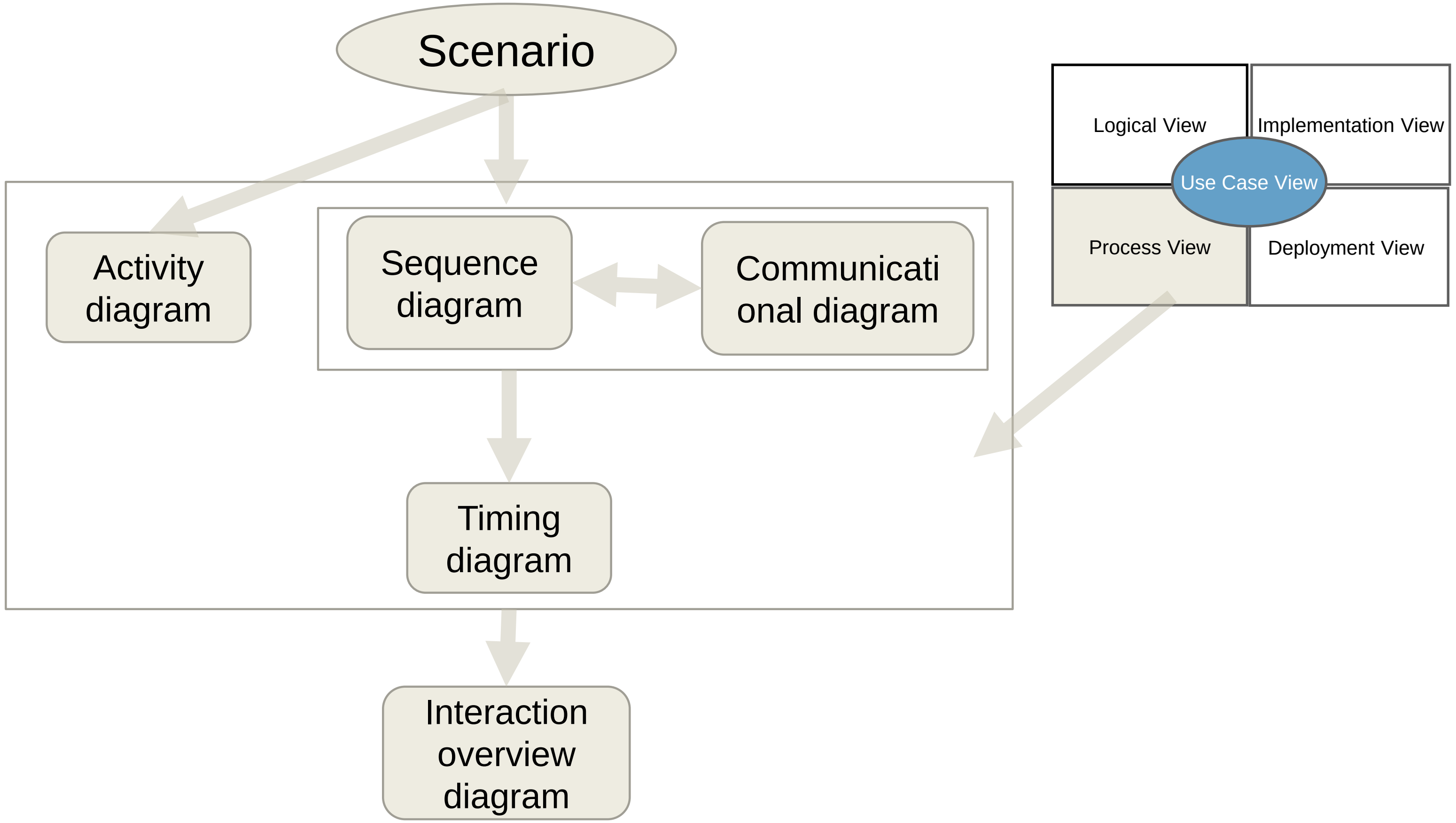
Logical view



["Applying 4+1 View Architecture with UML 2." Muchandi V.]

Modeling 4+1 views with UML 2

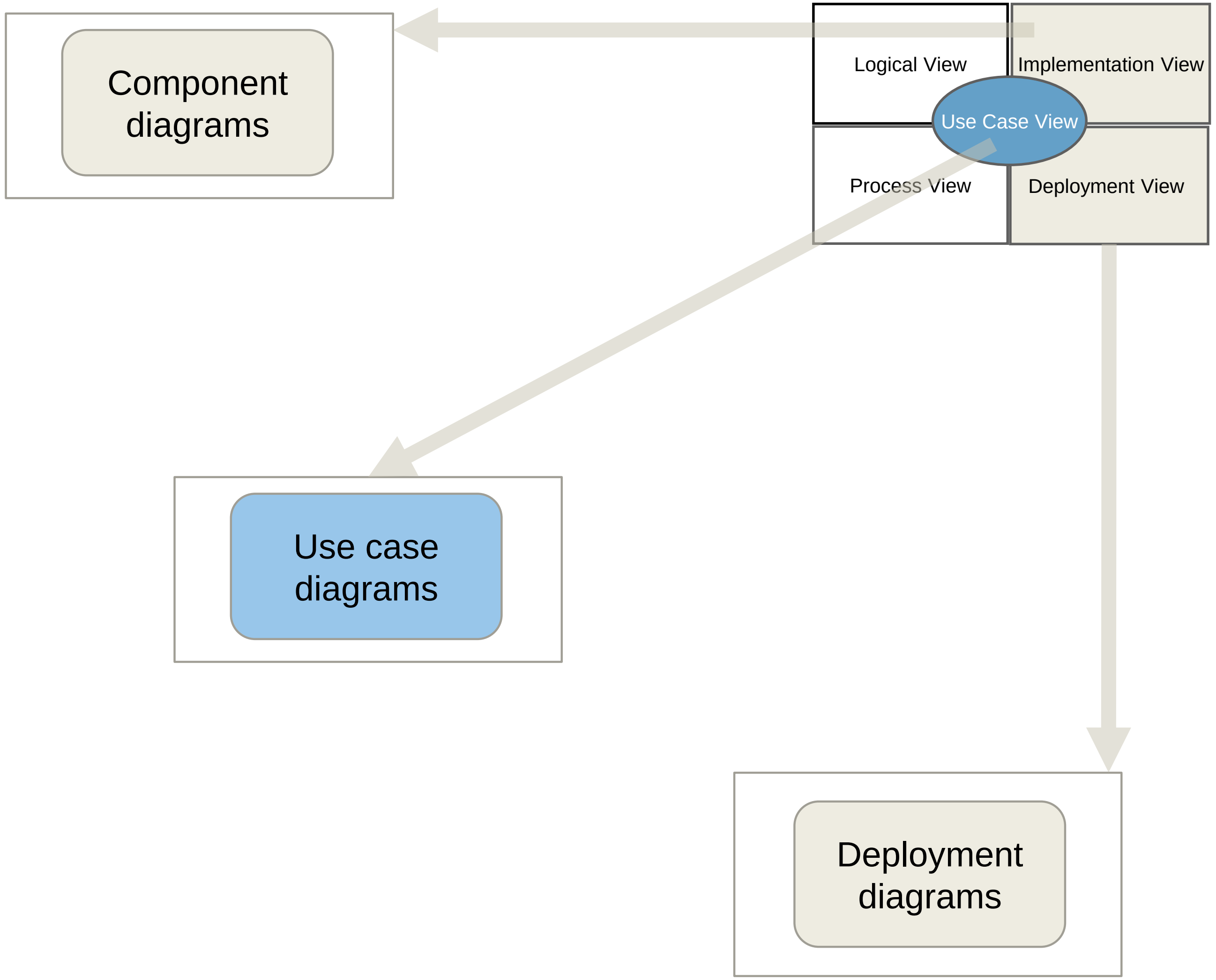
Process view



["Applying 4+1 View Architecture with UML 2." Muchandi V.]

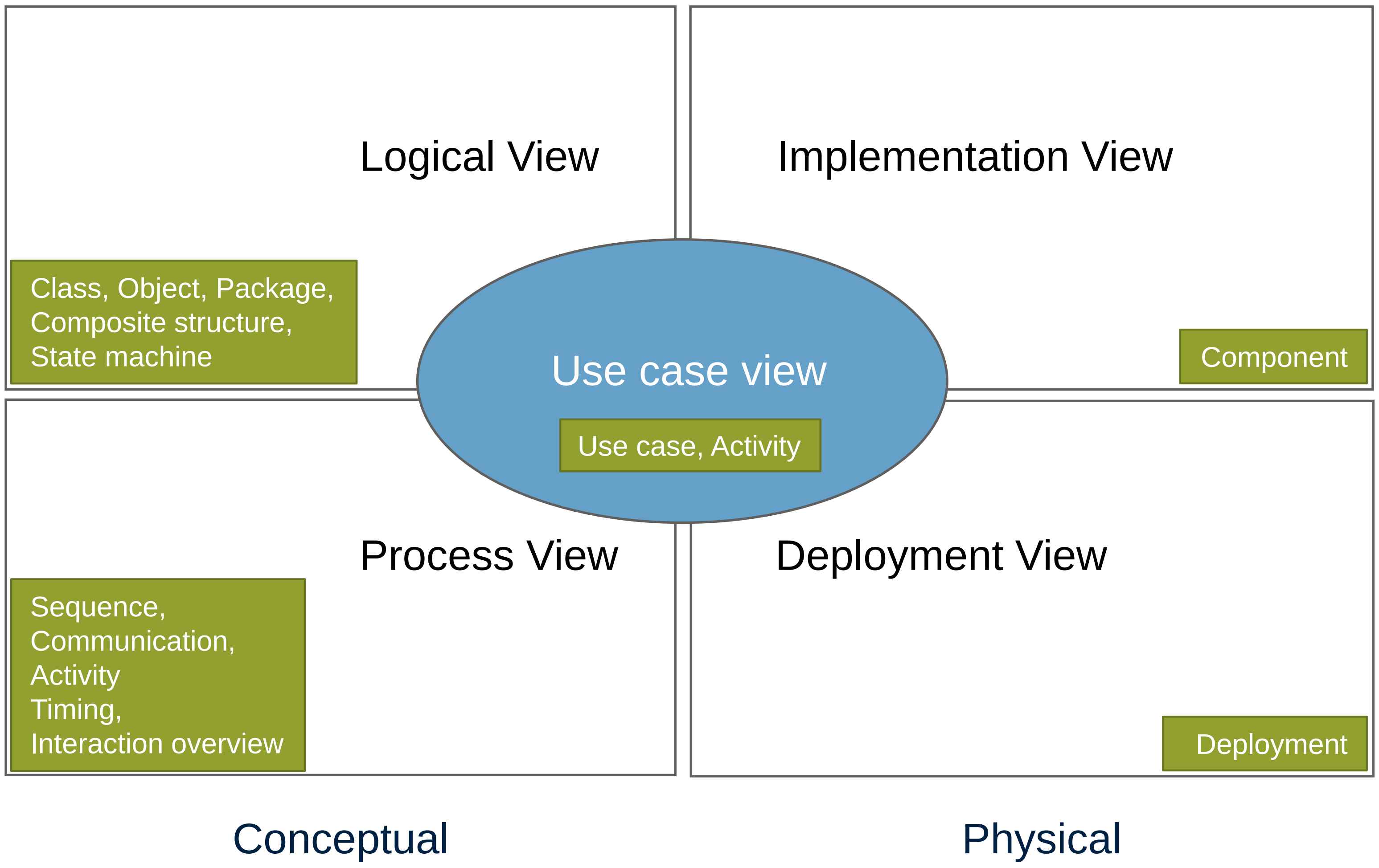
Modeling 4+1 views with UML 2

Implementation, deployment, and use case view



["Applying 4+1 View Architecture with UML 2." Muchandi V.]

Modeling 4+1 views with UML 2



["Applying 4+1 View Architecture with UML 2." Muchandi V.]

Another perspective on architectural viewpoints

C4 model for software architecture (1)

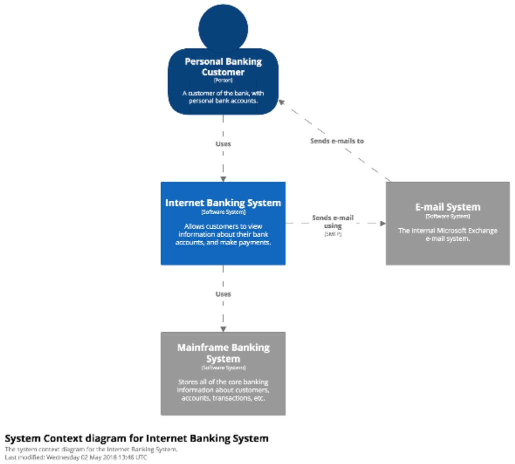
- First and foremost, think about the intended **audience** with whom you want to communicate your systems' architecture.
- Visualize the hierarchy of abstractions
  - for example, using C4 model – **Context**, **Container**, **Component** and **Code**

- **System Context diagram**
  - The focus is on **people** (actors, roles, personas, etc) and **software systems** rather than technologies, protocols and other low-level details. It's the sort of diagram that you could show to *non-technical* people.
- **Container diagram**
  - Shows the high-level shape of the software architecture and how responsibilities are distributed across it. It also shows the major technology choices and how the containers communicate with one another.
- **Component diagram**
  - Shows how a container is made up of a number of "components", what each of those components are, their responsibilities and the technology/implementation details.
- **Code**
  - Shows the classical class diagrams or entity-relationship diagrams

zoom in

[https://c4model.com – Simon Brown]

C4 model for software architecture (2)

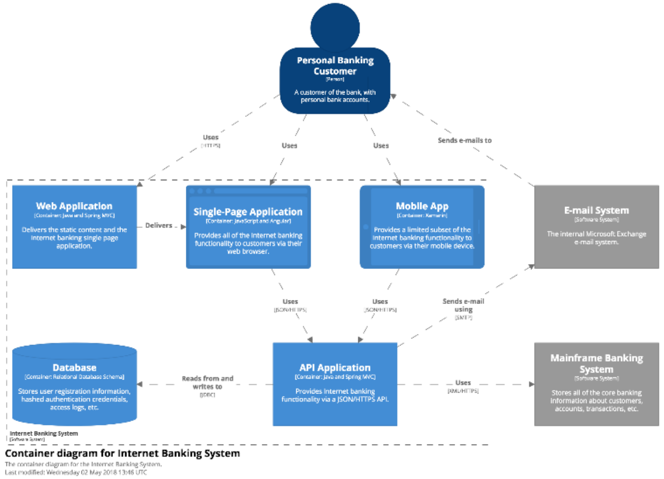


System Context diagram

Scope: A single software system.

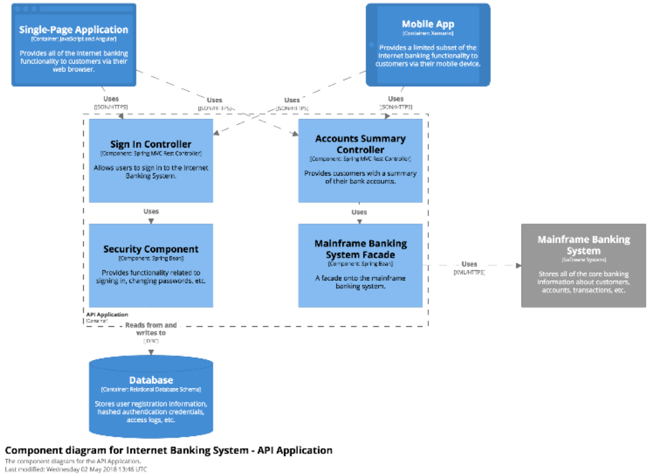
Primary elements: The software system in scope.
Supporting elements: People and software systems directly connected to the software system in scope.

Intended audience: Everybody, both technical and non-technical people, inside and outside of the software development team.



Container diagram

Scope: A single software system.
Primary elements: Containers within the software system in scope.
Supporting elements: People and software systems directly connected to the containers.
Intended audience: Technical people inside and outside of the software development team; including software architects, developers and operations/support staff.
Notes: This diagram says nothing about deployment scenarios, clustering, replication, failover, etc.

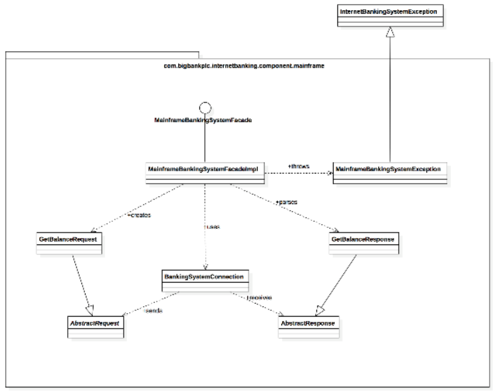


Component diagram

Scope: A single container.

Primary elements: Components within the container in scope.
Supporting elements: Containers (within the software system in scope) plus people and software systems directly connected to the components.

Intended audience: Software architects and developers.



Code

Scope: A single component.

Primary elements: Code elements (e.g. classes, interfaces, objects, functions, database tables, etc) within the component in scope.

Intended audience: Software architects and developers.

Don't be restricted by UML, ArchiMate, or SysML, if the sole purpose is **efficient communication of software architectures.**

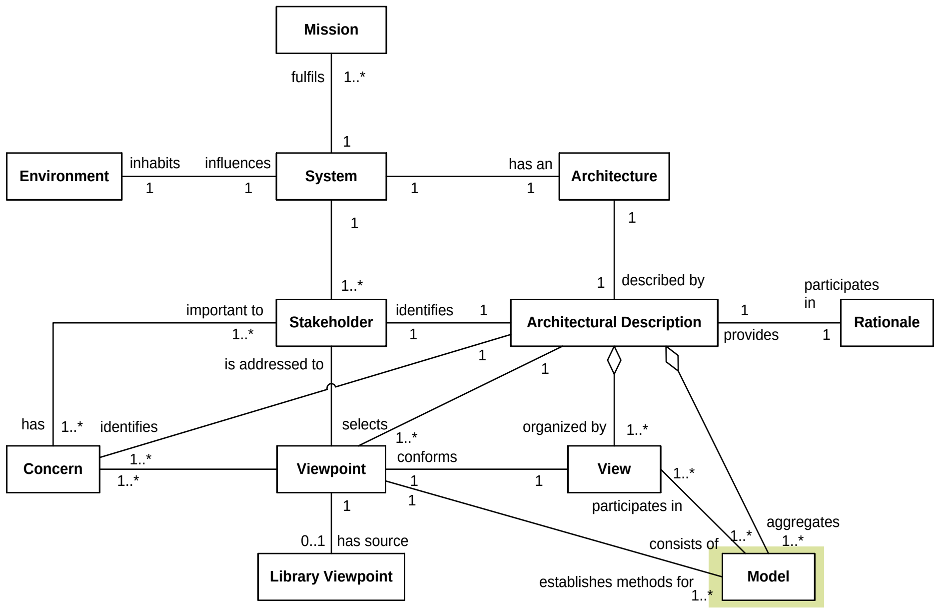
Recommended talk: Simon Brown - The Art of Visualising Software Architecture
https://www.youtube.com/watch?v=zcmU-OE452k

Architecture in software intensive systems (6)

System Model

- Object Model
  - What is the structure of the system?
  - What are the objects and how are they related?
- Functional model
  - What are the functions of the system?
  - How is data flowing through the system?
- Dynamic model
  - How does the system react to external events?
  - How is the event flow in the system ?

Models are used to provide **abstractions**



["Object-oriented software engineering." Bruegge B. and Dutoit A. H. (2004)]

Models

Models are characterized by the following three main attributes:

Reduction

- Models capture only parts of the represented original, which are of importance for the model creator
- Irrelevant details are omitted
- This attribute leads to an increase of the abstraction level

Pragmatism

- Models are created with a purpose in mind. They are adequate for this purpose or not, but never “good” or “bad” without a purpose.

Mapping

- Models are always models "of something", there is a mapping between elements of the models and the represented original.
- Models can be models of other models (recursively)

["Allgemeine Modelltheorie." Stachowiak H. (1973)]

As-Is vs To-Be model

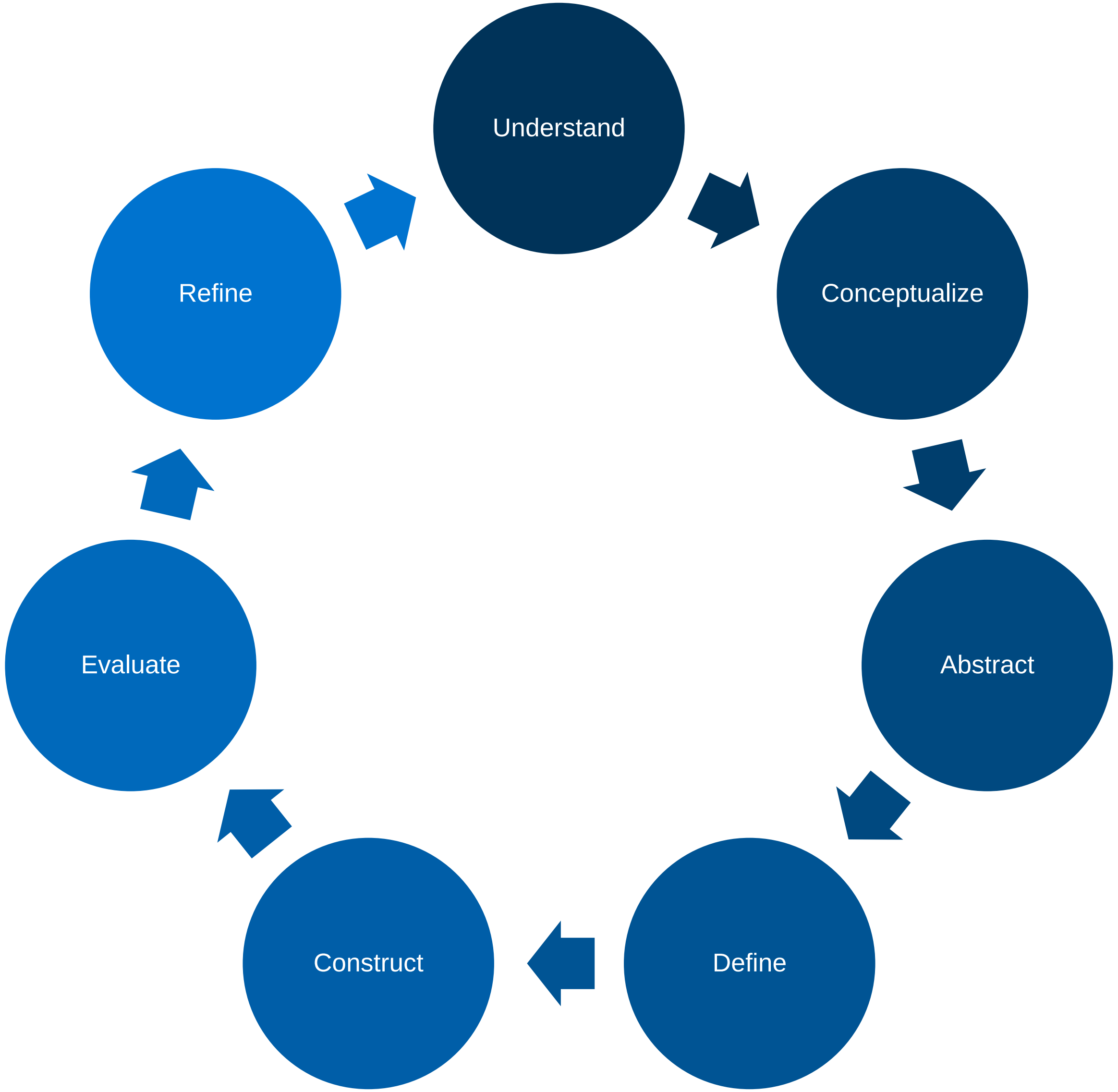
As-Is model: Usually during the modeling of a system, a scenario, situation etc. one needs to describe the current state of the system / the real world. Such models are called "As-Is" Models.

Such models are used to describe the starting point of a system and describe the differences with the envisioned future.

To-Be model: To describe the imagined / envisioned system, scenario, situation etc. models are created that describe how the system should look once it is changed from its "Is" state to its "To-Be" state. Such models are therefore called "To-Be" models.

The difference between the two models is the set of system components, scenario elements, situational steps etc. that need to change. Such elements are called the "**diff**".

The act of modeling



["Models, to Model, and modelling - towards a theory of conceptual models and modeling." Thalheim B. (2007)]

The act of modeling

- **Understand**: The understanding act support reasoning within the application domain. It results typically in drafts that can be used for development of conceptualizations. The problems and possible solutions are comprehended. Conjectures are drawn.

- **Conceptualize**: The conceptualization act aims in formalizing the part of the application domain that is of interest. We form a number of concepts of the application domain and represent those formally within the concept language. These concepts are conceptually interpreted in the application domain.

- **Abstract**: The abstraction act aims in outlining main problems that must be supported by the information system. It generalizes these problems and abstracts from unnecessary details on the basis of forgetful mappings.

- **Define**: The definition act is used to unambiguously specify, to delineate, and to delimit main concepts or annotations used for the development of the model. Definitions might be given in a variety of forms. We can use also different languages and target on visualization of concepts.

["Models, to Model, and modelling - towards a theory of conceptual models and modeling." Thalheim B. (2007)]

The act of modeling

- **Construct**: The construction act is often considered to be the main act during modelling. It aims in creating a model by organizing and linking ideas, judgements, or concepts. It may include the sub-act of rebuilding, i.e., reconstructing, framing up, and customizing. When we talk about anticipated behavior it includes activities of conjecturing and hypothesizing.

- **Evaluate**: The evaluation act is based on a set of quality characteristics that we have agreed to satisfy in advance. These quality characteristics are typically given in an abstract form and are not solely based on metrics. Evaluation is typically applied to a model or parts of it. It results in determination of the value of the judgements under evaluation.

- **Refine**: The refinement act is a basic act of iterative development. The model itself becomes enriched, more elaborated or sophisticated while preserving its main structures and behavior. It matches thus in a better, more precise manner the needs of the application. The refinement act is typically based on some evaluation or assessment and on analysis for improvement potential. Refinement uses scoping for restricting changes to a necessary extent.

["Models, to Model, and modelling - towards a theory of conceptual models and modeling." Thalheim B. (2007)]

From requirements to system design

Modularity

- **Decomposition** of systems into modules (components)
- Measures degree to which components can be separated and recombined
  - Idea: replacing one module by an "equivalent" one, with a different implementation, does not change overall system behavior
- Aims to implement **information hiding**
- Aims at easier **maintenance** and **reusability**
- Aims to **increase understandability**
- Allows for **work distribution**
- A technique used to **reduce complexity** (“divide and conquer”)
  - Functional decomposition
  - Modular decomposition

The need for modularity - stakeholders perspective

Project manager

- React on new requirements faster
- Shorter release cycles
- Keep investments of the past
- Avoid changes in the UI to not confuse customers

Development manager

- Increase internal productivity by parallelization
- Use external teams
- Use new developers with different or less skills

Sales representative

- Sell parts of existing product separately
- More combination options, also external products
- Installations at customer's site results in additional revenue streams

Developers and testers

- Less coordination effort by working on separate modules
- More guidance from the systems structure
- Keep proven technologies and complex algorithms

Architect

- React on new requirements easily
- Local changes in manageable parts
- Clear assignment of responsibilities to modules
- Flexible use of new technologies

Customizer

- Separate customizations clearly from product code
- Reuse parts of the system
- Keep existing customizations

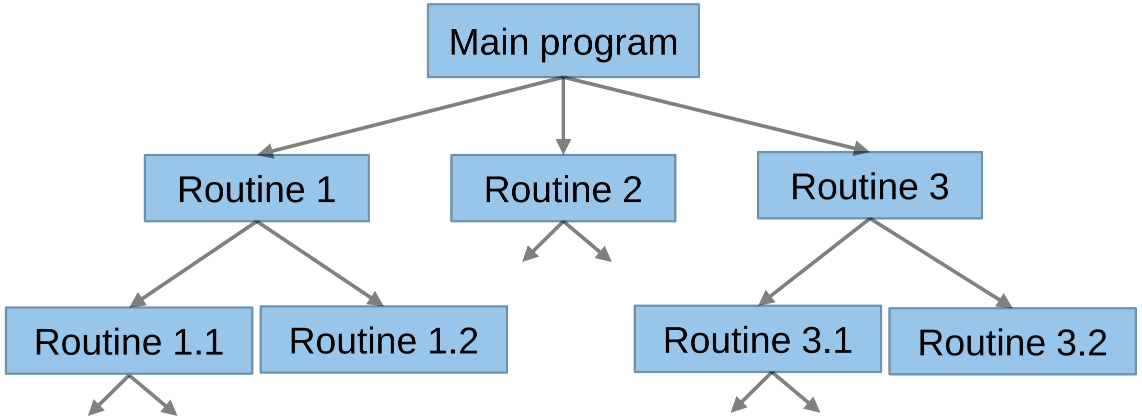
Support staff

- Reconstruct problem situations
- Locate problems easily

["Modularity – often desired, but rarely achieved." Knodel J. et al. (2015)]

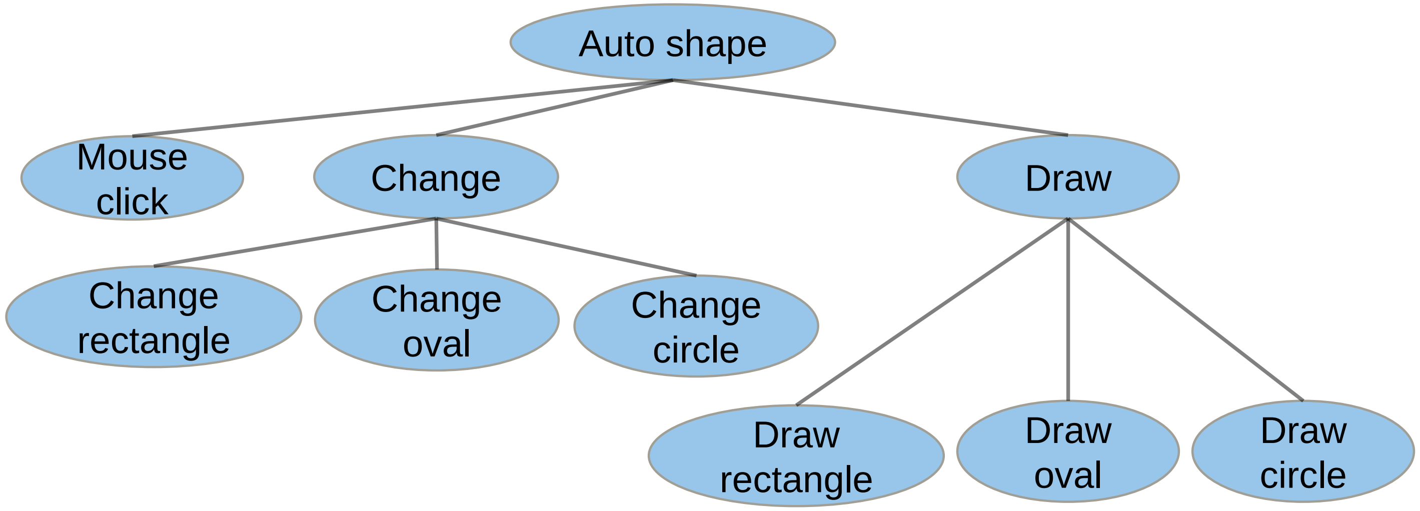
Functional decomposition

- The system is decomposed into modules
- Each module is a major processing step (***function***) in the application domain
- Modules can be decomposed into smaller modules



Functional decomposition - issues

- Functionality is spread all over the system
- Need to understand the whole system to make a single change to the system
- Consequence
  - Source codes is hard to understand
  - Code becomes complex and impossible to maintain
  - User interface is often awkward and non-intuitive
- Example: Microsoft PowerPoint's auto shapes



["Object-oriented software engineering." Bruegge B. and Dutoit A. H. (2004)]

Modular decomposition

- The system is decomposed into modules
- Each module is a major abstraction in the application domain
- Modules can be decomposed into smaller sub-modules

Basic assumptions

- We can find concepts for a new software system - greenfield engineering
- We can identify the concepts in an existing system – reengineering
- We can create a component-based interface to any system – interface engineering

Why can we do this?

- Experimental evidence in philosophy and science

Limitations

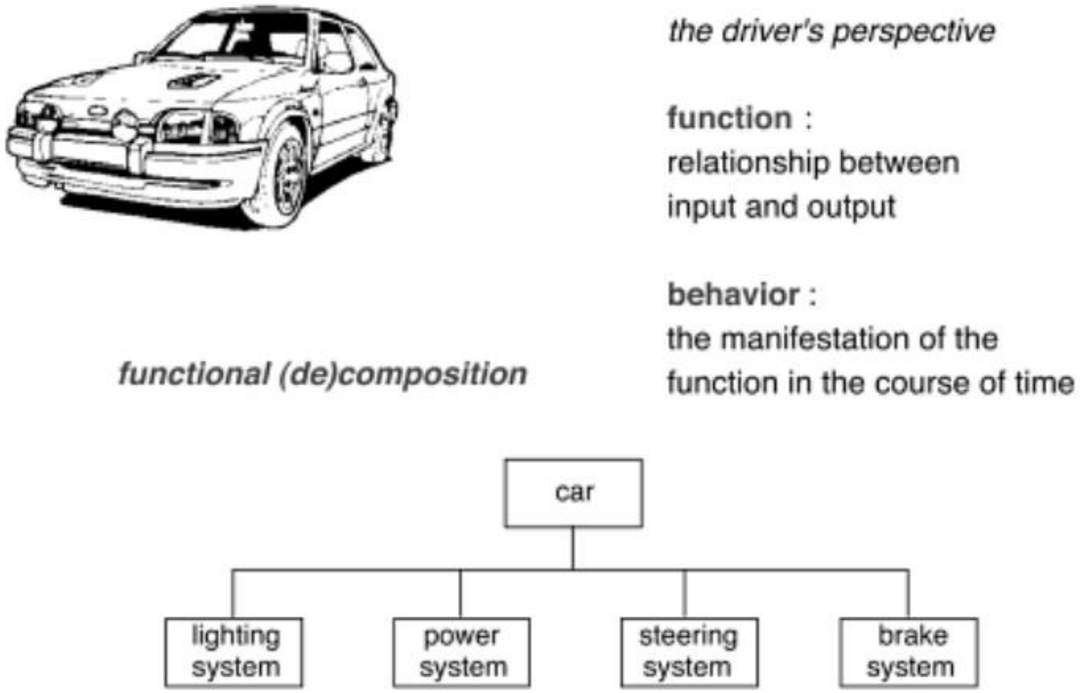
- Depending on the purpose of the system different concepts might be found

["Object-oriented software engineering." Bruegge B. and Dutoit A. H. (2004)]

Black-box model

A Black-box model presents the **functional perspective** on a system and captures the functionality and the (external) behavior of the system.

- Only the interactions between the composition and the environment are taken into account, in an abstract way: they are represented as aggregated values of input and output variables.
- Control or management model of enterprise. [managing system and managed system]
- Functional (de)composition

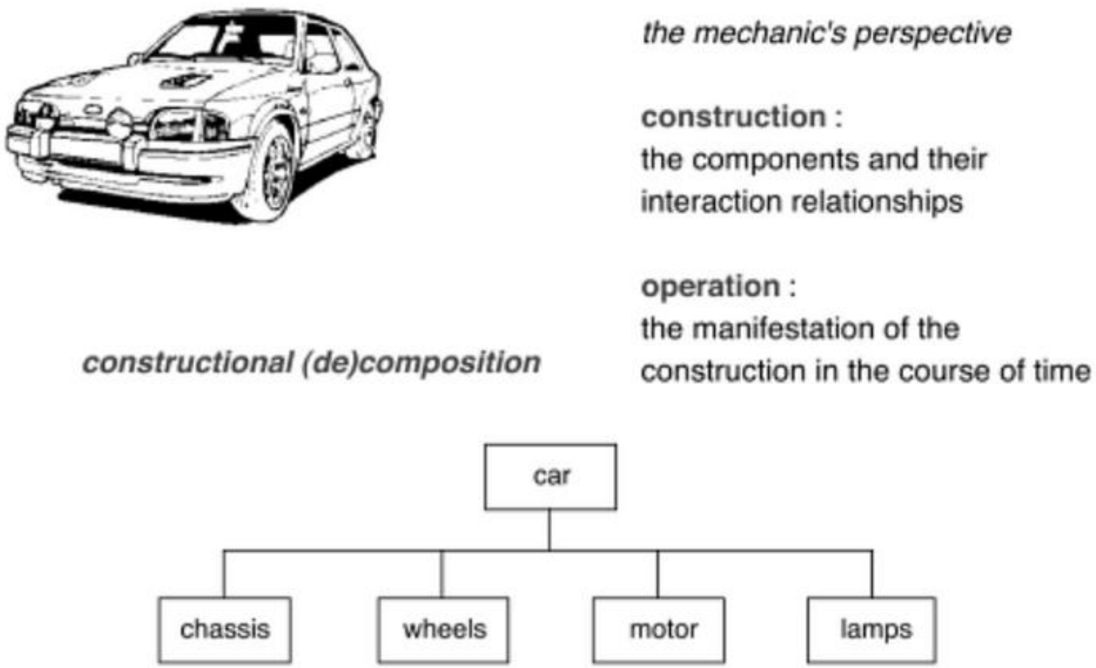


["What is Enterprise Ontology?" Dietz, Jan LG. (2006)]

White-box model

A White-box model presents the **construction perspective** on a system and captures the construction and the operation of the system.

- Constructional (de)composition

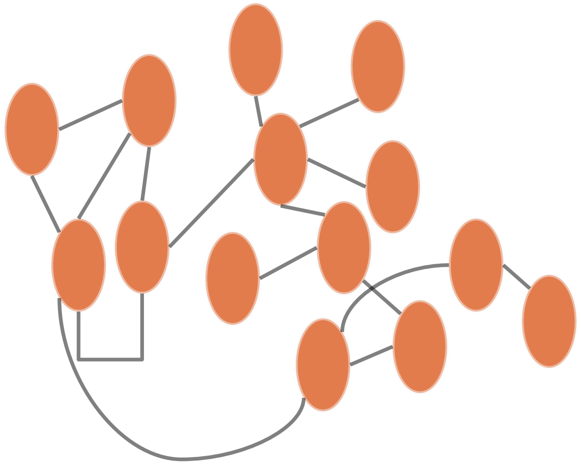


["What is Enterprise Ontology?" Dietz, Jan LG. (2006)]

Decomposition (1)

- A technique to master complexity ("divide and conquer")

- A typical way to come up with the software architecture
  - analyze **dependencies** between elements
    - **decompose** the system into smaller, manageable parts

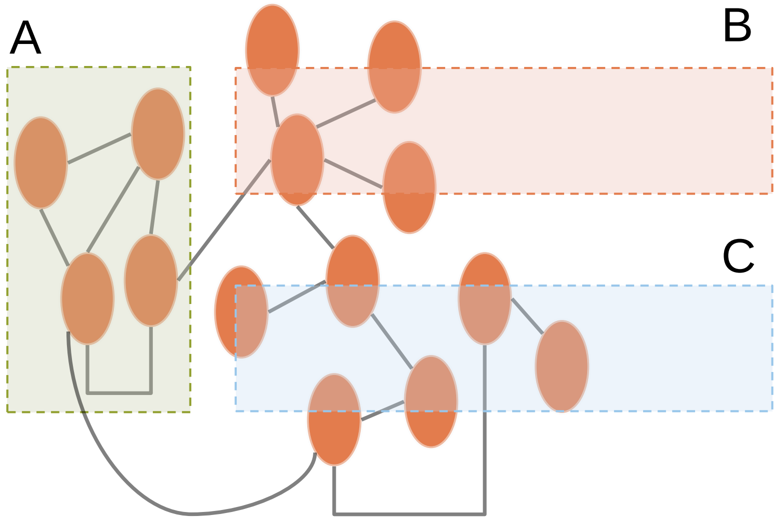


„Chapter 2.1.1 Slide 37-38: Boxes“

Decomposition (2)

- A technique to master complexity ("divide and conquer")

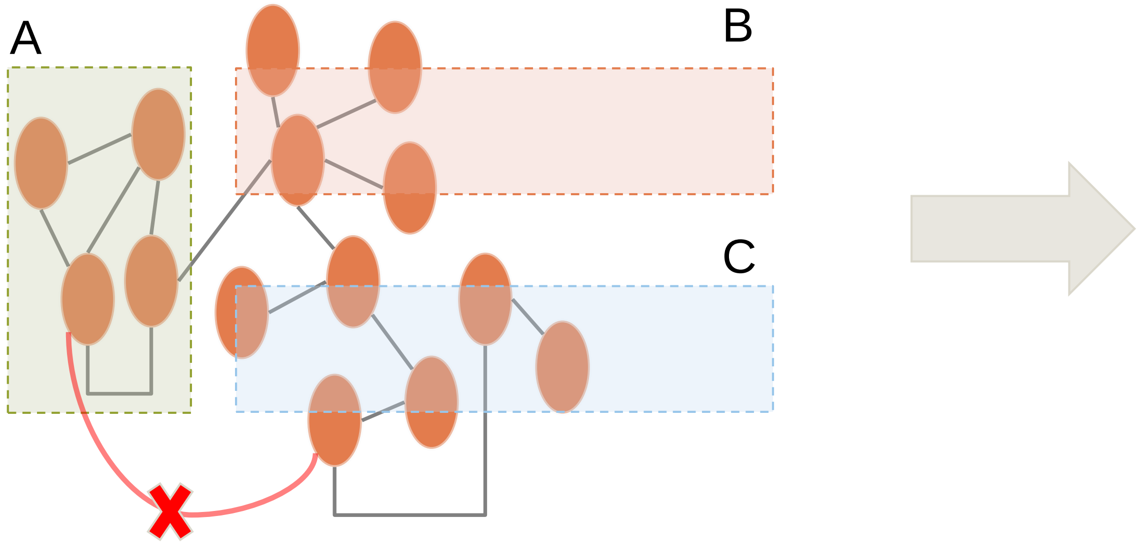
- A typical way to come up with the software architecture
  - analyze **dependencies** between elements
  - elements with strong dependencies form **components**

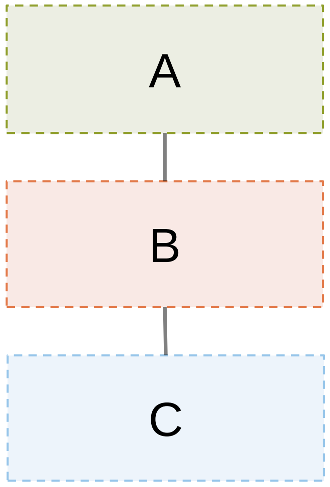


Decomposition (3)

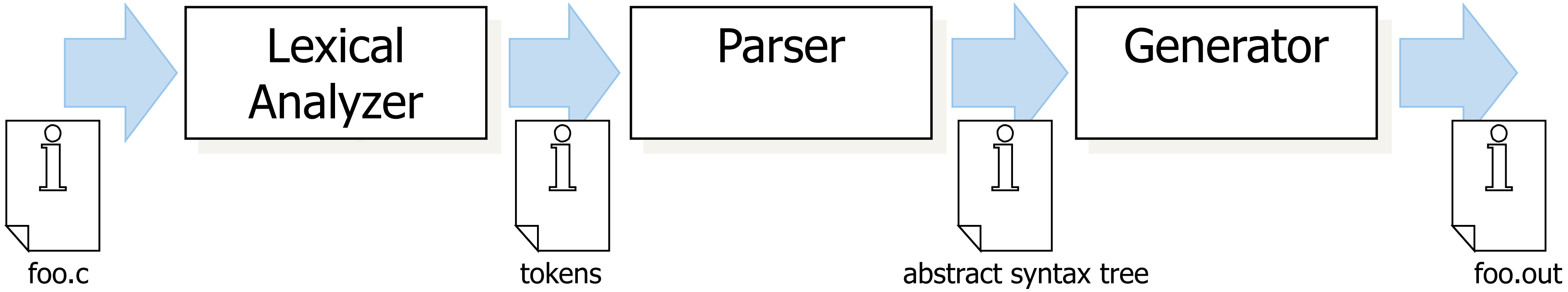
- A technique to master complexity ("divide and conquer")

- A typical way to come up with the software architecture
  - analyze **dependencies** between elements
  - elements with strong dependencies form **components**
  - **dependencies** between **components** captured by well-defined **interfaces**





Example – compiler architecture



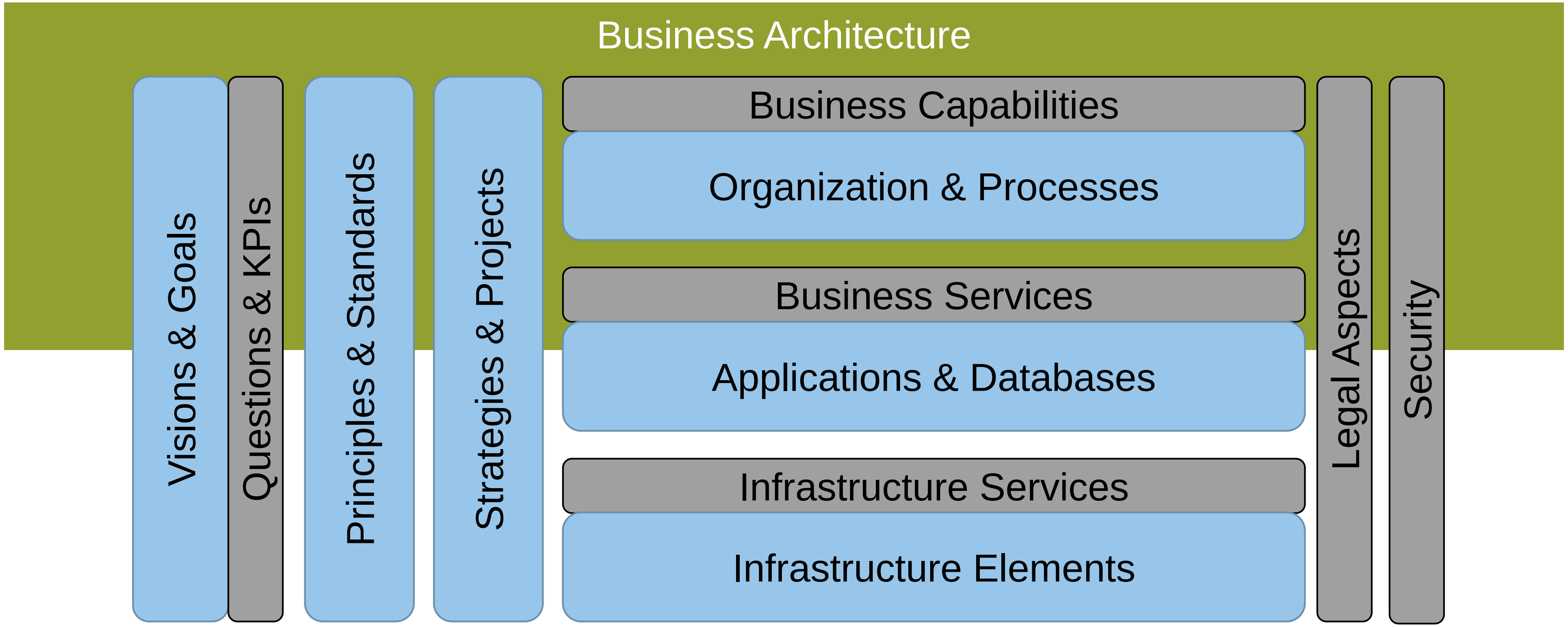
Key characteristics

- separation of concerns: clear **sequential** process
- modularization: allows distributed development
- decoupling: allows replacement and extension

Architectural style

- Pipes-and-filters architecture

Example – enterprise architecture



Key characteristics

- **information hiding**: communication only through defined interfaces
- **modularization**: allows distributed development
- **low coupling**: eases replacement and extension

Architectural style

- **layered-architecture**

From requirements to system design

Software components

"A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."

["Component-based software engineering: putting the pieces together." Heineman G. T. and William T. (2001)]

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties."

["Component software: beyond object-oriented programming." Szyperski, C. (2002)]

Component-based software engineering

- Component-based software engineering (**CBSE**) is an approach to software development that relies on the reuse of entities called "**software components**".

- It emerged from the failure of object-oriented development to support effective reuse. **Single object classes are too detailed and specific.**

- Software components are more abstract than object classes and can be considered to be stand-alone service providers. They can exist as stand-alone entities.

- Software components provide a service without regard to where the component is executing or its programming language
  - A software component is **an independent executable entity** that can be made up of one or more executable objects;
  - The software component **interface is published and all interactions are through the published interface;**

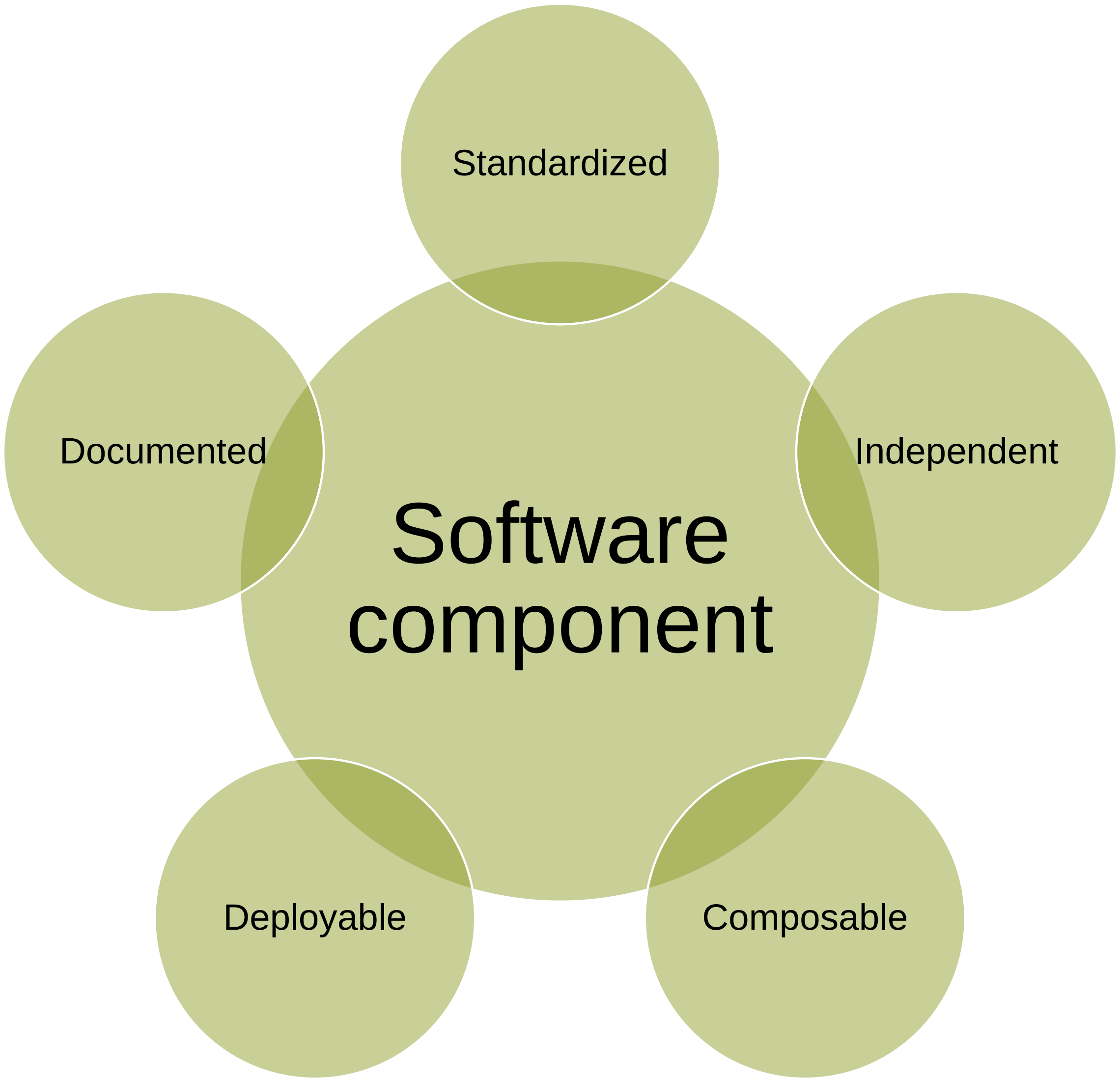
["Software Engineering." Sommerville I. (2010)]

The essentials of CBSE

- Independent components that are completely specified by their interfaces
- Component standards that facilitate the integration of components
- Middleware that provides software support for component integration
- A development process that is geared to component-based software engineering

["Software Engineering." Sommerville I. (2010)]

Characteristics of software components



["Software Engineering." Sommerville I. (2010)]

Characteristics of software components

- **Standardized** - Component standardization means that a component used in a CBSE process has to conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment.

- **Independent** - A component should be independent—it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a ‘requires’ interface specification.

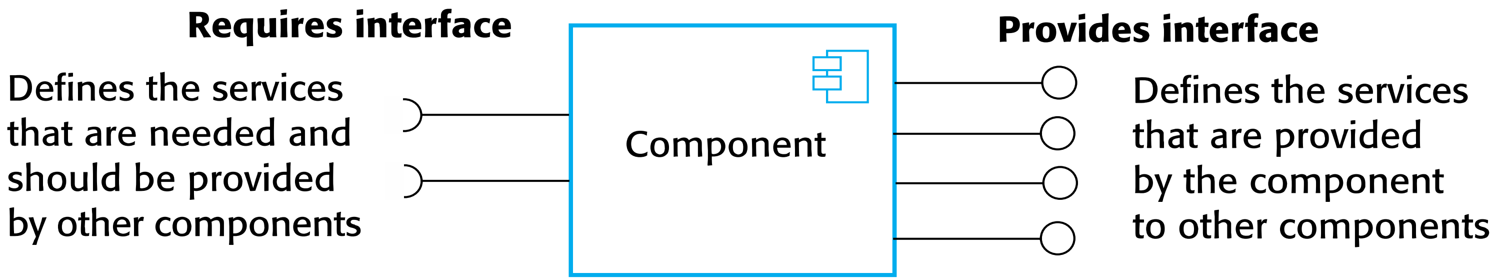
- **Composable** - For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself, such as its methods and attributes

- **Deployable** - To be deployable, a component has to be self-contained. It must be able to operate as a stand-alone entity on a component platform that provides an implementation of the component model. This usually means that the component is binary and does not have to be compiled before it is deployed. If a component is implemented as a service, it does not have to be deployed by a user of a component. Rather, it is deployed by the service provider.

- **Documented** - Components have to be fully documented so that potential users can decide whether or not the components meet their needs. The syntax and, ideally, the semantics of all component interfaces should be specified.

["Software Engineering." Sommerville I. (2010)]

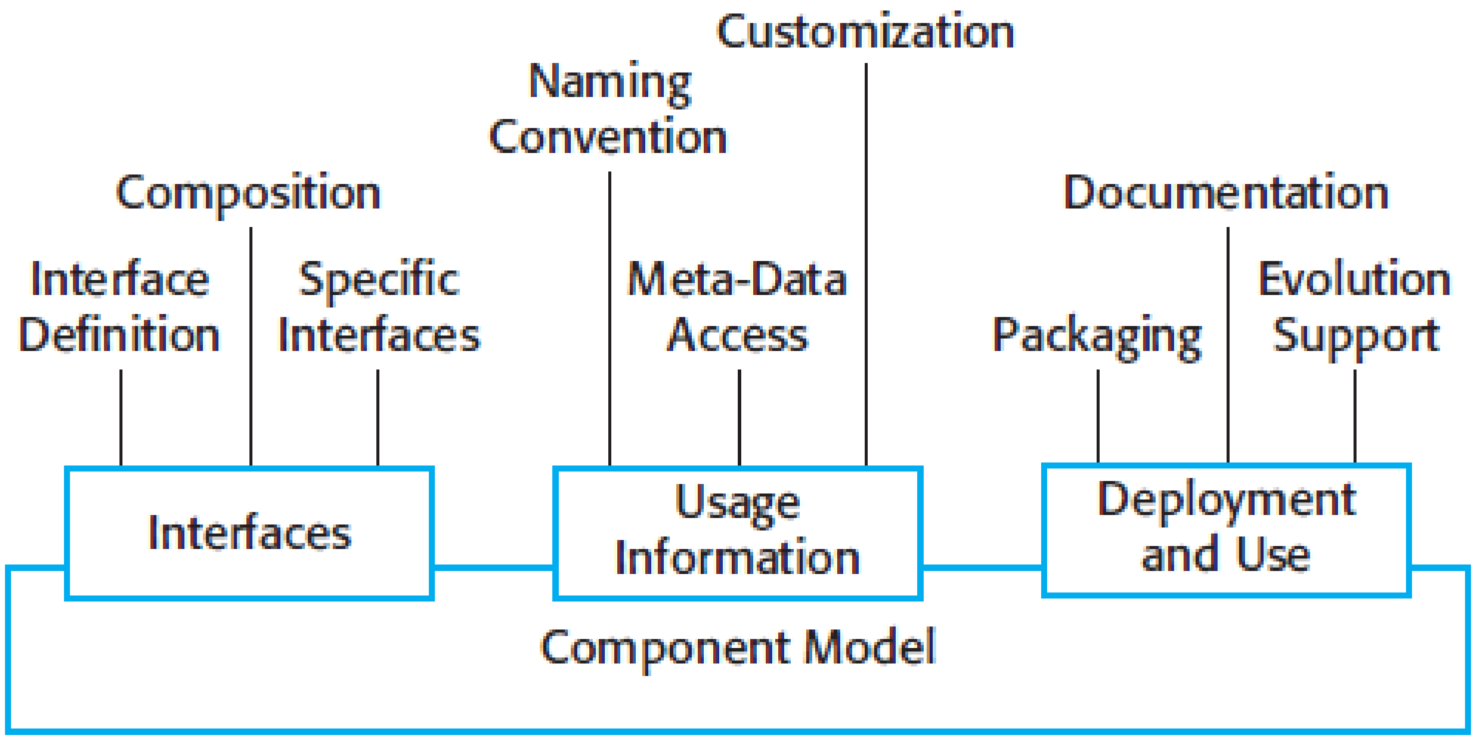
Component interfaces



["Software Engineering." Sommerville I. (2010)]

Component model and its elements

A component model is a definition of standards for component implementation, documentation, and deployment.



["Software Engineering." Sommerville I. (2010)]

Basic elements of a component model (1)

Interfaces

- Components are defined by specifying their interfaces. (Interface definition)

- The component model specifies how the interfaces should be defined and the elements, such as operation names, parameters, and exceptions, which should be included in the interface definition.

- The model should also specify the language used to define the component interfaces

- Examples
  - WSDL for web services
  - EJB

- Some component models require specific interfaces that must be defined by a component. These are used to compose the component with the component model infrastructure, which provides standardized services such as security and transaction management.

["Software Engineering." Sommerville I. (2010)]

Basic elements of a component model (2)

Usage

- Naming convention
  - In order for components to be distributed and accessed remotely, they need to have a unique name or handle associated with them.
  - Examples
    - URI for web services
    - In EJB, a hierarchical name is generated with the root based on an Internet domain name.

- Meta-data
  - Data about the component, such as information about its interfaces and attributes
  - Helps users to find out what services are provided and required
  - Example
    - Use of a reflection interface in Java

["Software Engineering." Sommerville I. (2010)]

Basic elements of a component model (3)

Deployment

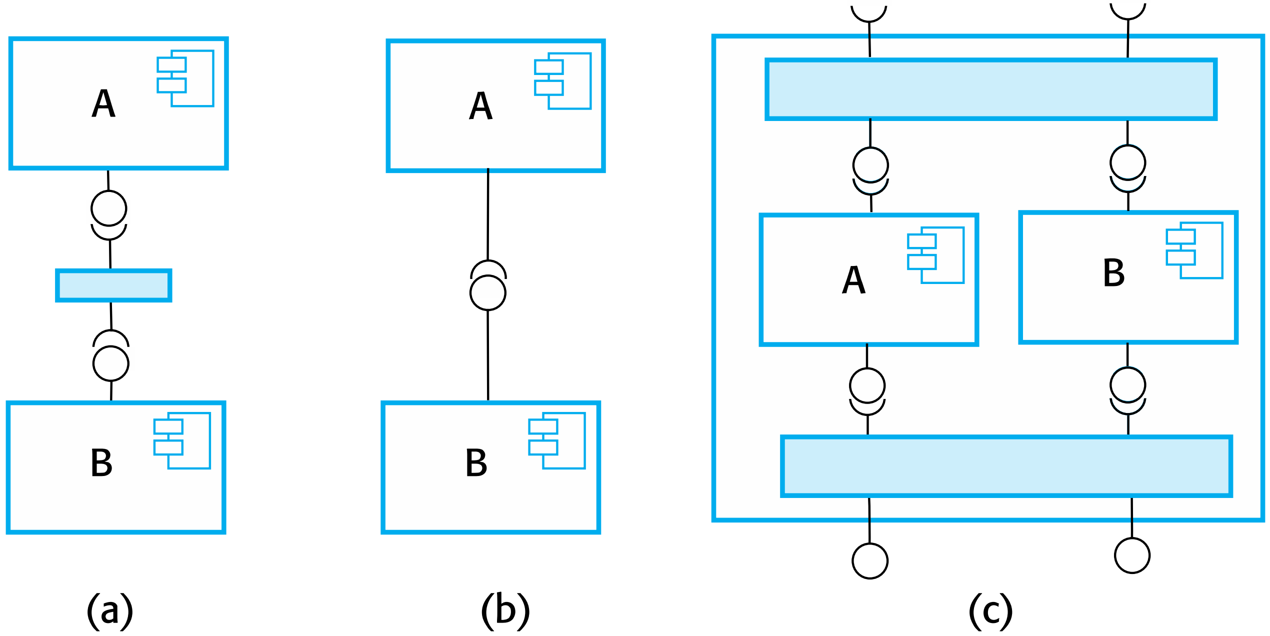
- Packaging
  - The component model includes a specification of how components should be packaged for deployment as independent, executable entities.
  - Because components are independent entities, they have to be packaged with all supporting software that is not provided by the component infrastructure, or is not defined in a "requires" interface.
  - Deployment information includes information about the contents of a package and its binary organization.

["Software Engineering." Sommerville I. (2010)]

Component composition

The process of assembling components to create a system. Composition involves integrating components with each other and with the component infrastructure.

*Normally one has to write "glue code" to integrate components



Sequential composition

Hierarchical composition

Adaptive composition

["Software Engineering." Sommerville I. (2010)]

Component-based software engineering

Pros:

- **Independent components** specified by their interfaces.
- **Component standards** to facilitate component integration.
- **Middleware** that provides support for component inter-operability.
- A **development process** that is geared to reuse.

Cons:

- **Component trustworthiness** - how can a component with no available source code be trusted?
- **Component certification** - who will certify the quality of components?
- **Emergent property prediction** - how can the emergent properties of component compositions be predicted?
- **Requirements trade-offs** - how do we do trade-off analysis between the features of one component and another?

["Software Engineering." Sommerville I. (2010)]

From requirements to system design

Design by contract

> "For designing and developing the Eiffel programming language, method and environment, embodying the Design by Contract approach to software development and other features that facilitate the construction of *reliable*, *extendible* and *efficient* software."

[Bertrand Meyer (2003)]

- Also known as
  - contract programming
  - programming by contract
  - design-by-contract programming

- Design by contract presents a set of principles to produce **dependable** and **robust** object-oriented software

- An important aspect of object-oriented design is **reuse**
  - For reusable components correctness is crucial since an error in a module can affect every other module that uses it

["Object oriented software construction." Meyer B. (1988)]

What is a contract?

- Contract is the agreement between the client and the supplier
  - Client which requests/consumes a service
  - Supplier which supplies the service

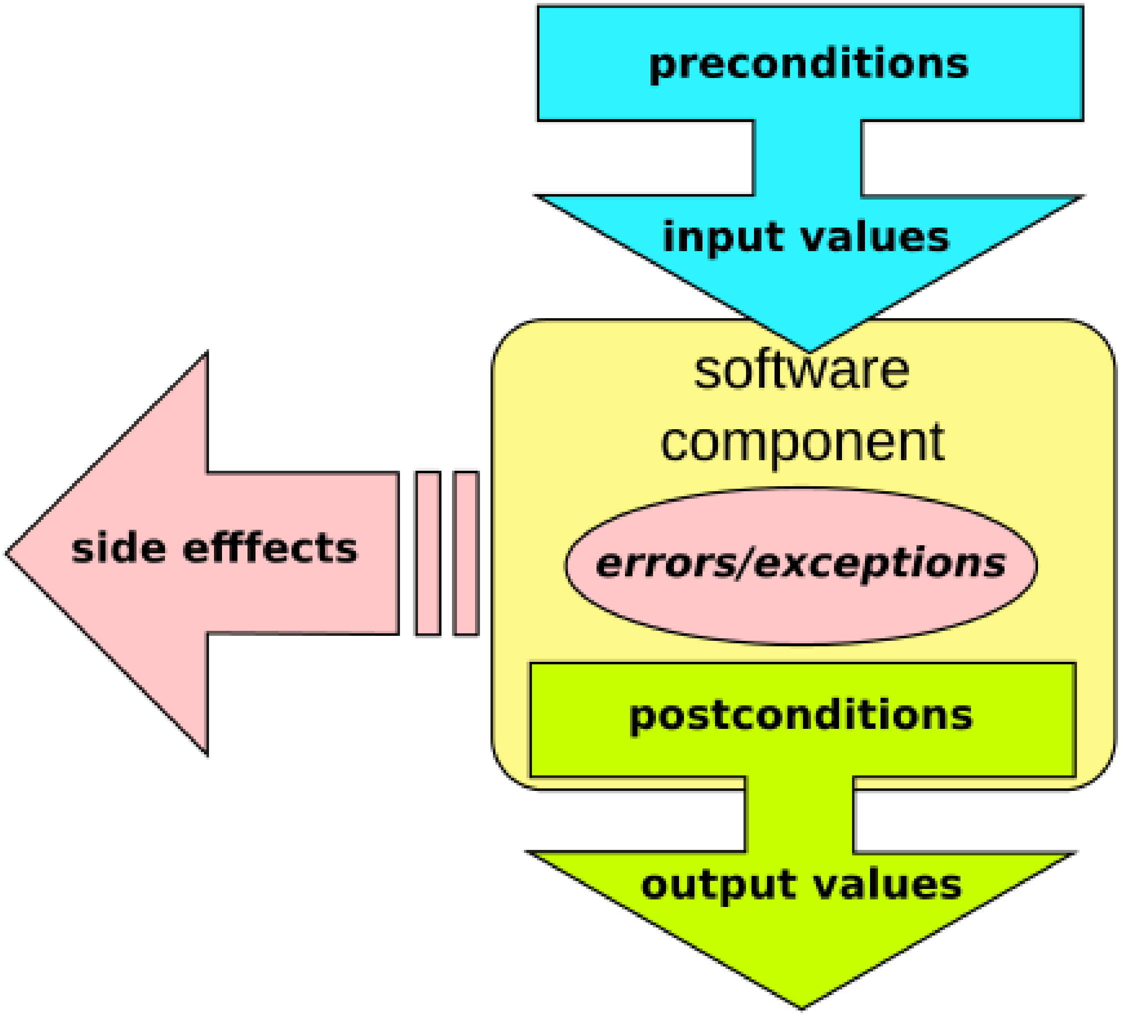
Contracts are "specs 'n' checks."

- Two major characteristics of a contract
  - Each party expects some **benefits** from the contract and is prepared to incur some **obligations** to obtain them
  - These benefits and obligations are documented in a contract document
    - *No Hidden Clauses Rule*: no requirement other than the obligations written in the contract can be imposed on a party to obtain the benefits

- The pre- and post-conditions are assertions, i.e., they are expressions which evaluate to true or false
  - The pre-condition expresses the requirements that any call must satisfy
  - The post-condition expresses the properties that are ensured at the end of the procedure execution

["Object oriented software construction." Meyer B. (1988)]

A design by contract schema



["Object oriented software construction." Meyer B. (1988)]

Design by contract - Eiffel example

- Create a routine called *put* to insert an element to a generic class DICTIONARY [ELEMENT] so that it is retrievable through a key
  - dictionary (a table where each element is identified by a certain character string used as key) of bounded capacity

- Given -- defined in the generic class
  - count - current number of elements in the dictionary
  - capacity - maximum number of elements that a dictionary can contain
  - **old** - used to denote the value of a variable on entry to the routine
  - **old** count - refers to the value of count on entry to the routine

  - *has* - boolean query that tells if a certain element is present
  - *empty* - boolean query that tells if a certain string is empty
  - *item* - returns the element associated with a certain key -- *item(key)*

- Note that "=" is the equality operator (== in Java) and "/=" is the inequality operator (!= in Java)

Design by contract - Eiffel example

	Obligations	Benefits
Client	*(Must ensure precondition)* Make sure table is not full and key is a non-empty string.	*(May benefit from post-condition)* Get updated dictionary where the given element now appears, associated with the given key.
Supplier	*(Must ensure post-condition)* Record given element in dictionary, associated with given key.	*(May assume precondition)* No need to do anything if table is full, or key is empty string.

Design by contract - Eiffel example

```
class DICTIONARY  [ELEMENT]
feature
        put (x: ELEMENT; key: STRING) is
        -- Insert x so that it will be retrievable through key.      <- Header comment
require
        count <= capacity                                             <- Preconditions
        not key.empty
do
        ... Some insertion logic ...
ensure
        has (x)                                                       <- Post-conditions
        item (key) = x
        count = old count + 1
end
```

Design by contract - Principles

- **Non-redundancy principle** Under no circumstances shall the routine's body ever test its own precondition

- **Reasonable precondition principle** Every routine precondition must satisfy the following requirements:
  - The precondition appears in the official documentation distributed to authors of client modules.
  - It is possible to justify the need for the precondition in terms of the specification only.

- **Failure principle** Execution of a rescue clause to its end, not leading to a retry instruction, causes the current routine call to fail.

- **Disciplined exception handling principle** There are only two legitimate responses to an exception that occurs during the execution of a routine:
  - *Retrying*: attempt to change the conditions that led to the exception and to execute the routine again from the start.
  - *Failure* (also known as *organized panic*): clean up the environment, terminate the call and report failure to the caller. In addition, exceptions resulting from some operating system signals may in rare cases justify a false alarm response: determine that the exception is harmless and pick up the routine's execution where it started.

- **Exception simplicity principle** All processing done in a rescue clause should remain simple, and focused on the sole goal of bringing the recipient object back to a stable state, permitting a retry if possible.

["Object oriented software construction." Meyer B. (1988)]

Defensive programming vs. design by contract

- Defensive programming is an approach that promotes putting checks in every module to detect unexpected situations

- This results in redundant checks (for example, both caller and callee may check the same condition)
  - A lot of checks makes the software more complex and harder to maintain

- In design by contract the responsibility assignment is clear and it is part of the module interface
  - prevents redundant checks
  - easier to maintain
  - provides a (partial) specification of functionality

["Object oriented software construction." Meyer B. (1988)]