Software Architecture Design Process

Oleksandr Savchenko

Speaker.

- 12+ years in software design & development (inc. 4 as Architect)
- SEI / TOGAF trained, icAgile Certified
- Software Developer, Architect,
  Head of Departments
- Engineering Direct at GlobalLogic
- Led big programs (> 100 engineers)
- winner of Ukrainian IT Awards in category Software Engineering in 2019
- speaker on conferences, meet ups
- conducts partnership programs with other big IT companies

What our Goals ?

- Spend time with **benefits**
- Get **new knowledge** and/or **structural understanding**
- Get a **plan** for further development

What to expect for You?

- 3 theoretical sections with Q&A
- Breaks after all Q&A session
- 2 practices
- On-line chat

Agreements.

- Mute
- Smart questions to the Chat
- Be on focus

Agenda.

Section 1 (~1h)

- Theory: Software Architecture Fundamentals - 40 min
- Practice: Architectural Drivers clarification - 10 min
- Q&A session - 10 min
- Break - 10 min

Section 2 (~40 min)

- Theory: Architectural reusable methods & standards
- Q&A session - 5 min
- Break - 5 min

Section 3 (~40 min)

- Theory: Process overview for creation of Software Architecture Documentation
- Practice: Creation of Architecture Documentation
- Break - 5 min

Section 4

- Summary
- Learning materials
- Q&A session

Software Architecture.

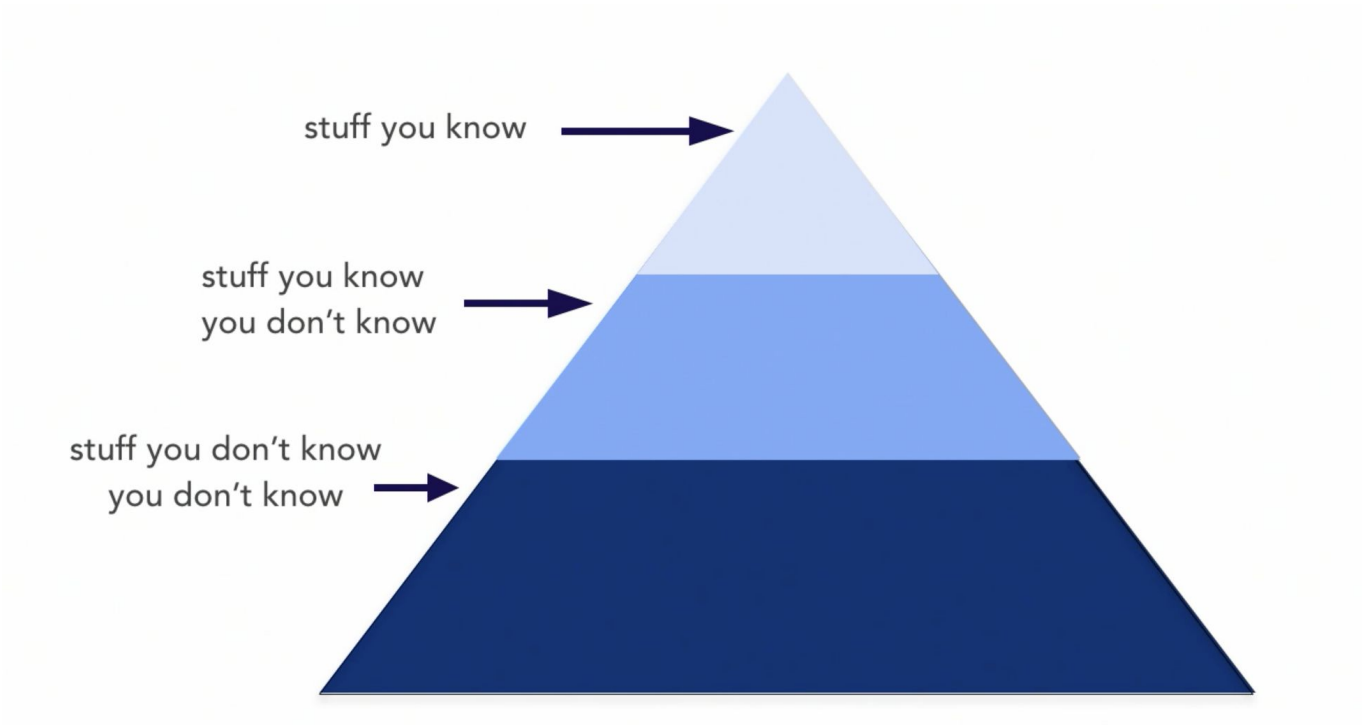
Fundamentals

Agenda.

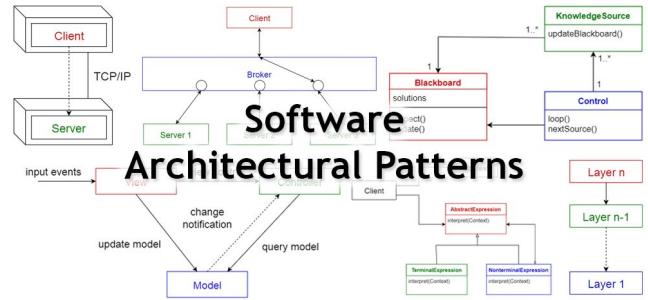
- First look into Software Architecture
- What is Software Architecture and why it’s important?
- Architect roles & responsibilities;
- Architecture development lifecycle:
  - 1. Analysis of Architectural Requirements
    - Architectural Drivers overview
    - Quality Attributes
  - 2. Architectural Design
    - Architecture Tactics vs Patterns vs Styles
  - 3. Architectural Documentation
  - 4. Architectural Evaluation
  - 5. Architectural Implementation

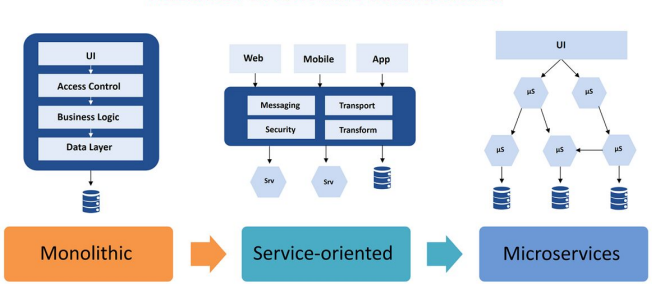
First look into
Software Architecture.

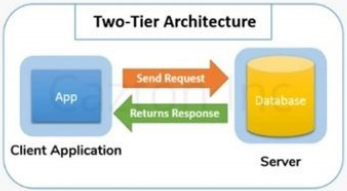
What we know about Software Architecture?

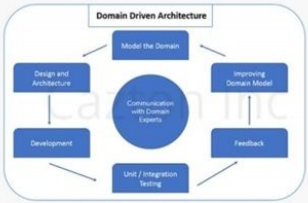


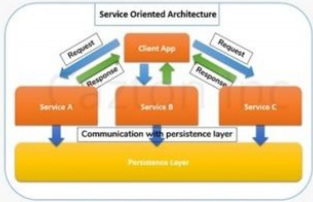
What we know about Software Architecture? - Patterns

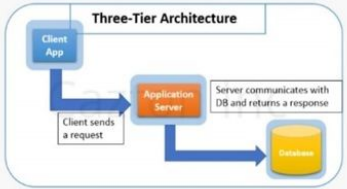


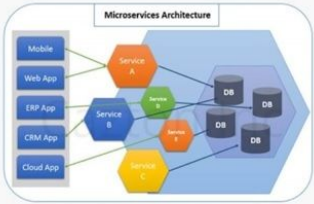


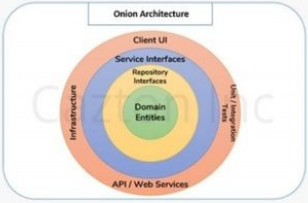


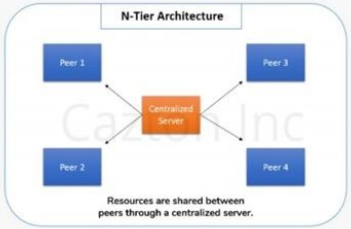


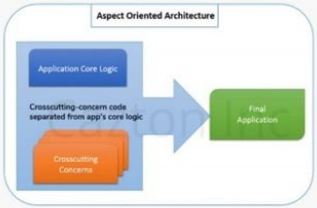












What we know about Software Architecture? - Technologies

kubernetes

HashiCorp Terraform

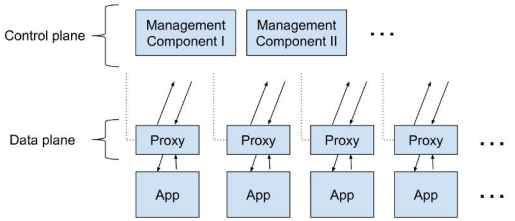
amazon web services

Google Cloud

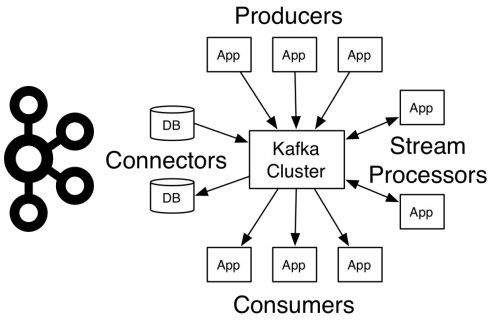
Azure

HashiCorp Consul

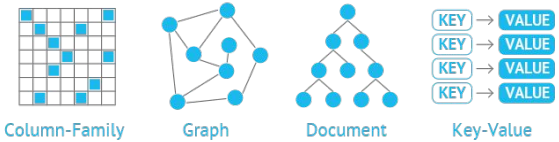
NETFLIX OSS Eureka

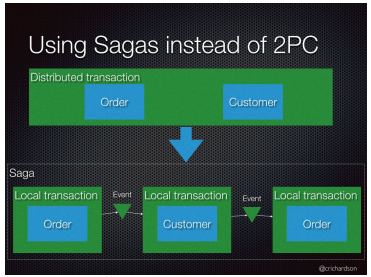


Service Mesh

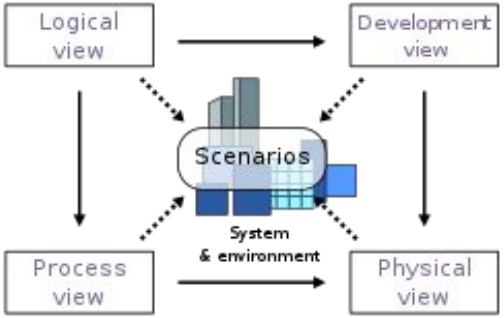


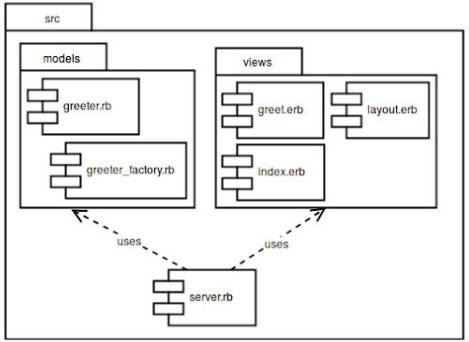
NoSQL Database



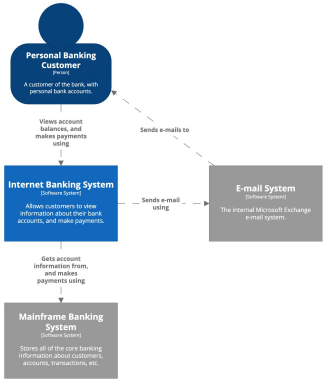


What we know about Software Architecture? - Schemas



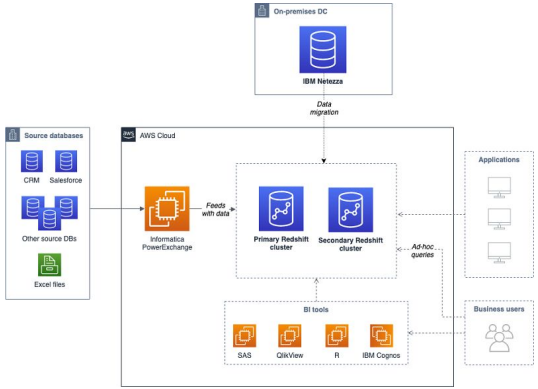


UNIFIED MODELING LANGUAGE™

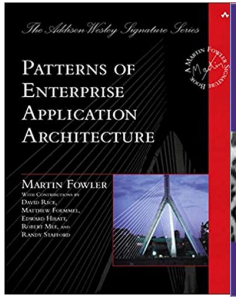


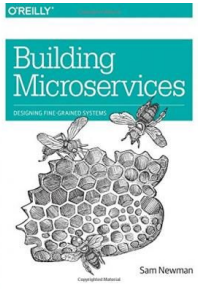
System Context diagram for Internet Banking System

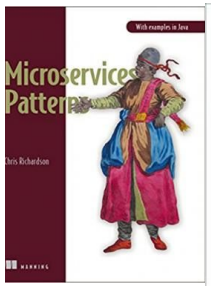
The system context diagram for the Internet Banking System.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

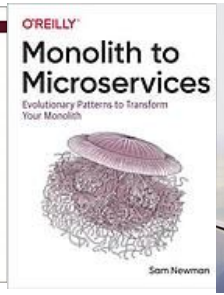


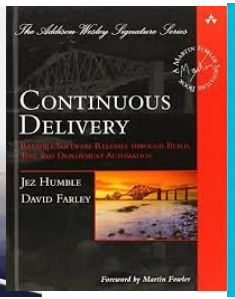
What we know about Software Architecture? - Books

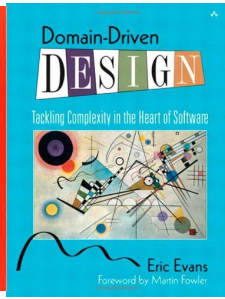


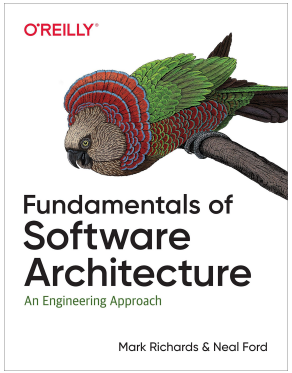


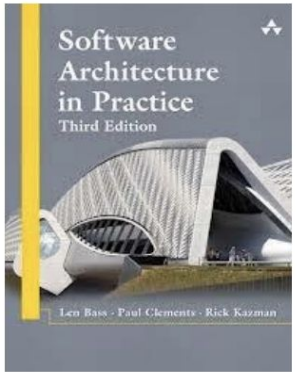


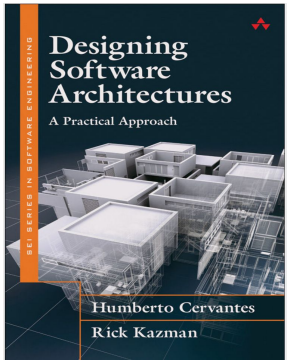


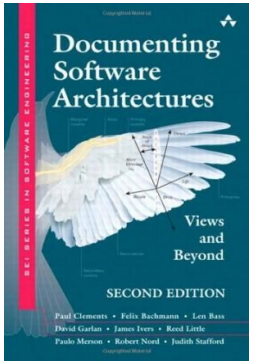












What we know about Software Architecture? - Persons

Martin Fowler

Rick Kazman

Mark Richards

Neal Ford

What we know about Software Architecture? - Organizations

Software Engineering Institute

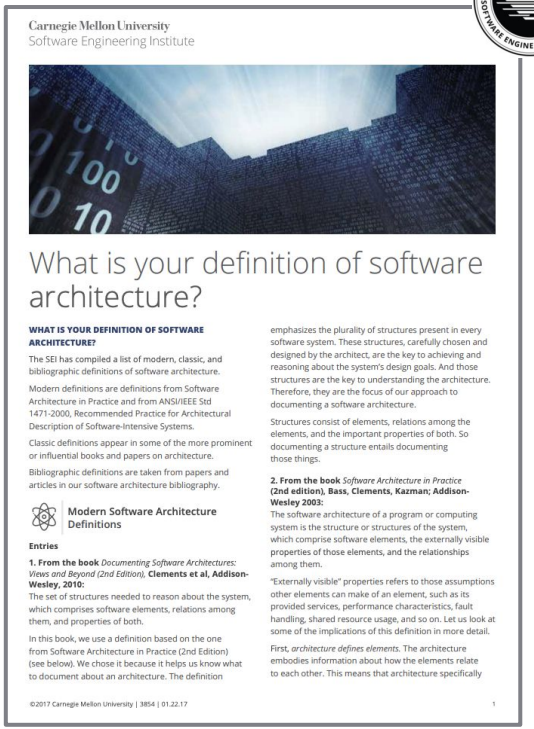
Carnegie Mellon

ISO/IEC/IEEE 42010 Users Group

What is Architecture?

What is Software Architecture?

- **[Martin Fowler]** - “Architecture is about the important stuff. Whatever that is.”
- **[Bass 2003]** - “Architecture - structure or structures for the system, which includes elements, “external visible properties” and relationships among them”
- **[ISO/IEEE 42010]** - “fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution”
- **[UML]** - “Architecture is the organizational structure and associated behavior of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components and subsystems.”
- **[SEI Views & Beyond]** - “The software architecture of a system is the **set of structures** needed to reason about the system, which comprise software elements, relations among them, and properties of both.”



Carnegie Mellon University
Software Engineering Institute

What is your definition of software architecture?

WHAT IS YOUR DEFINITION OF SOFTWARE ARCHITECTURE?

The SEI has compiled a list of modern, classic, and bibliographic definitions of software architecture.

Modern definitions are definitions from Software Architecture in Practice and from ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems.

Classic definitions appear in some of the more prominent or influential books and papers on architecture.

Bibliographic definitions are taken from papers and articles in our software architecture bibliography.

Modern Software Architecture Definitions

Entries

1. From the book *Documenting Software Architectures: Views and Beyond (2nd Edition)*, **Clements et al, Addison-Wesley, 2010:**
The set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both.

In this book, we use a definition based on the one from Software Architecture in Practice (2nd Edition) (see below). We chose it because it helps us know what to document about an architecture. The definition emphasizes the plurality of structures present in every software system. These structures, carefully chosen and designed by the architect, are the key to achieving and reasoning about the system's design goals. And those structures are the key to understanding the architecture. Therefore, they are the focus of our approach to documenting a software architecture.

Structures consist of elements, relations among the elements, and the important properties of both. So documenting a structure entails documenting those things.

2. From the book *Software Architecture in Practice* **(2nd edition), Bass, Clements, Kazman; Addison-Wesley 2003:**
The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

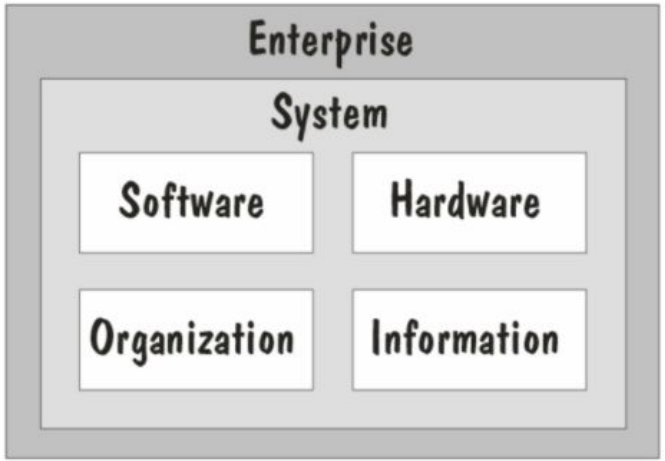
“Externally visible” properties refers to those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. Let us look at some of the implications of this definition in more detail.

First, *architecture defines elements.* The architecture embodies information about how the elements relate to each other. This means that architecture specifically

https://resources.sei.cmu.edu/asset_files/FactSheet/2010_010_001_513810.pdf

Different scope of Architecture?

- Software Architecture
- System Architecture
- Enterprise Architecture



Why you should create Architecture?

- Architecting addresses system qualities
- Architecting helps manage complexity, changes, risks
- Architecting provides a basis for reuse
- Architecting ensures architectural integrity
- Architecting drives communications of stakeholders
- Architecting reduces maintenance costs
- Architecting supports the planning process

Why Software Architecture is important? SEI

1. An architecture will inhibit or enable a system's **driving quality attributes**.
2. The decisions made in an architecture allow you **to reason about** and **manage change** as the system evolves.
3. The **analysis** of an architecture **enables** early prediction of a **system's qualities.**
4. A documented architecture enhances **communication** among stakeholders.
5. The architecture is a **carrier** of the earliest and hence **most fundamental**, **hardest-to-change** design decisions.
6. An architecture **defines a set of constraints** on subsequent implementation.
7. The architecture **influences** the **structure of an organization**, and vice versa.
8. An architecture can provide the **basis for evolutionary**, or even throwaway, prototyping.
9. An architecture is the key artifact that allows the architect and the project manager to reason about **cost and schedule**.
10. An architecture can be created as a **transferable, reusable model** that forms the heart of a product line.
11. Architecture-based development focuses attention on the **assembly of components**, rather than simply on their creation.
12. By restricting design alternatives, architecture channels the creativity of developers, **reducing** design and system **complexity**.
13. An architecture can be the foundation for **training** a new team member.

Benefits of an Enterprise Architecture? TOGAF

- **More effective and efficient business operations:**
  - Lower business operation costs
  - More agile organization
  - Business capabilities shared across the organization
  - Lower change management costs
  - More flexible workforce
  - Improved business productivity

- **More effective and efficient Digital Transformation and IT operations**:
  - Extending effective reach of the enterprise through digital capability
  - Bringing all components of the enterprise into a harmonized environment
  - Lower software development, support, and maintenance costs
  - Increased portability of applications
  - Improved interoperability and easier system and network management
  - Improved ability to address critical enterprise-wide issues like security
  - Easier upgrade and exchange of system components

- **Better return on existing investment, reduced risk for future investment**:
  - Reduced complexity in the business and IT
  - Maximum return on investment in existing business and IT infrastructure
  - The flexibility to make, buy, or out-source business and IT solutions
  - Reduced risk overall in new investments and their cost of ownership

- **Faster, simpler, and cheaper procurement**:
  - Buying decisions are simpler, because the information governing procurement is readily available in a coherent plan
  - The procurement process is faster - maximizing procurement speed and flexibility without sacrificing architectural coherence
  - The ability to procure heterogeneous, multi-vendor open systems
  - The ability to secure more economic capabilities

What is Good Architecture?

PROCESS RECOMMENDATIONS

- should be the product of a single architect / small group of architects with leader;
- architect/architecture team should have the functional requirements for the system and an articulated, prioritized list of quality attributes;
- should be well documented;
- should be circulated to the system's stakeholders;
- should be analyzed for applicable quantitative measures;
- should lend itself to incremental implementation;
- should result in a specific (and small) set of resource contention areas, the resolution of which is clearly specified, circulated, and maintained.

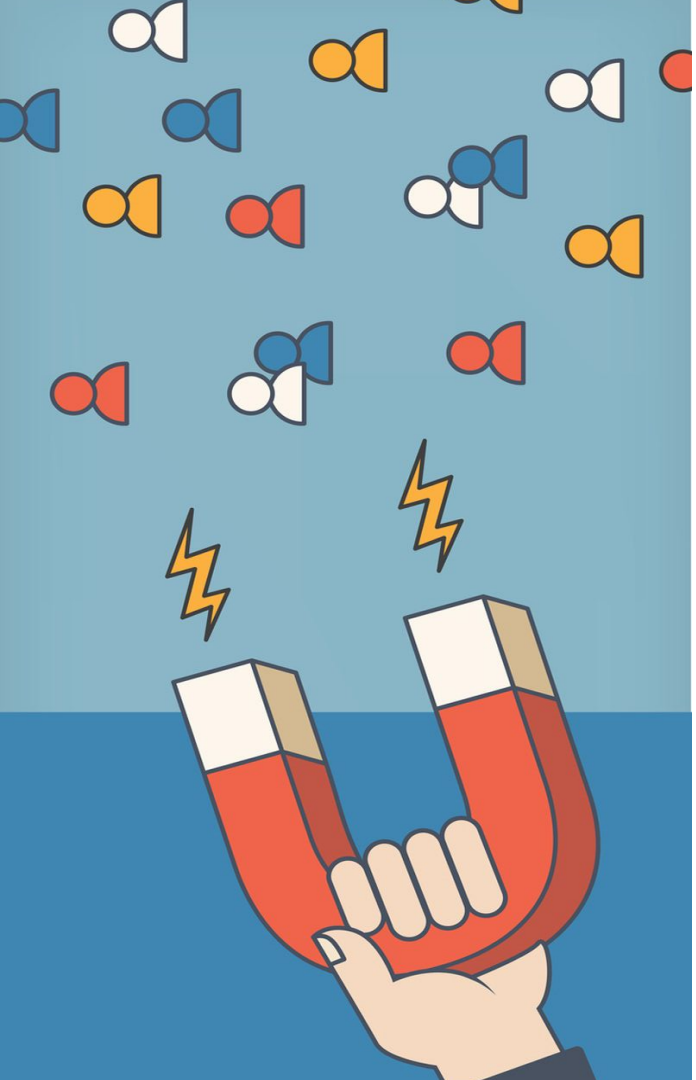
STRUCTURAL RULES OF THUMB

- The architecture should feature well-defined modules whose functional responsibilities are allocated on the principles of information hiding and separation of concerns.
- Each module should have a well-defined interface that encapsulates or "hides" changeable aspects from other software that uses its facilities.
- Quality attributes should be achieved using well-known architectural tactics specific to each attribute.
- The architecture should never depend on a particular version of a commercial product or tool. If it depends upon a particular commercial product, it should be structured such that changing to a different product is straightforward and inexpensive.
- Modules that produce data should be separate from modules that consume data.
- For parallel-processing systems, the architecture should feature well-defined processes or tasks that do not necessarily mirror the module decomposition structure.
- Every task or process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.
- The architecture should feature a small number of simple interaction patterns.

Architect role overview.

Architecture Contexts.

- Solutions Architect
- Application Architect
- Systems Architect
- Enterprise Architect
- Business Architect
- Technical Architect
- Integration Architect
- Information Architect
- Security Architect
- Data Architect
- Network Architect



Architect involvements.

- Being a **bridge** between customer or client and the technical implementation team. **Understands** what a customer wants as well as what a customer needs, because they are not always the same thing
- **Participates** on Workshops, Meetings
- **Creates, documents** and **presents** Architecture to stakeholders
- Contributes to **Roadmap, Team Composition**
- Conducts Architectural **Audits** (Evaluation)
- Be a **mentor** for development team
- Creates **Trade-offs, POC** (codding, setup infrastructure, etc)

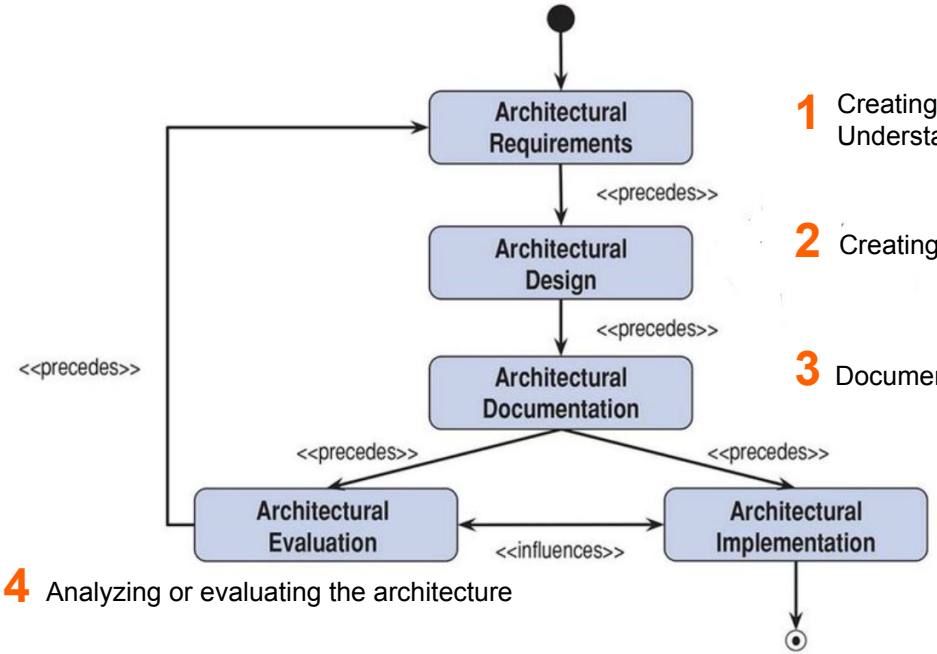
Who is Architect?

- Expert
  - Wide engineering background
  - Delivery methodologies
  - Business Domains
- Analytic
- Communicator
  - Listening
  - Presenting
- Leader / Mentor
- Smart

Architecture Development

Life cycle.

Architecture Development Life cycle.



1 Creating the business case for the system
Understanding the requirements

2 Creating or selecting the architecture

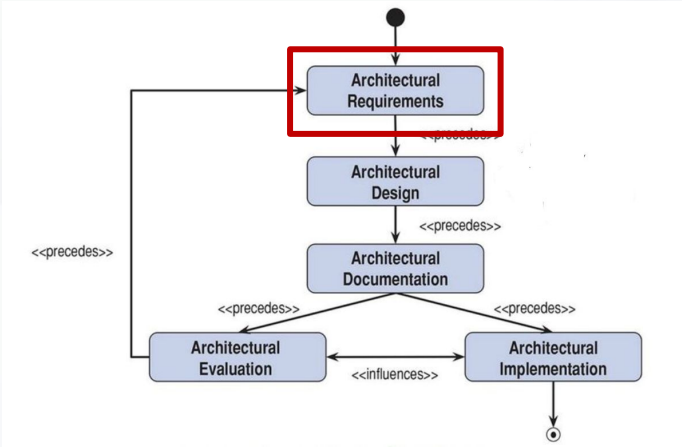
3 Documenting and communicating the architecture

4 Analyzing or evaluating the architecture

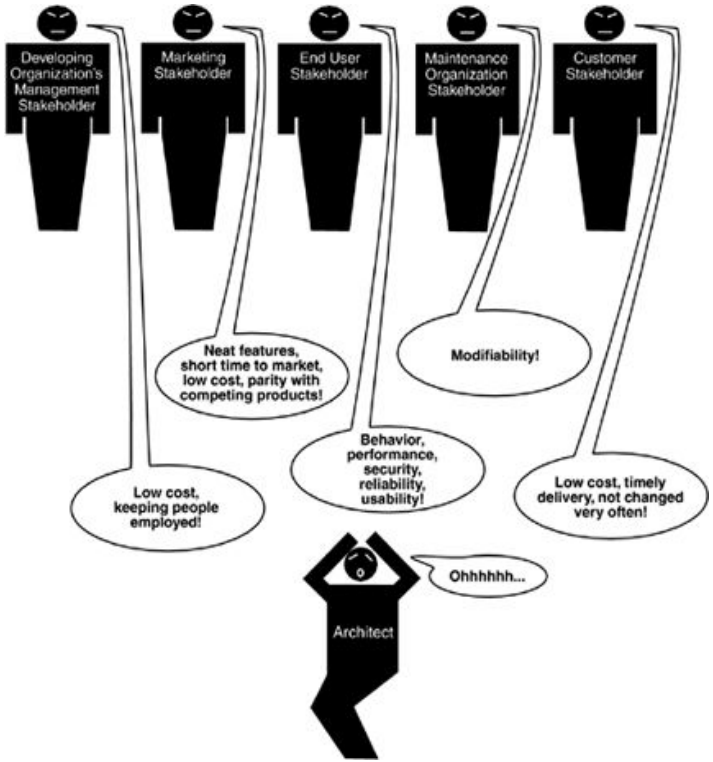
5 Implementing and testing the system based on the architecture

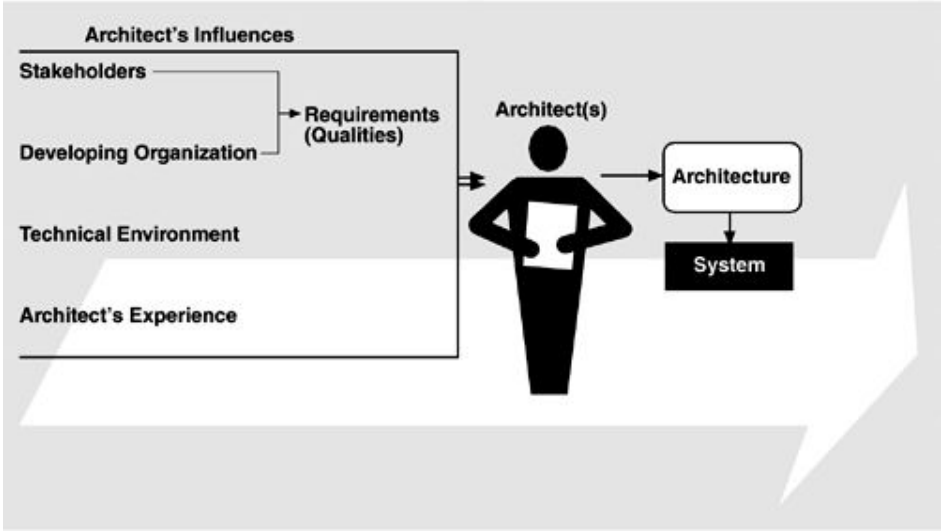
6 Ensuring that the implementation conforms to the architecture

1. Architectural Requirements

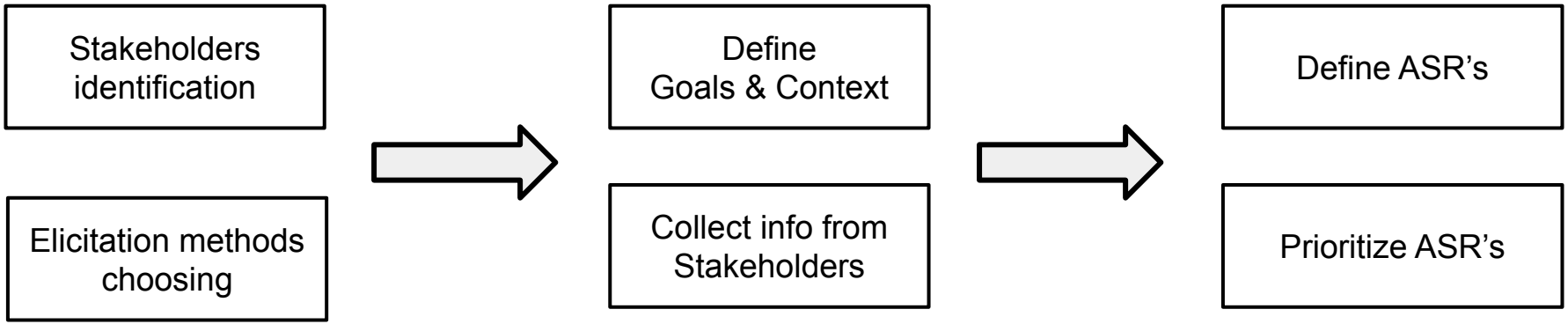


Architecture Requirements. Common challenges





Architectural Requirements. Elicitation Process



Elicitation Techniques.

- Document analysis
- Observation
- Interview
- Prototyping
- Brainstorming Sessions
- Workshop
- Reverse engineering
- Surveys/Questionnaire
- Use Case Approach

Architecture Significant Requirements.

- **Objectives**
- **Functional Requirements** *(answers on question “WHAT” - What system does in this or other case)*
- **Quality Attribute Scenarios** *(answers on question “HOW” - How fast/how secure/how durable/etc.)*
- **Architectural Constraints** *(Specific limitations/restrictions for solution - Use only PostgreSQL 9; Use AWS)*
- **Architectural Concerns** *(Some concerns came from architect/industry/company/etc. experience - typically for such distributed async system we need powerful and detailed logs with log-levels)*

Architecture Significant Requirements.

Functional Requirement

- Drive
- Have roof
- Have space for luggage

Quality Attributes

- Drive fast (performance)
- Consume less fuel (efficiency)
- Not be stolen (security)

Constraints

- Budget is limited
- 5 passengers has to fit in concurrently

Objectives

Pre-Sale

Concerns

There are too many “road holes” - with low wheel profile will brake it more likely. Better to find car with higher wheel profile

ASR - Objectives & Constraints & Concerns.

OBJECTIVES

- Project proposal (pre-sales process as called in the consulting world)
- Exploratory prototype
- During development

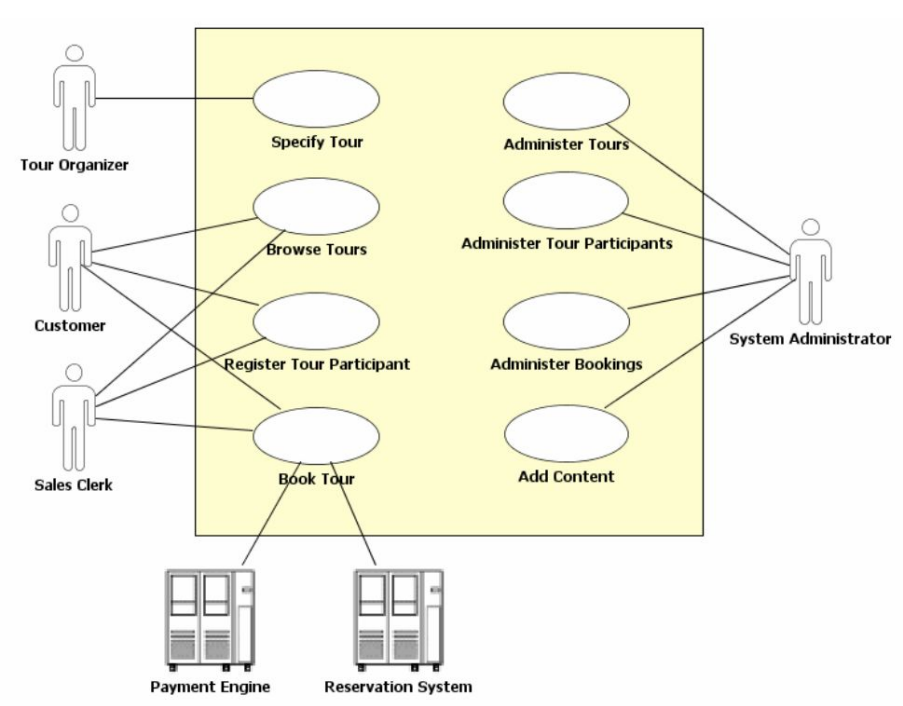
CONSTRAINTS

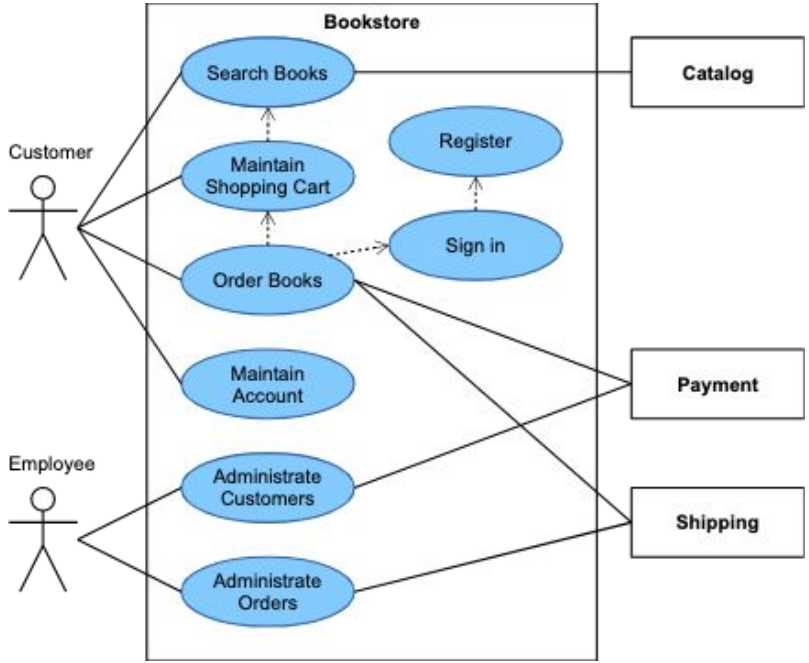
- General concerns
- Specific concerns
- Internal requirements
- Issues
- They may be technical or organizational
- They may originate from the customer but also from the development organization
- Usually limit the alternatives that can be considered for particular design decisions
- They can actually be your “friends”

CONCERNS

- Mandated technologies
- other systems with which your system
- needs to interoperate
- Laws that the system need to be
- compliant with
- Abilities and availability of the developers
- Not negotiable deadlines
- Backward compatibility with older versions of systems …

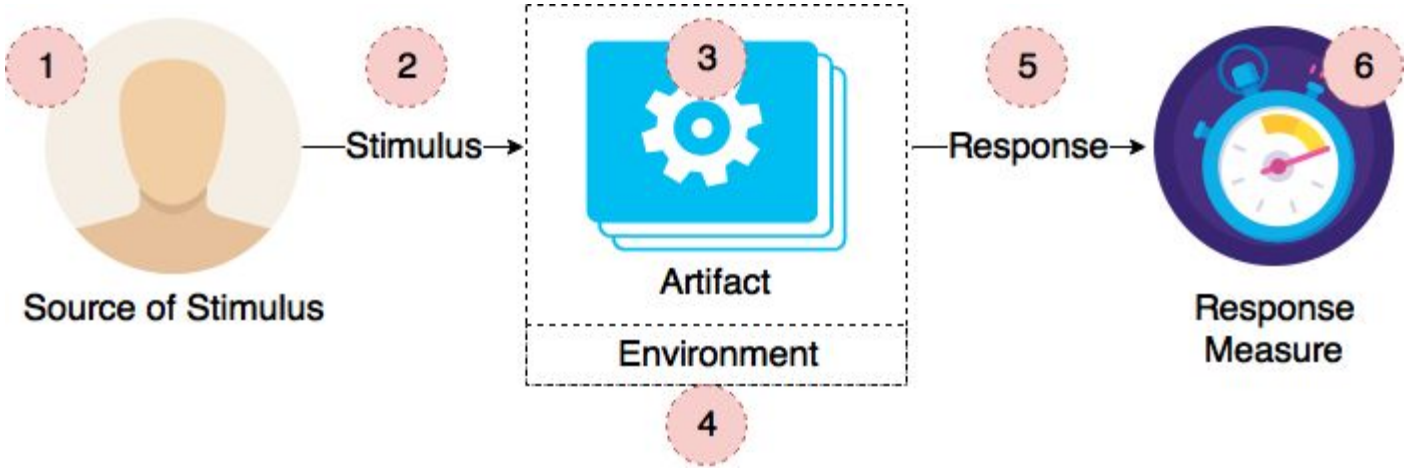
ASR - Functional Requirements.



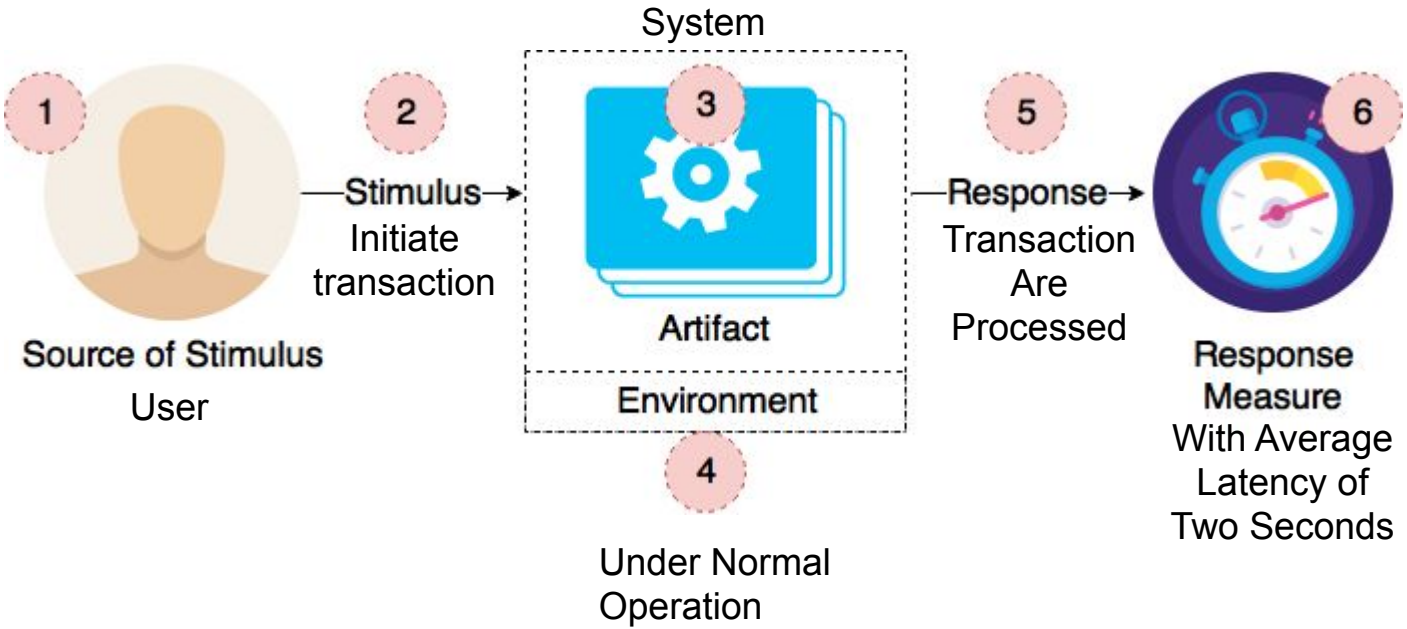


ASR - Quality Attributes. Scenarios

To define a tangible quality attribute, a good starting point is to write a **quality attribute scenario**. A quality attribute scenario is an unambiguous way to specify a testable quality attribute.



ASR - Quality Attributes. Scenarios



ASR - Quality Attributes. ISO/IEC 25010

https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

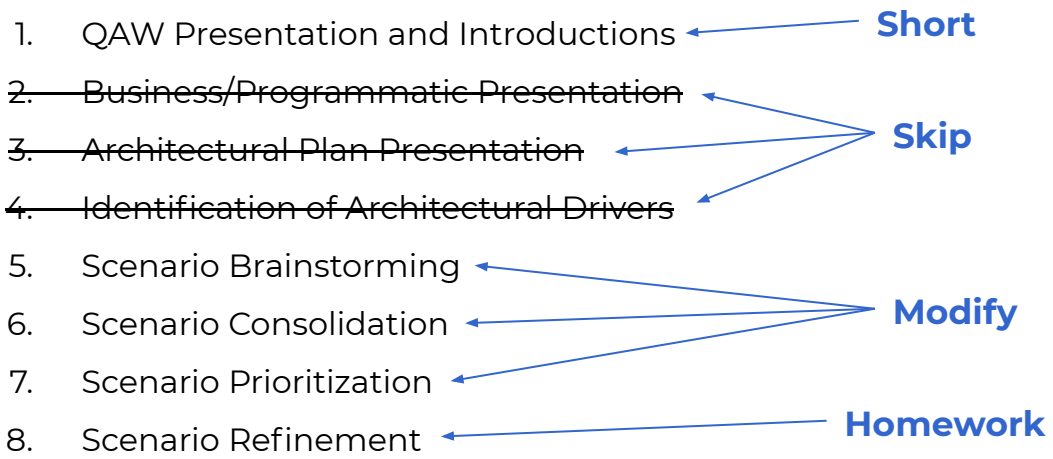
ASR - Quality Attributes. Quality Attributes Workshop

STEPS

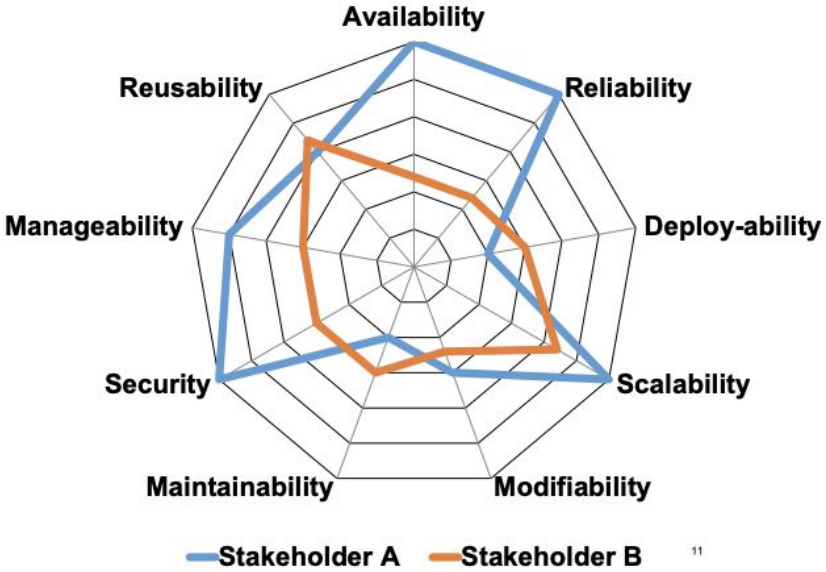
1. QAW Presentation and Introductions
2. Business/Programmatic Presentation
3. Architectural Plan Presentation
4. Identification of Architectural Drivers
5. Scenario Brainstorming
6. Scenario Consolidation
7. Scenario Prioritization
8. Scenario Refinement

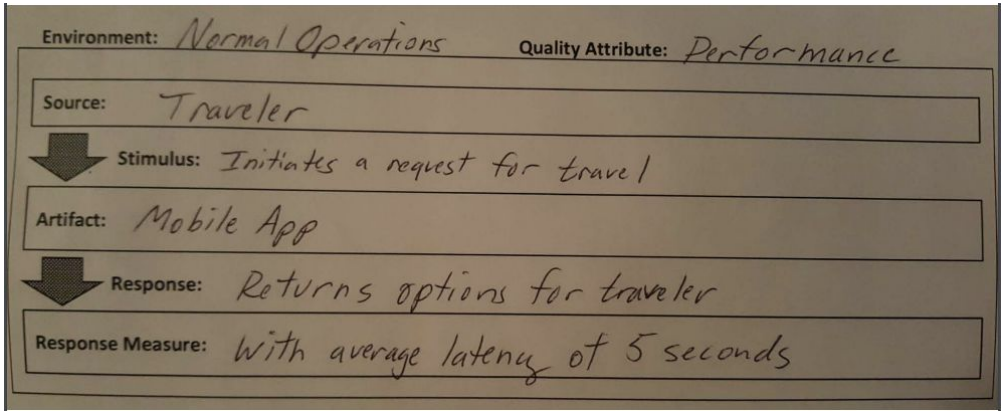
Mini-QAW

1. QAW Presentation and Introductions
2. ~~Business/Programmatic Presentation~~
3. ~~Architectural Plan Presentation~~
4. ~~Identification of Architectural Drivers~~
5. Scenario Brainstorming
6. Scenario Consolidation
7. Scenario Prioritization
8. Scenario Refinement

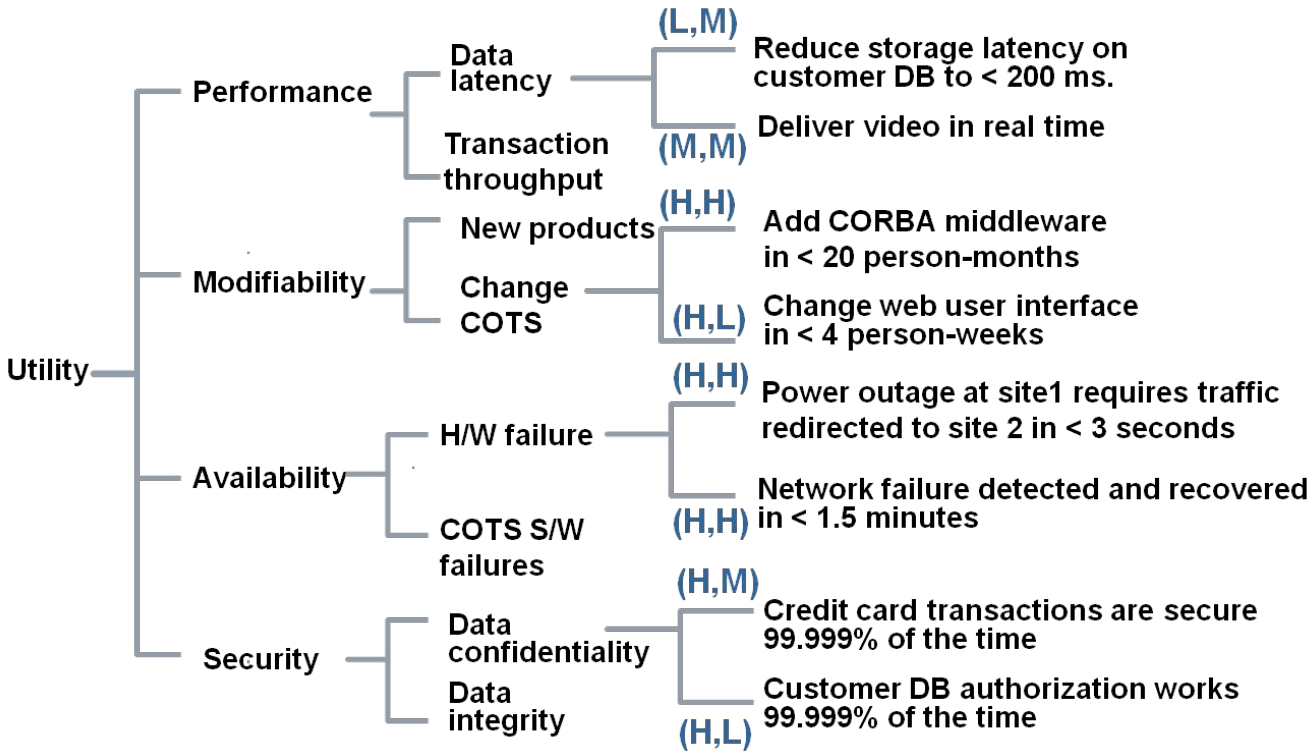


ASR - Quality Attributes. Quality Attributes Workshop

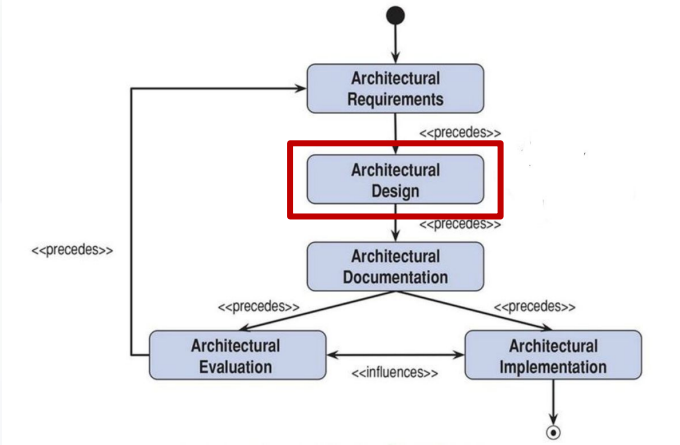




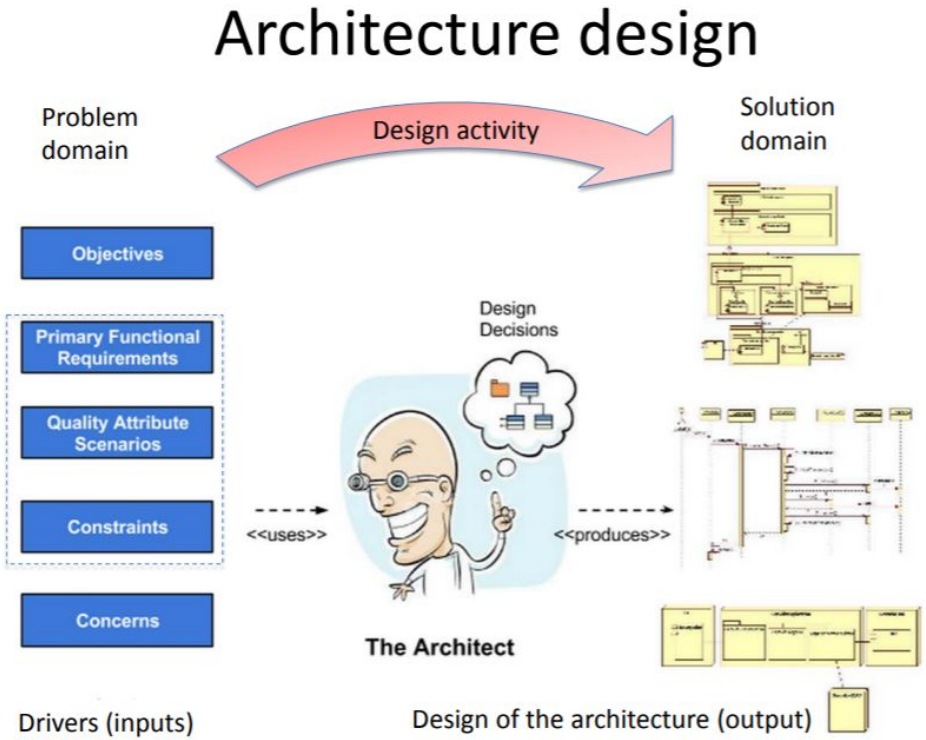
ASR - Quality Attributes. Prioritization



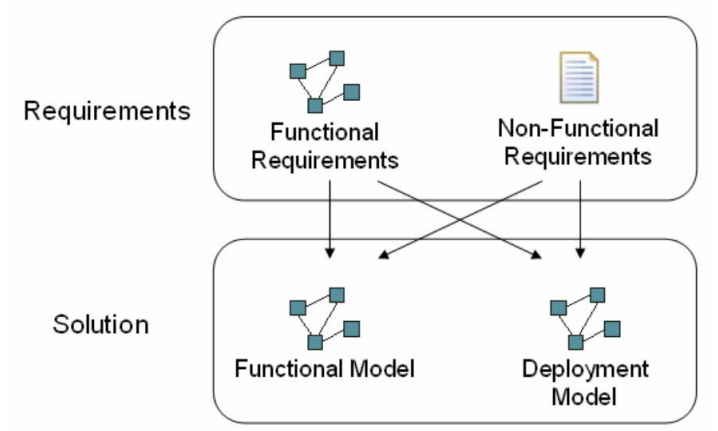
2. Architectural Design



Architecture Design.



From Requirements to Solution

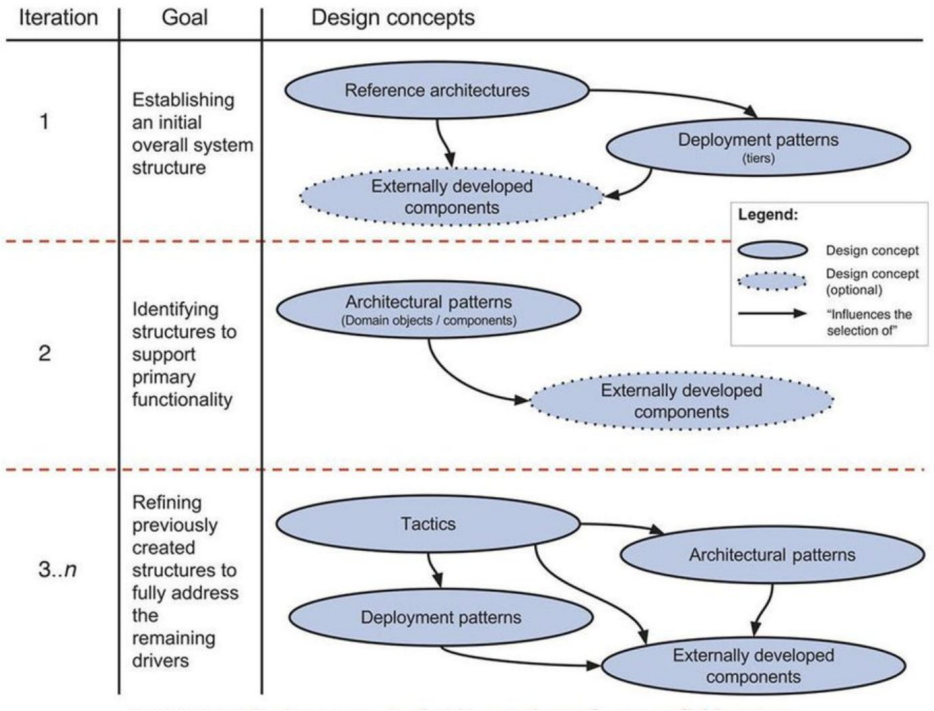


Architecture Design. Section Roadmap

DESIGN CONCEPTS

- Reference Architectures
- Deployment Patterns
- Architectural / Design Patterns
- Tactics
- Externally developed components

  (e.g. Frameworks)

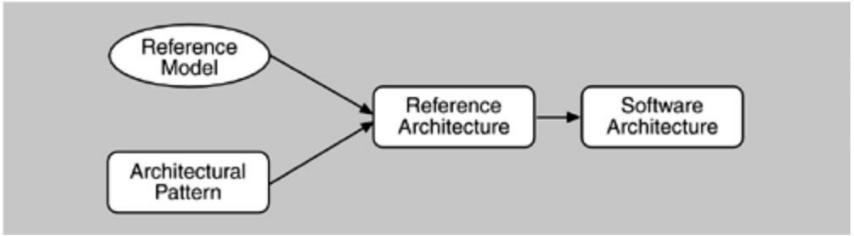


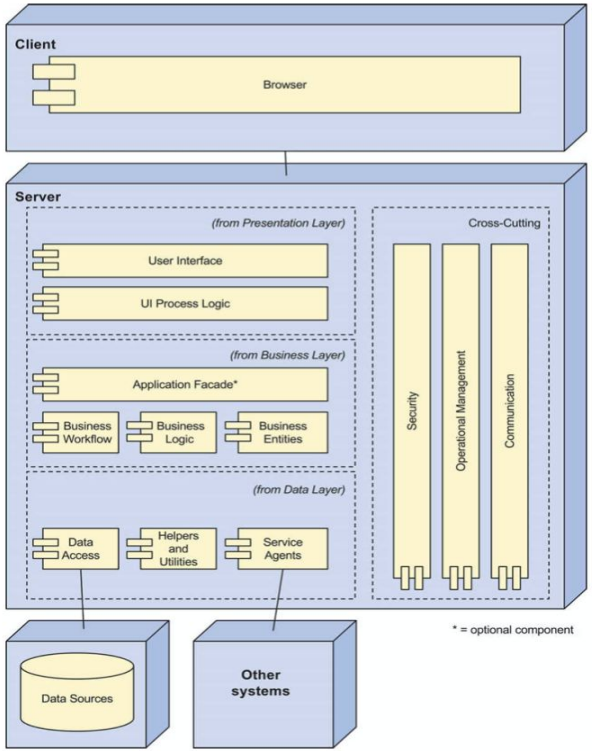
Example reference architecture for the development of web applications from the *Microsoft Application Architecture Guide*

Architecture Design. Reference Architectures

Reference architectures are blueprints that provide an overall logical structure for particular types of applications.

Examples for the enterprise application domain include:
– Mobile applications
– Rich client applications
– Rich internet applications
– Service Applications
– Web applications

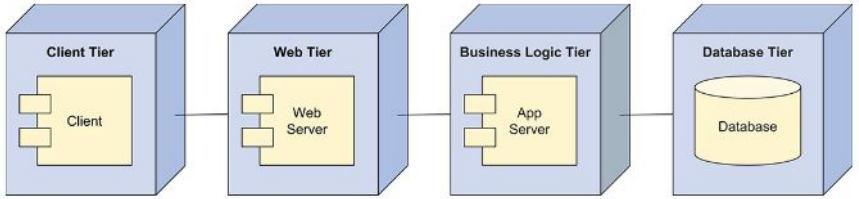




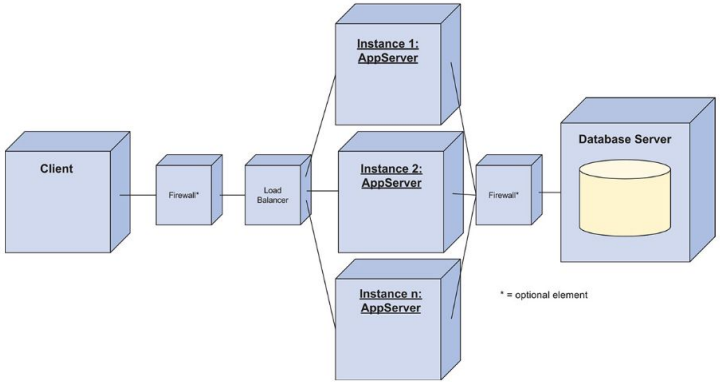
Architecture Design. Deployment Patterns

Deployment pattern - A pattern that provides a model for how to physically structure the system to deploy it.

Examples:
– 2, 3, 4 and n-tier deployment
– Load balanced cluster
– Failover cluster
– Private/public cloud
– Etc…



Four-tier deployment pattern from the *Microsoft Application Architecture Guide* (Key: UML)



Load-Balanced Cluster deployment pattern for performance from the Microsoft Application Architecture Guide (Key: UML)

Architecture Design. Patterns/Styles

PATTERN is:

- Is a package of design decisions that is found repeatedly in practice
- Has known properties that permit reuse
- Describe a class of architectures

PATTERN overview:

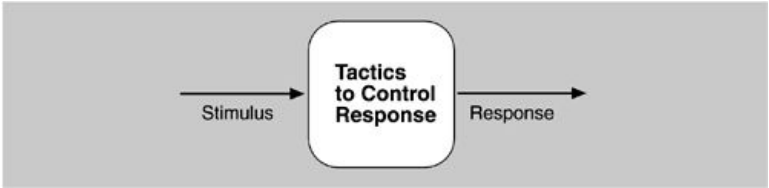
- Context
- Problem
- Solution

Monolith, Microservices, Service-Based Architecture, Service Oriented Architecture, Space-Based Architecture, Hexagonal architecture, ETL, Layered pattern, Client-server pattern, Master-slave pattern, Pipe-filter pattern, Broker pattern, Peer-to-peer pattern, Event-bus pattern, Model-view-controller pattern, Blackboard pattern, Interpreter pattern

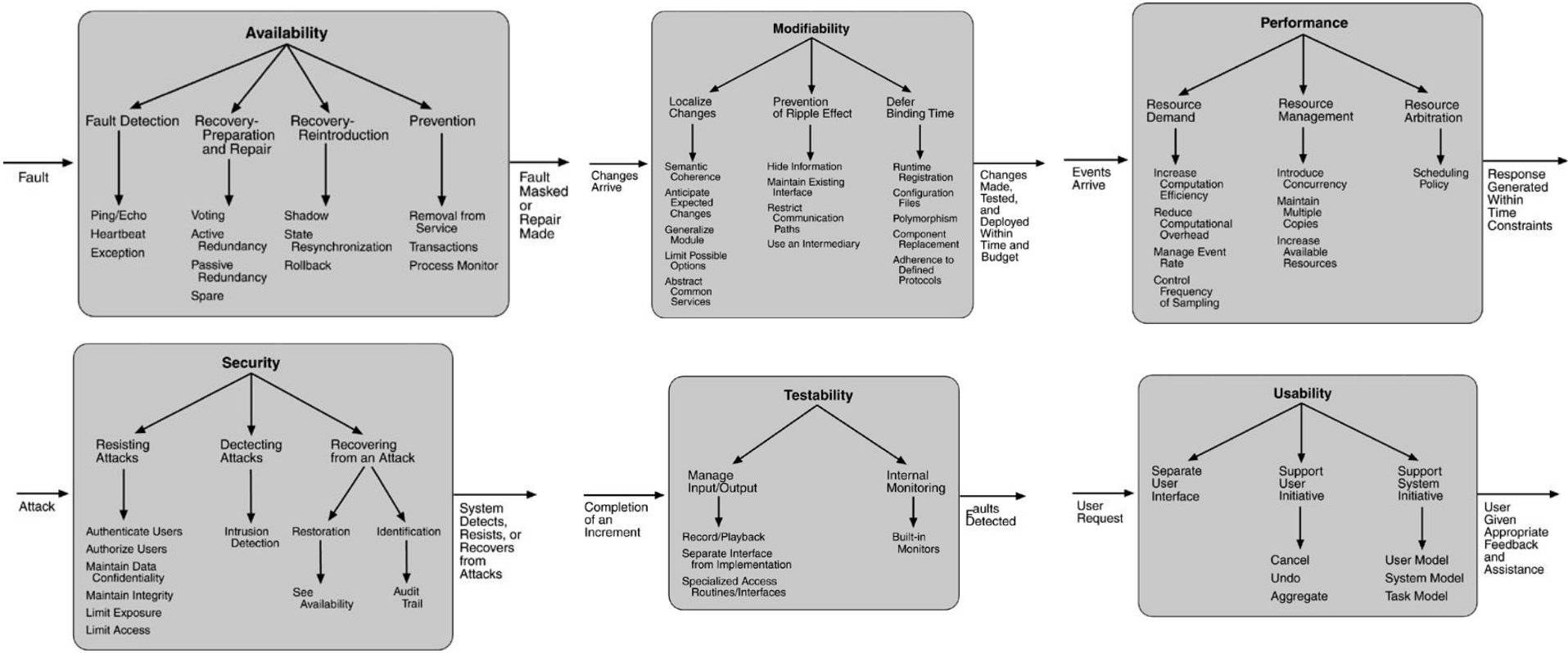
Architecture Design. Analysis of Quality Attributes

	Availability	Efficiency	Flexibility	Integrity	Interoperability	Maintainability	Portability	Reliability
Availability								+
Efficiency			–		–	–	–	–
Flexibility		–		–		+	+	+
Integrity		–			–			
Interoperability		–	+	–			+	
Maintainability	+	–	+					+
Portability		–	+		+	–		
Reliability	+	–	+			+		

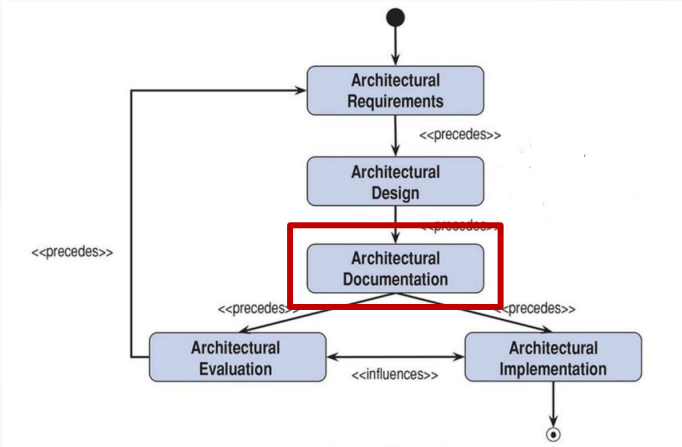
Tactics -
design decisions that influence the control of a quality attribute response.



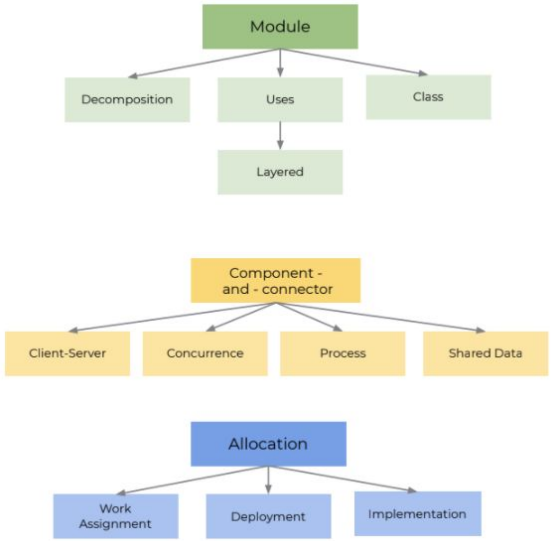
Architecture Design. Quality Attributes Tactics

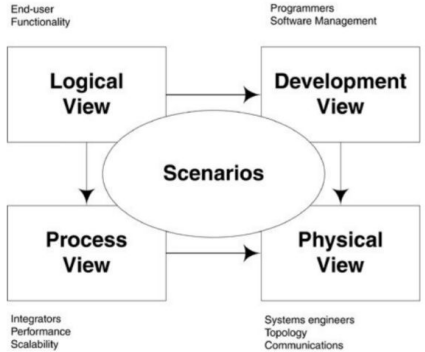


3. Architectural Documentation

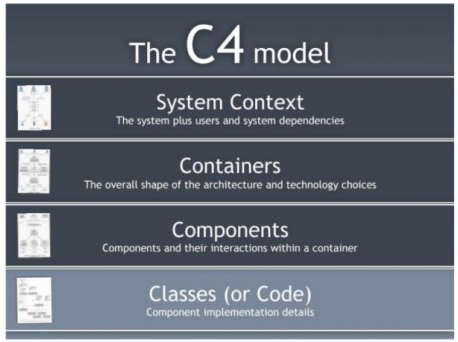


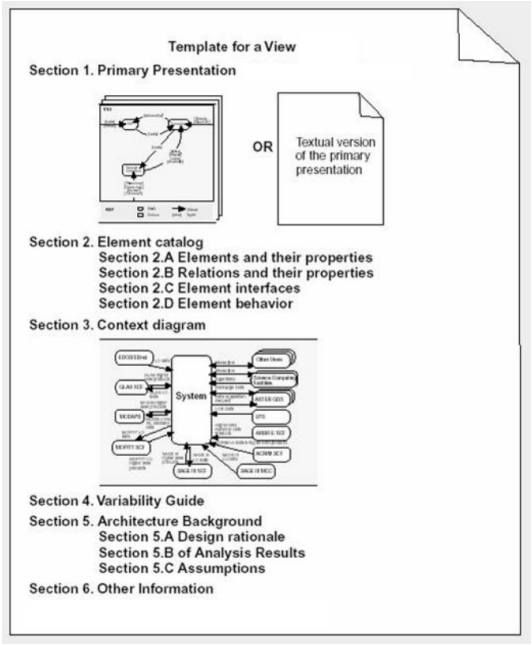
Architectural Documentation.



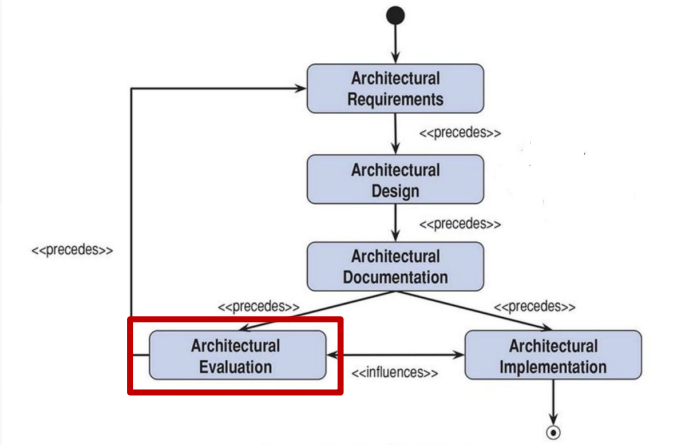


4+1 View Model

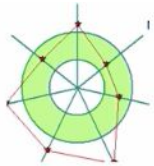




4. Architectural Evaluation



Architectural Evaluation. Focus on Quality Attributes



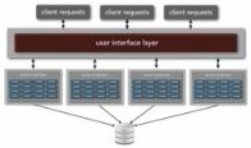
component size

complexity / WMC

coupling (CE, CA, CT)

inheritance depth (DIT)

percent comments



modularity

maintainability

testability

availability

deployment

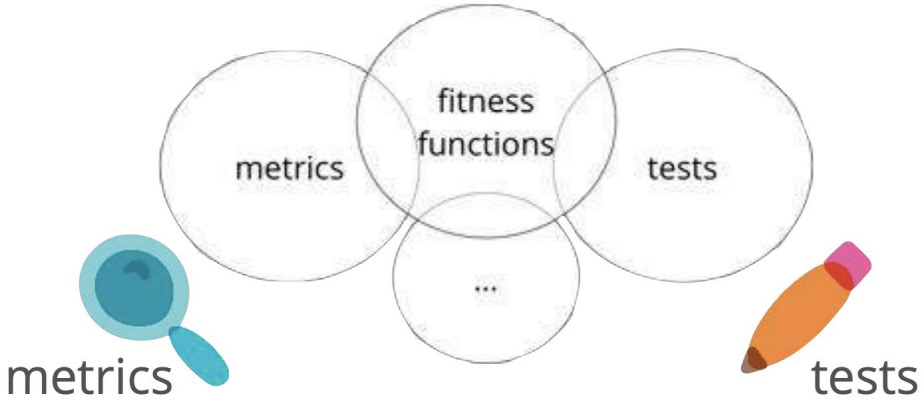
reliability

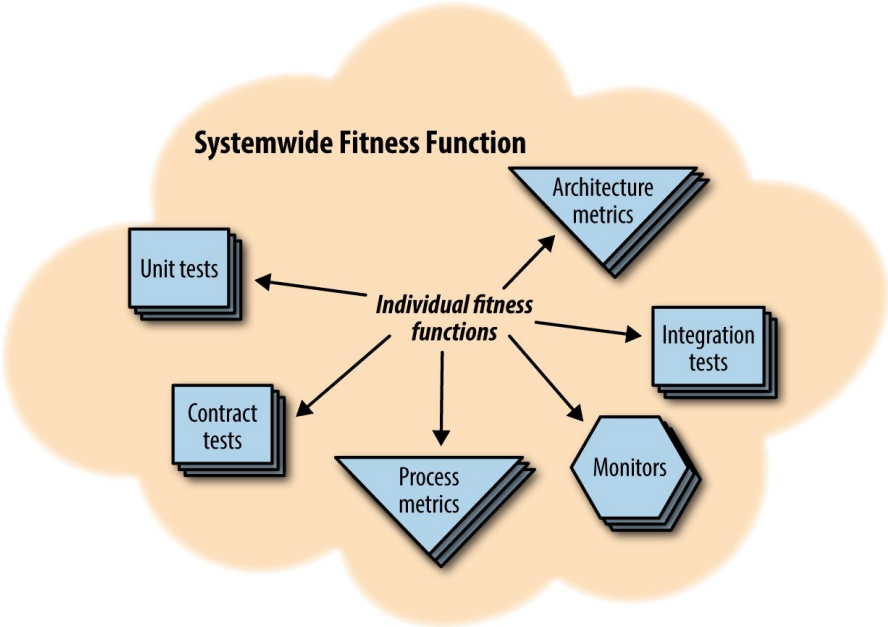
scalability

evolutionary / migration

Architectural Evaluation. Fitness Functions

Fitness Function - is used to summarize how close a given design solution is to achieving the set aims.





Practice I

Q&A session

5 minutes


Break.
10 minutes

Reusable Methods & Standards.

30 minutes

Agenda.

- System and Software Engineering - Architecture description - ISO/IEC/IEEE 42010
- Architecture Frameworks overview
- Architecture Design reusable methods
- Architecture Evaluation reusable methods
- Architecture Katas

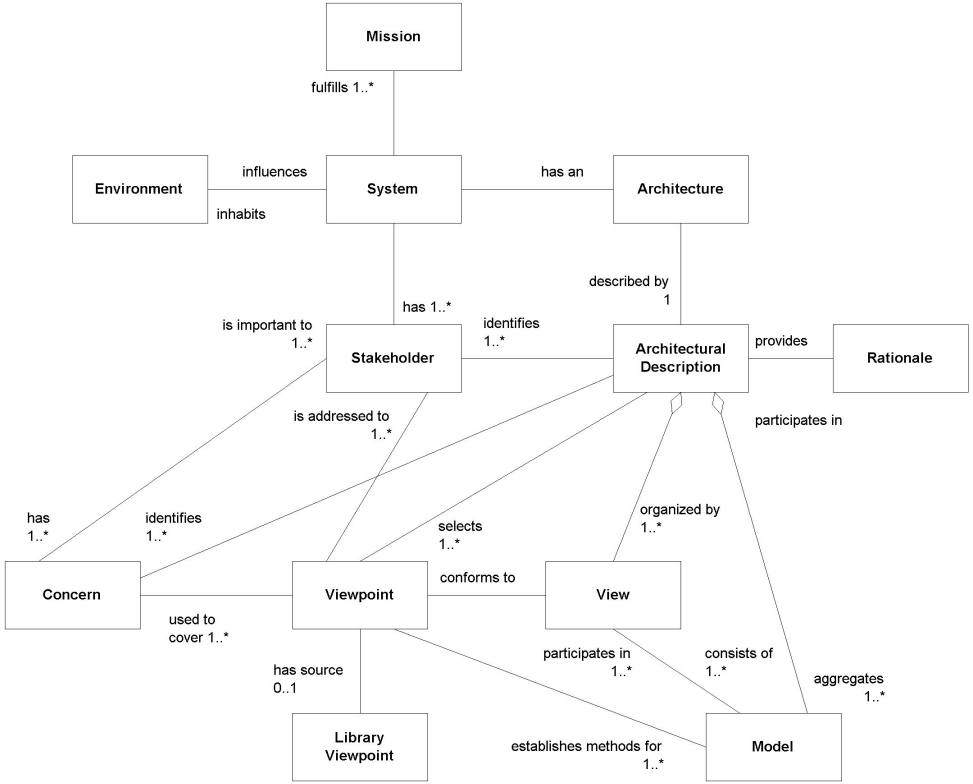
ISO/IEC/IEEE 42010.

ISO/IEC/IEEE 42010. Resource

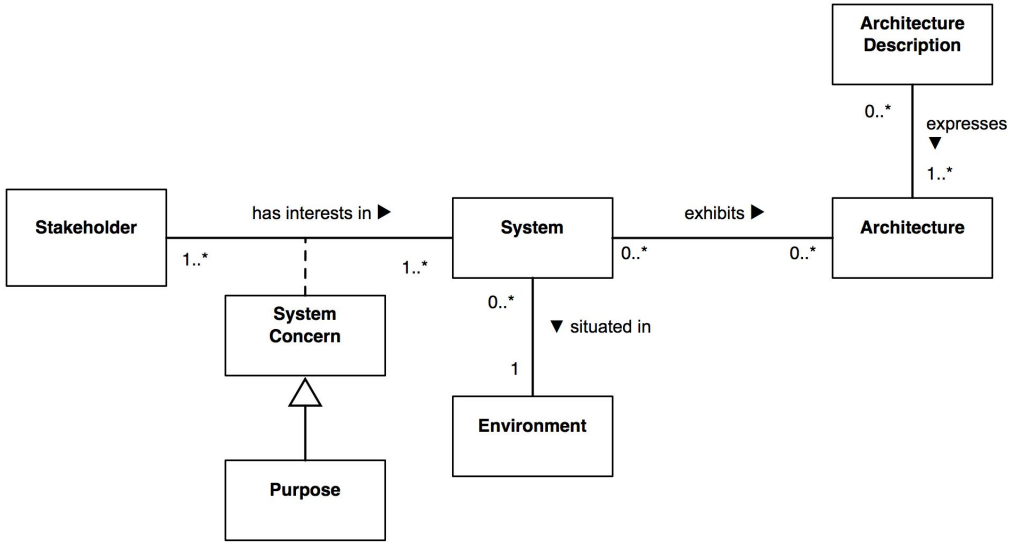
ISO/IEC/IEEE 42010. History

- **1995** - the IEEE Architecture Planning Group (APG) was chartered by the IEEE Software Engineering Standards Committee (*note: SESC is now S2ESC – Software and Systems Engineering Standards Committee*) to set the direction for the next generation of architecture-related standards and practices for the IEEE.
- **22 August 1997** - Design Specification for IEEE 1471 [PDF]. The Design Specification reflects AWG's "contract" with its sponsor, IEEE Software Engineering Standards Committee (SESC) on the form and content of the standard. Approved by SESC,
- **22 October 2000 - Press release** on publication of IEEE-Std-1471-2000
- **11 January 2001 -** IEEE Std 1471 User Group is formed, as an open forum for discussion of IEEE 1471 usage.
- **March 2001 -** IEEE 1471 has been submitted to ANSI for approval as an American National Standard. This submission is routine for all IEEE standards.
- **April 2001 -** "Introducing IEEE 1471", by Mark Maier, David Emery and Rich Hilliard, appears in *IEEE Computer*, volume 34, number 4 (April 2001).
- **7 August 2001** ANSI makes IEEE 1471 an American National Standard.
- **8 July 2005** IEEE 1471 could become an **ISO standard**.
- **2011** - ISO/IEC/IEEE 42010 was approved for use and published in 2011

IEEE 1471-2000.



ISO/IEC/IEEE 42010. Conceptual Model



Systems exist.

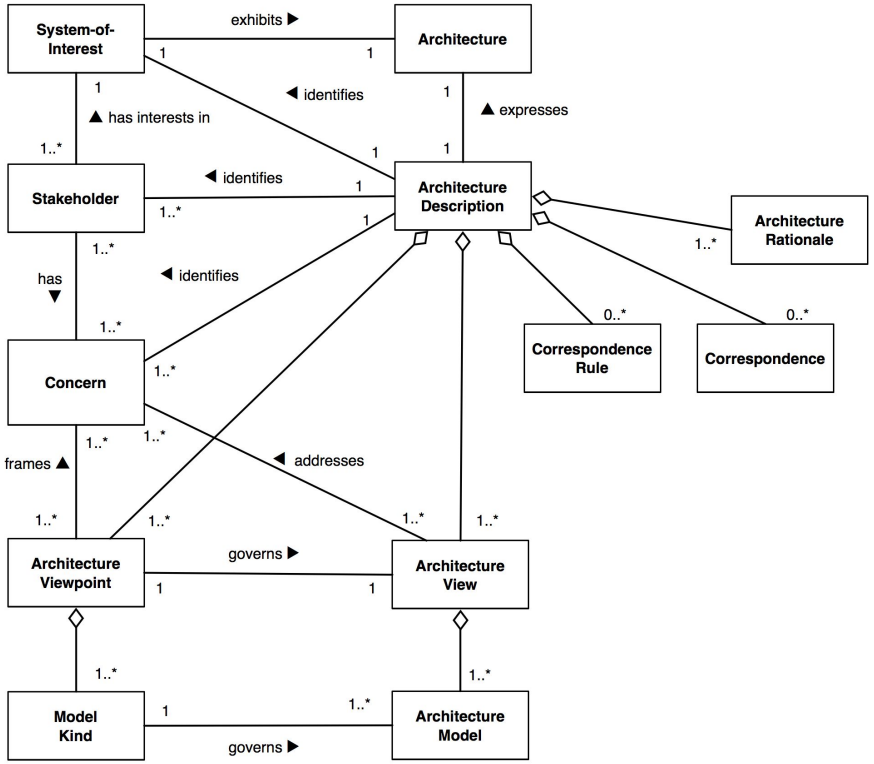
A System is situated in its **Environment**. That environment could include other Systems.

Stakeholders have interests in a System; those interests are called **Concerns**. A **System's Purpose** is one very common Concern.

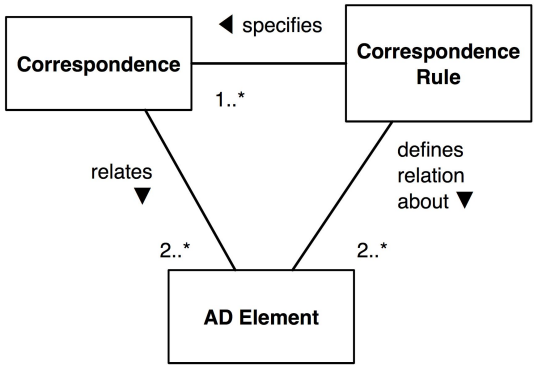
Systems have **Architectures**.

An **Architecture Description** is used to express an Architecture of a System.

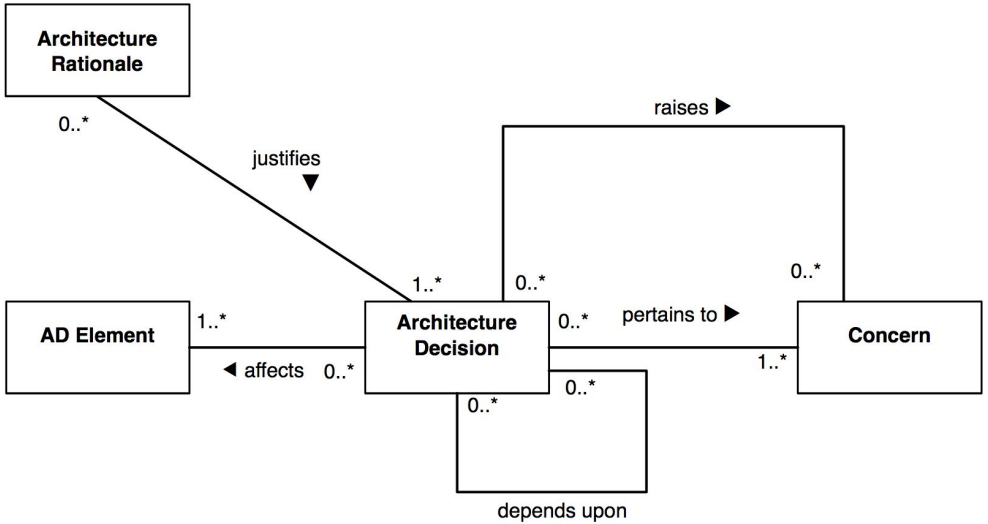
ISO/IEC/IEEE 42010. Core of Architecture Description



ISO/IEC/IEEE 42010. Core of Architecture Description

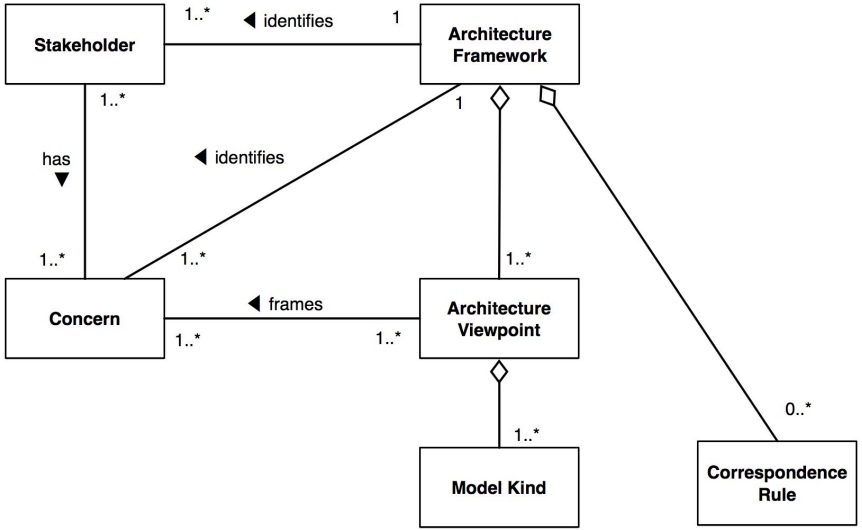


AD Elements and Correspondences



Architecture Decisions and Rationale

ISO/IEC/IEEE 42010. Architecture Framework & ADLs



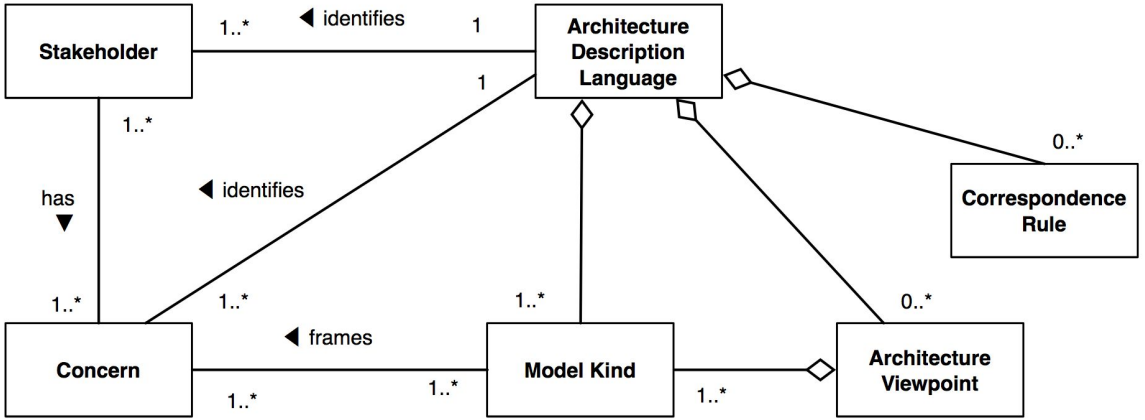
An architecture framework establishes a common practice for creating, interpreting, analyzing and using architecture descriptions within a particular domain of application or stakeholder community.
architecture framework: conventions, principles and practices for the description of architectures established within a specific domain of application and/or community of stakeholders

Requirements on Frameworks

An architecture framework **conforms** to the International Standard (IS) when it specifies:

1. information identifying the framework;
2. one or more concerns;
3. one or more stakeholders having those concerns;
4. one or more architecture viewpoints (and their specifications conforming to the IS);
5. correspondence rules, integrating the viewpoints;
6. conditions on applicability;
7. consistency of the framework with the provisions of the ISO/IEC/IEEE 42010 conceptual model. (see also: Meta model matters)

ISO/IEC/IEEE 42010. ADL



An ADL is ***any form of expression*** for use in Architecture Descriptions.
An ADL might include a single Model Kind, a single viewpoint or multiple viewpoints.
Examples of ADLs: Rapide, SysML, ArchiMate, ACME, xADL.

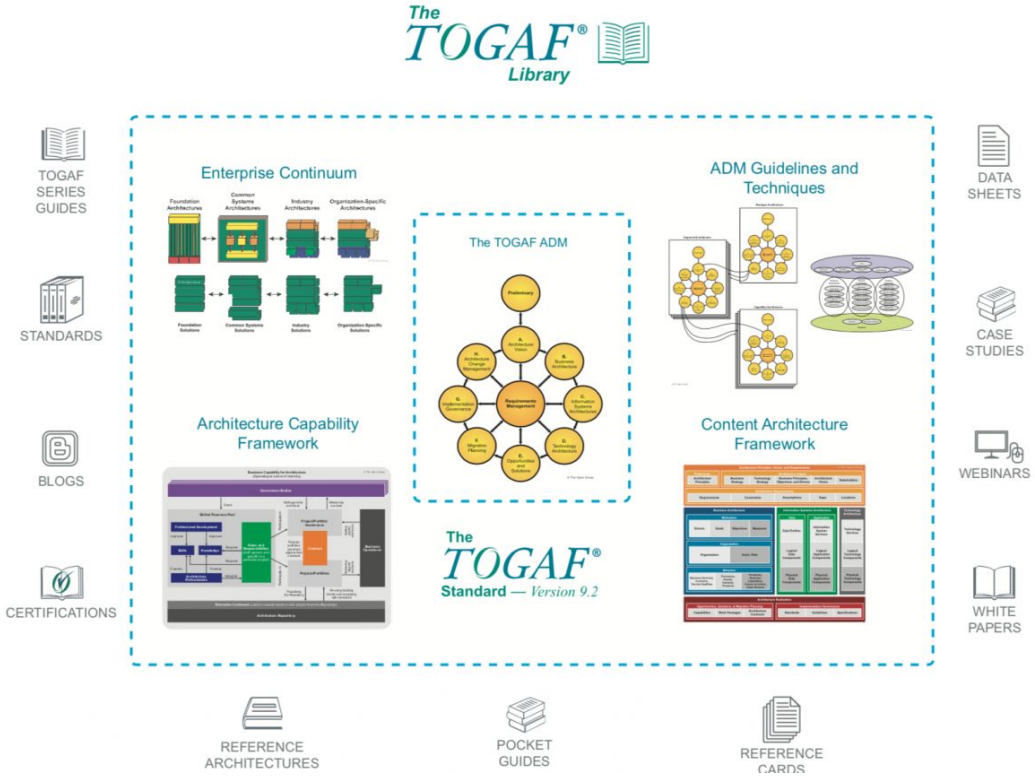
ISO/IEC/IEEE 42010. Architecture Framework survey

BDAF - Big Data Architecture Framework
DoDAF - US Department of Defense Architecture Framework
GEAF - Gartner's Enterprise Architecture Framework
4+1 - Kruchten's 4+1 view model
MODAF - (UK) Ministry of Defence Architecture Framework
NAF - NATO C3 Systems Architecture Framework
TOGAF - The Open Group Architecture Framework
ZF - Zachman Framework

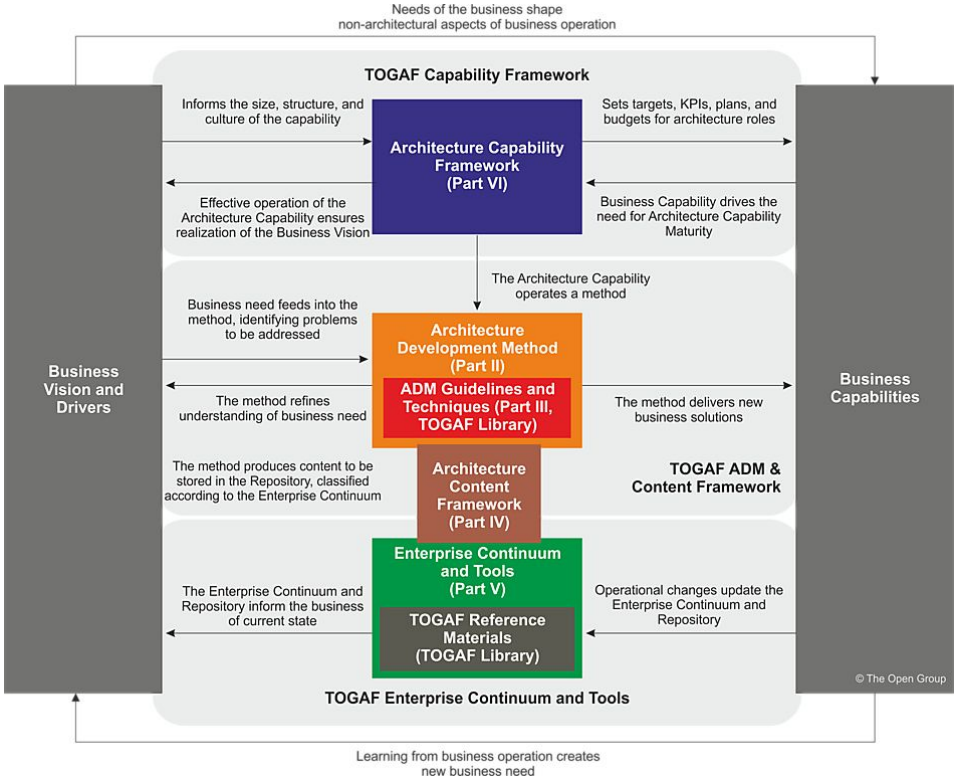
ID	Name	Purpose	Scope	Classifiers	Notes
AF-EAF	Air Force Enterprise Architecture Framework	"The AF Enterprise Architecture Framework (AF-EAF) provides a logical structure for classifying, organizing and relating the breadth and depth of information that describes and documents the Air Force Enterprise Architecture (AF-EA)."	Air Force IT systems	Communication, Guidance, Enterprise Architecture Descriptions	"The AF-EAF does not define the AF-EA content, rather it consists of various approaches, models, and definitions for communicating and facilitating the presentation of key architecture components (i.e. architecture vision, governance, principles, guidance, products, etc.) required for the development and integration of AF architectures. The AF-EAF establishes a common foundation for understanding, comparing and integrating architectures and as such provides the overarching guidance for generating AF architectures." [All quotes from AF-EAF v2.01, 6 June 2003]
AFIoT	IEEE Std 2413–2019, IEEE Standard for an Architectural	"An architecture framework description for the Internet of Things			"The architectural framework for IoT provides a reference model that defines relationships

TOGAF.

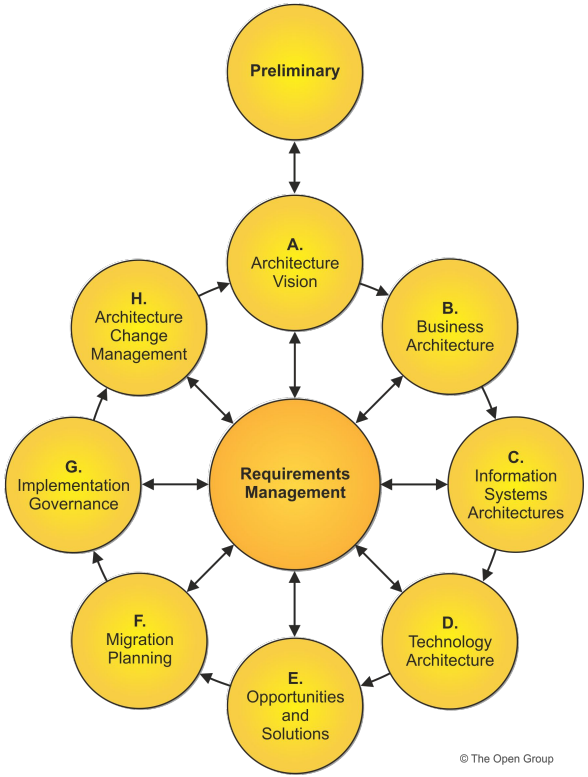
TOGAF.

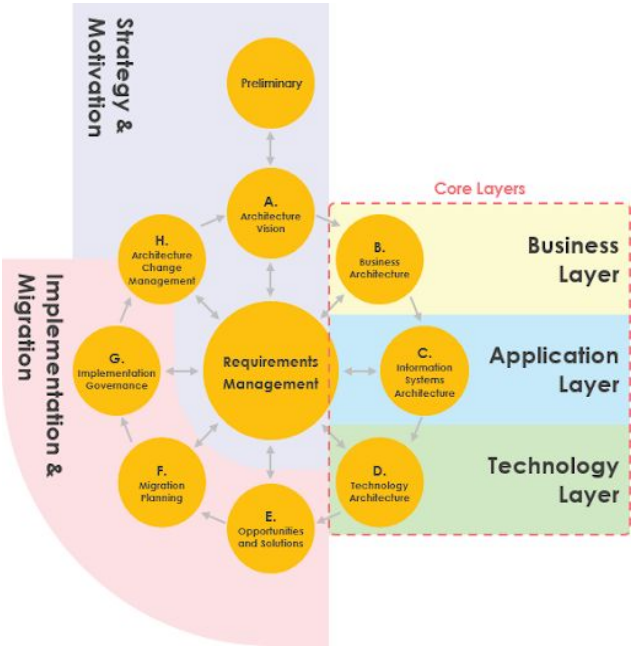

The TOGAF® Library
TOGAF SERIES GUIDES
STANDARDS
BLOGS
CERTIFICATIONS
Enterprise Continuum
The TOGAF ADM
ADM Guidelines and Techniques
Architecture Capability Framework
Content Architecture Framework
The TOGAF® Standard — Version 9.2
DATA SHEETS
CASE STUDIES
WEBINARS
WHITE PAPERS
REFERENCE ARCHITECTURES
POCKET GUIDES
REFERENCE CARDS

TOGAF.


Needs of the business shape
non-architectural aspects of business operation
TOGAF Capability Framework
Informs the size, structure, and culture of the capability
Sets targets, KPIs, plans, and budgets for architecture roles
Architecture Capability Framework (Part VI)
Effective operation of the Architecture Capability ensures realization of the Business Vision
Business Capability drives the need for Architecture Capability Maturity
The Architecture Capability operates a method
Business need feeds into the method, identifying problems to be addressed
Architecture Development Method (Part II)
ADM Guidelines and Techniques (Part III, TOGAF Library)
Business Vision and Drivers
The method refines understanding of business need
The method delivers new business solutions
Business Capabilities
The method produces content to be stored in the Repository, classified according to the Enterprise Continuum
Architecture Content Framework (Part IV)
TOGAF ADM & Content Framework
Enterprise Continuum and Tools (Part V)
The Enterprise Continuum and Repository inform the business of current state
Operational changes update the Enterprise Continuum and Repository
TOGAF Reference Materials (TOGAF Library)
© The Open Group
TOGAF Enterprise Continuum and Tools
Learning from business operation creates new business need

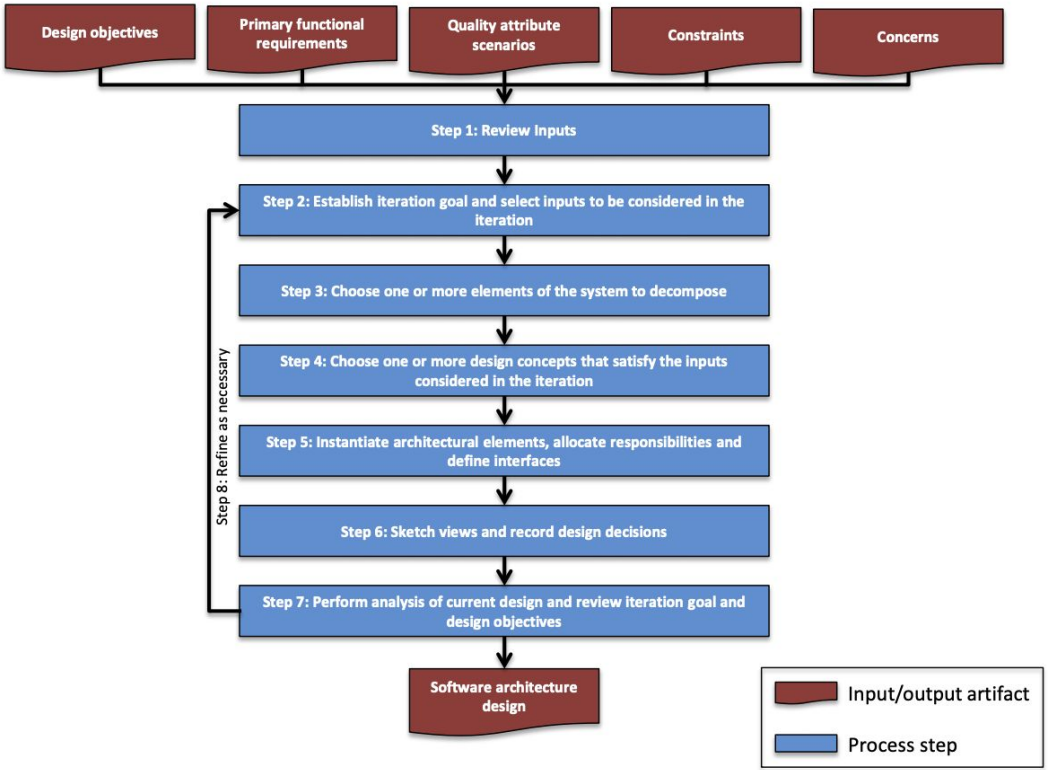
TOGAF. Architecture Development Method



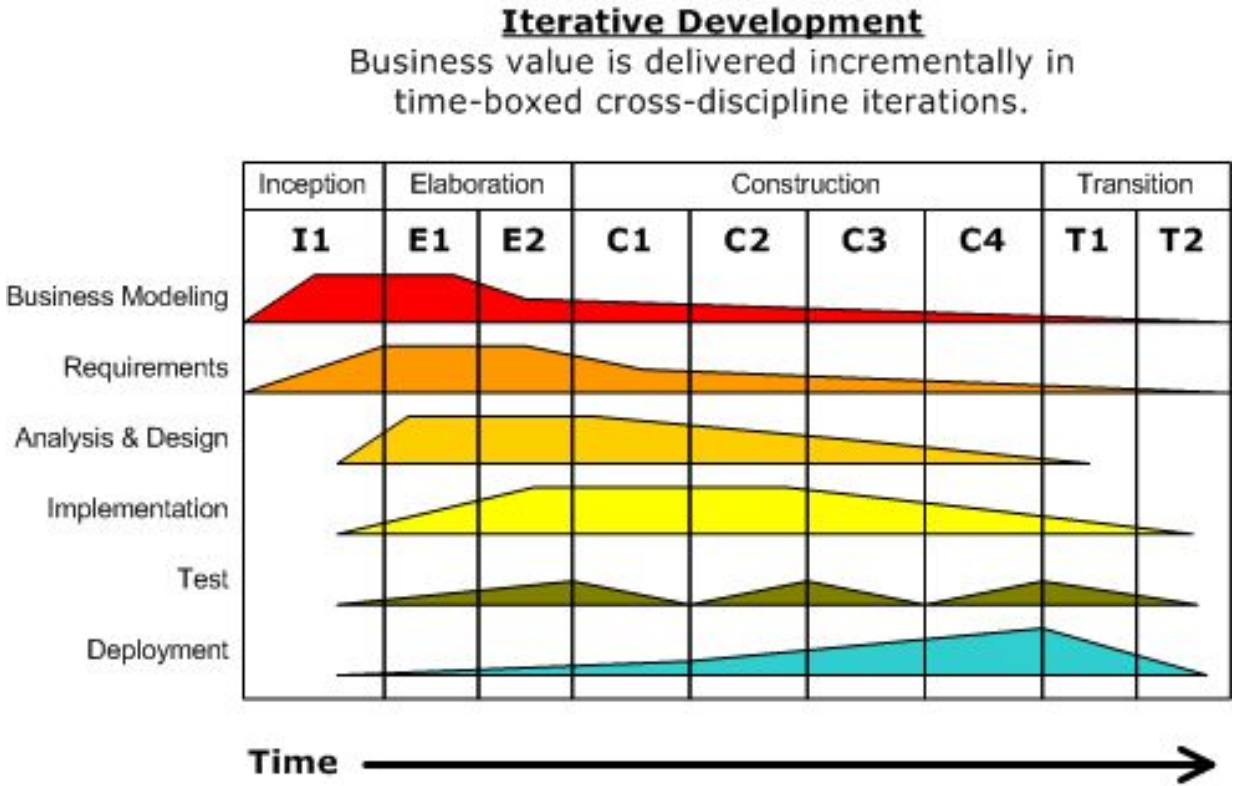


Architecture Design
reusable methods.

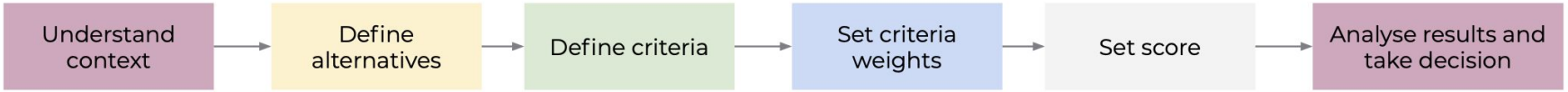
Architecture Design. Attribute Driven Design (ADD 3.0)



Architecture Design. RUP



Architecture Design. Trade offs



	agility	deployment	testability	performance	scalability	simplicity	cost
Monolithic	-	-	+	-	-	+	$
Microservices	+	+	+	-	-	+	$$
Space-based	+	+	-	+	+	-	$$$$
Service-oriented	-	-	-	-	+	-	$$$$
Service-Based	+	+	+	-	+	-	$$

Architecture Evaluation
reusable methods.

Architectural Evaluation. Reusable Methods

International Journal of Computer Applications (0975 – 8887)
Volume 49– No.16, July 2012

Software Architecture Evaluation Methods – A survey

P. Shanmugapriya,
Research Scholar,
Department of CSE,
SCSVMV University,
Enathur, Tamilnadu,INDIA

R. M. Suresh
Principal,
Jerusalem College of Engineering,
Chennai, Tamilnadu,INDIA

ABSTRACT
Software architectural evaluation becomes a familiar practice in software engineering community for developing quality software. Architectural evaluation reduces software development effort and costs, and enhances the quality of the software by verifying the addressability of quality requirements and identifying potential risks and it provides assurance to developers that their chosen architecture will meet both functional and non-functional quality requirements. This paper presents a discussion on different software architectural evaluation methods and techniques and concentrates on summarizing the importance of the different early and late evaluation methods, similarities and difference between them, their applicability, strengths and weaknesses.

Keywords
Software architectural, evaluation, early and late evaluation methods

1. INTRODUCTION
Software architecture evaluation is a technique or method which determines the properties, strengths and weaknesses of software architecture or software architectural style or a design pattern. Software architectural evaluation provides assurance to developers that their chosen architecture will meet both functional and non-functional quality requirements. An architectural evaluation should provide more benefits than the cost of conducting the evaluation itself [1]. Software architectural evaluation ensures increased understanding and documentation of the system, detection of problems with existing architecture, and enhanced organizational learning. Several methods and techniques have been proposed for software architectural evaluation. Among those *scenario-based approaches* are considered quite mature [17, 6]. There are also *attribute model-based methods* [25] and *quantitative models* [39] for software architecture evaluation. Other approaches have been developed to systematically justify the properties of architectural styles [25] and design patterns [33, 20]. The goal of this paper is to review existing software architectural evaluation methods and to classify the methods in the form of taxonomy. The presented taxonomy also considers two phases of a software life cycle: early and late.

2.EVALUATION METHODS
A number of evaluation methods have been developed which are applicable in different phases of the software development cycle. The main two opportunities for evaluation are before and after implementation [15]. Early software architecture evaluation methods are applied to software architecture before its implementation. Quality goals can primarily be achieved if the software architecture is evaluated with respect to its specific quality requirements at the early stage of software development. Late software architecture evaluation methods identify the difference between the actual and planned architectures. These methods provide useful guidelines of how to reconstruct the actual architecture, so that it conforms to the planned architecture.

3.EARLY EVALUATION METHODS
Early software architectural evaluation can be conducted on the basis of the specification and description of the software architecture. The scenario-based approaches are flexible and simple [5, 6]. Mathematical model-based evaluation techniques for assessing the operational quality attributes, such as reliability and performance are also well used, particularly in real-time software systems.

3.1Scenario-based Software Architecture Evaluation Methods
Scenario-based evaluation methods evaluate software architecture's ability with respect to a set of scenarios of interest. Scenario is brief descriptions of a single interaction of a stakeholder with a system [8]. Different scenario based methods have been developed so far [22, 23, 29, 31, 11, 38, 40, 44].
The scenario-based evaluation methods offer a systematic means to investigate software, architecture using scenarios. These methods determine whether software architecture can execute a scenario or not. Evaluation team explores/maps the scenario onto the software architecture to find out the desired architectural components and their interactions, which can accomplish the tasks expressed through the scenario. If the software architecture fails to execute the scenario, these methods list the changes to the software architecture required to support the scenario and estimate the cost of performing the changes.
Scenario-based evaluation methods require presence of relevant stakeholders to elicit scenarios according to their requirements. Scenario-based methods can ensure discovery of problems in software architectures from different point of views by incorporating multiple stakeholders during the scenario elicitation process whereas the end user can indicate the performance issues. The following are the set of Scenario-based evaluation methods

Fig – 1: Common behavior in scenario-based evaluation methods

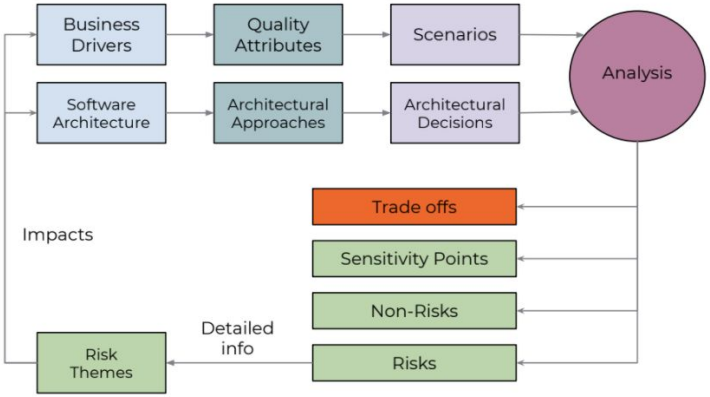
Table 1. Comparison of the various scenario-based evaluation methods

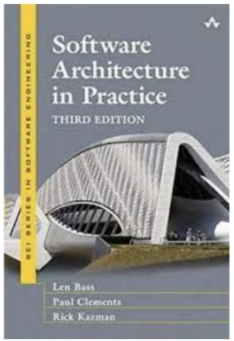
Evaluation Method	Main objective	Steps in Evaluation	Scenario classification & impact analysis	Approaches used	Objects analyzed	Addressed QAs
SAAM	Architectural suitability and risks analysis	Six activities, some activities carried out in parallel- includes no preparation activities	Direct and indirect scenarios. Counts the number of components affected by the scenarios	Scenario elicitation via brainstorming with Stakeholders. Mapping scenarios onto SAs to verify functionality or estimate change cost	Architectural documentation, especially, showing the logical views	Mainly modifiability but can be adapted for others
ATAM	Sensitivity and tradeoff analysis	Nine activities, some activities carried out in parallel; includes preparation activities	Use-case, growth and Exploratory scenarios. Counts the sensitivity points and Tradeoff points	Creation of utility tree to elicit scenarios. Analysis of architectural approaches using analytic models to identify tradeoff points and risks	Architectural approaches or styles; architectural documentation mainly showing Kurcten's 4+1	Multiple QAs
ALMA	Maintenance cost prediction, risk assessment, architectures comparison	Five, the scenario elicitation activity consists of six activities, executed sequentially; no preparation activities	Change Scenarios. Estimates the change of the size of each component	Scenario elicitation based on the goals of the evaluation. Mapping scenarios onto SAs to estimate maintenance cost considering ripple effects.	Architectural documentation, concerning the system's structure comprising components and connectors (like the logical view)	Reusability
CBAM	Provide business measures for particular system changes Make explicit the uncertainty associated with the estimates	Six main steps, quantify the Quality benefits, cost and schedule implications of the architectural strategies	Direct, indirect and exploratory scenarios. Counts the Time and cost utilized by the scenarios	Analyze the benefits of the different architectural strategies ,Assess the quality, and calculate the desirability with respect to cost and time factor	Time and Costs factors involved in analyzing the quality factors and architectural documentation	Costs, Benefits, and Schedule Implications
FAAM	Emphasis on empowering the teams in applying the	Six main steps, these steps must be adapted in	Focusing on interoperable scenarios. The general assessment	Creation of guidelines and templates in generating	It has a well-defined process workbench Description .	Interoperability and Extensibility

https://research.ijcaonline.org/volume49/number16/pxc3881107.pdf

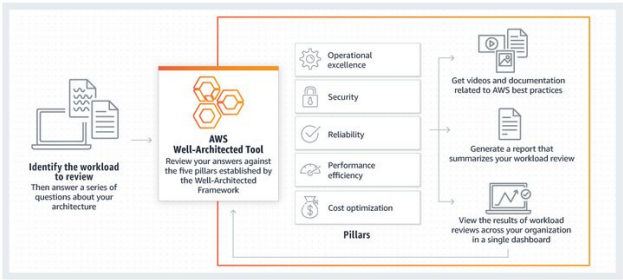
Architectural Evaluation. Reusable Methods

Architecture Trade-offs Analysis Method (ATAM)





AWS Well-Architectured Tool



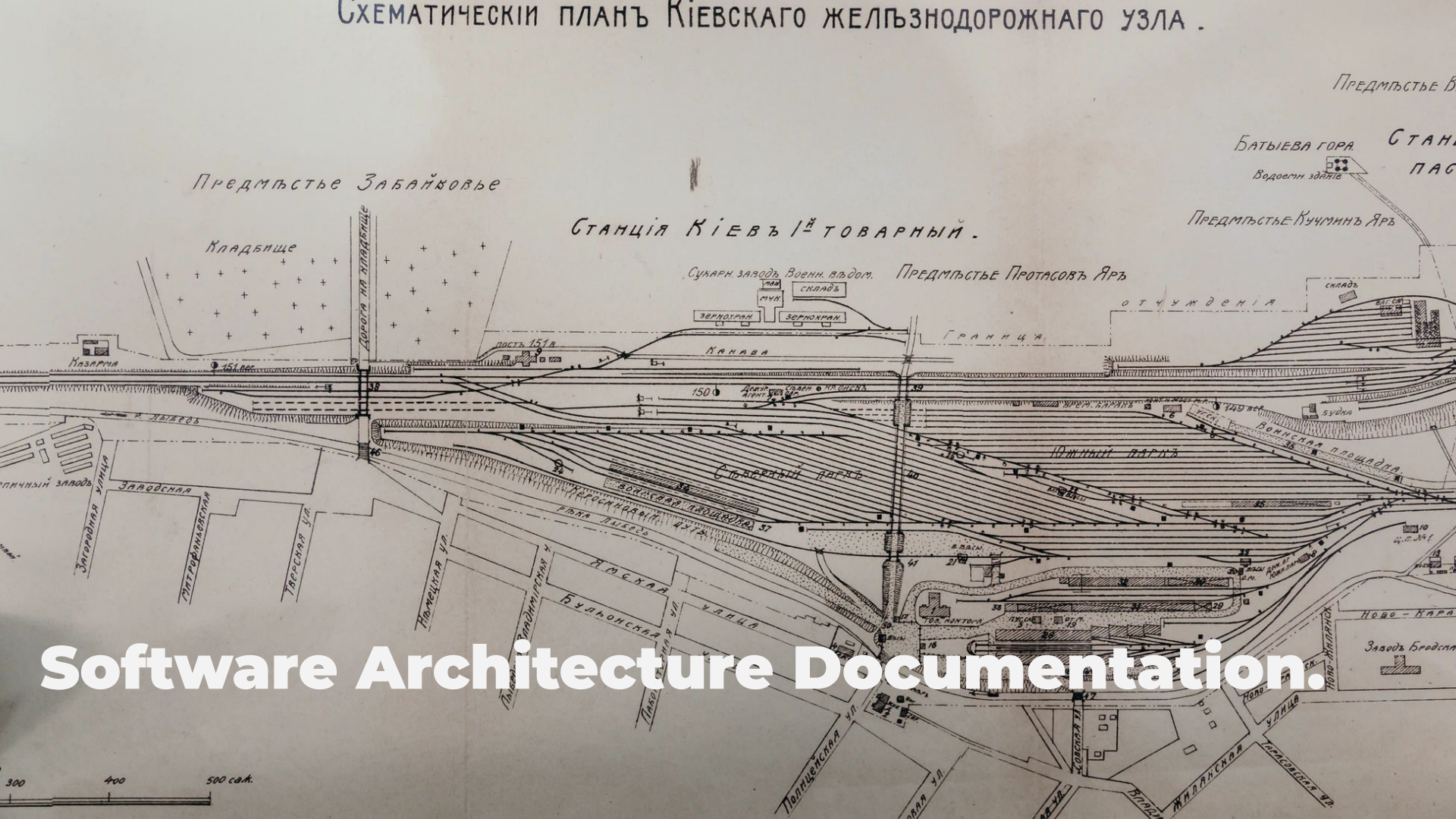
DORA
DEVOPS RESEARCH & ASSESSMENT

Google Cloud

Q&A session

5 minutes


Break.
5 minutes



Software Architecture Documentation.

Agenda.

- Documentation fundamentals;
- UML;
- SEI Views & Beyond;
- Kruchten’s 4+1 View Model;
- C4 model;
- ADR’s;
- Architecture documents templates;
- Architect’s toolset;

Documentation objectives.

- **Education** - introducing people to the system: new members of the team, external analysts, the customer, or even a new architect
- **Communication** - as a vehicle among stakeholders and to/from the architect
- **Analysis** - especially for the quality attributes that the architecture design enables the system to deliver

What can we document?

Problem description & Context

Analysys Results & Decisions

Architecture Views

Architecture Significant Requirements

- Objectives
- Concerns
- Constraints
- Quality Attributes Scenarios

Software

Architecture

Document

Principles / Guidelines / Structure

What should we document?

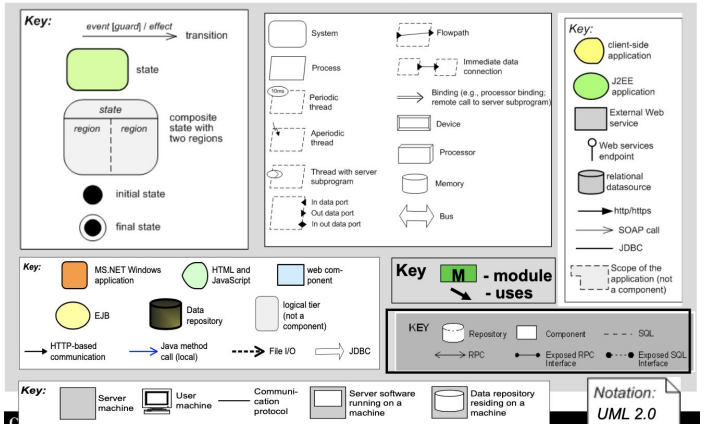
What **should** we document?

1. **Choosing structures** to work with, design them to achieve functional requirements and Quality Attributes using patterns.
2. Producing **Candidate View List**. Start with **stakeholders**. Viewpoint applies to different types of stakeholders -> lot of views
3. **Combining**. Some Views can be merged or stakeholders can use some general Views.
4. **Prioritization**

Common Principles. From SEI

1. Write from the reader's point of view.
2. Avoid unnecessary repetition. Information should stored on one place.
3. Avoid ambiguity. During creation a View specify all information what needed.
4. Use a standard organization.
5. Record your rationale.
6. Keep documentation current, but not too current.
7. Review documentation.

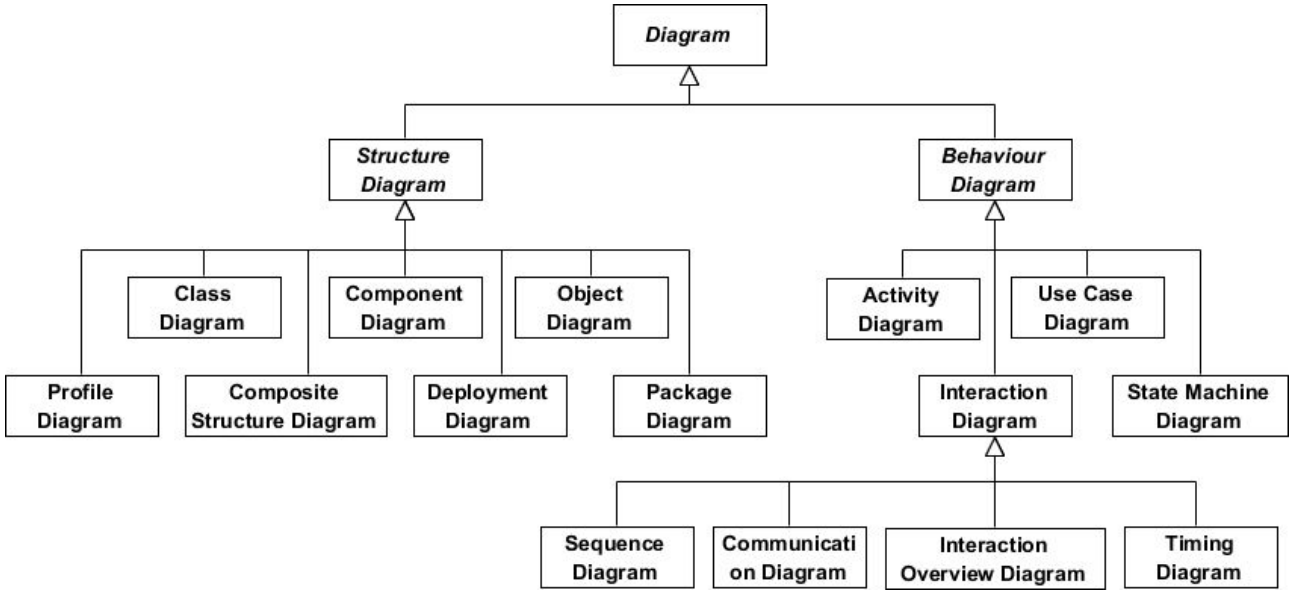
Examples of keys



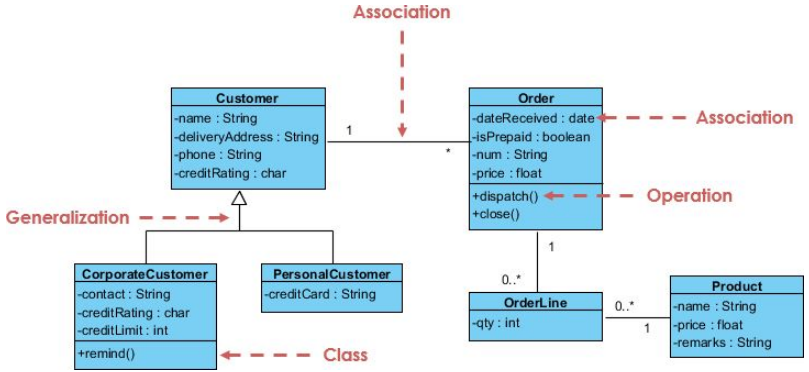
UML.

UML.

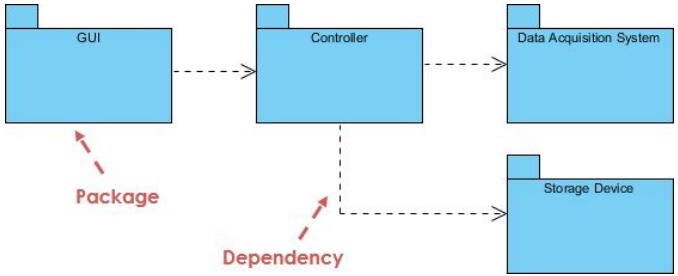
Unified Modeling Language is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.



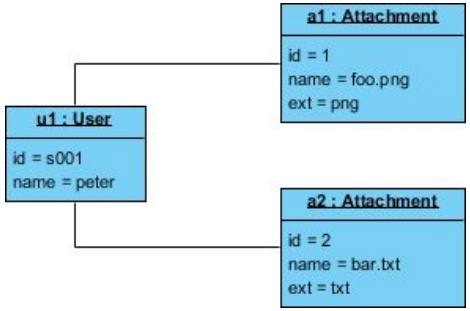
UML. Types of Diagrams



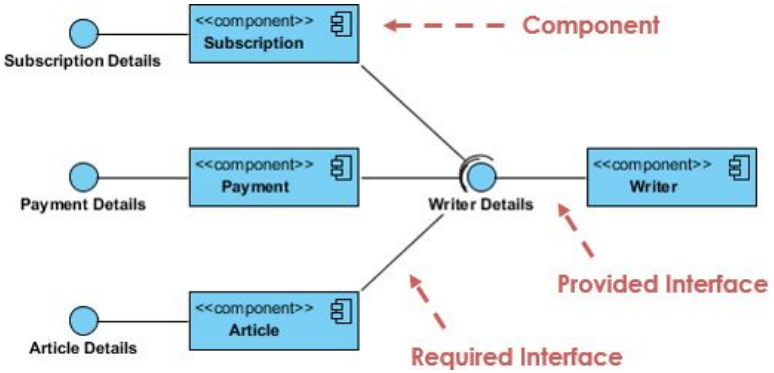
Class



Package

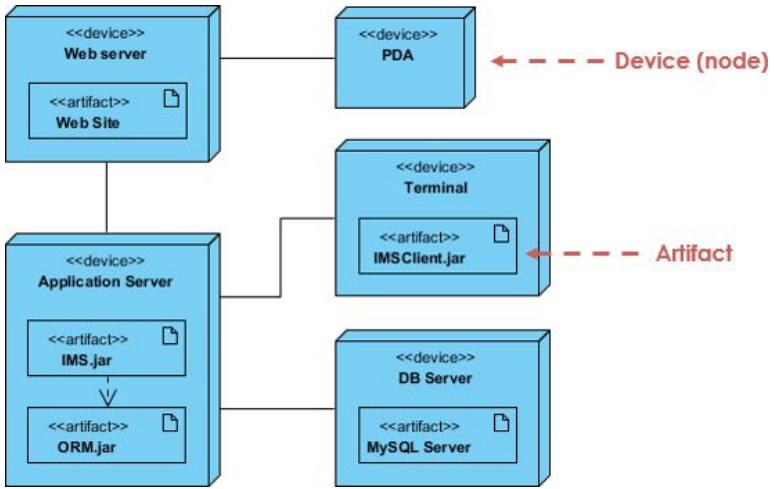


Object

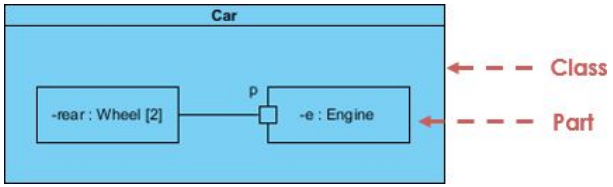


Component

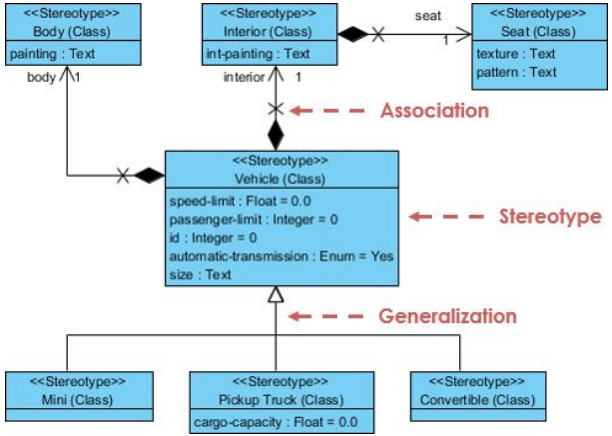
UML. Types of Diagrams



Deployment

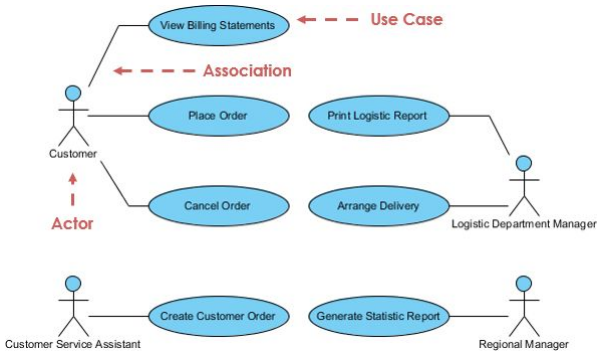


Composite Structure

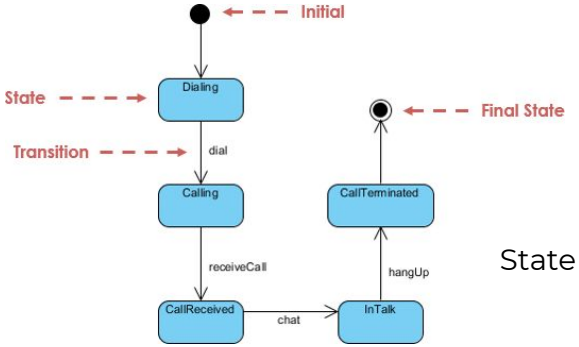


Profile

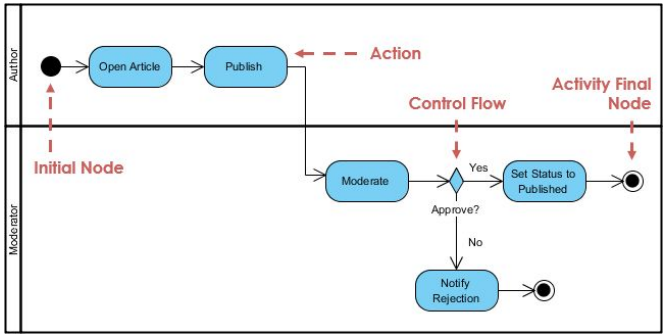
UML. Types of Diagrams



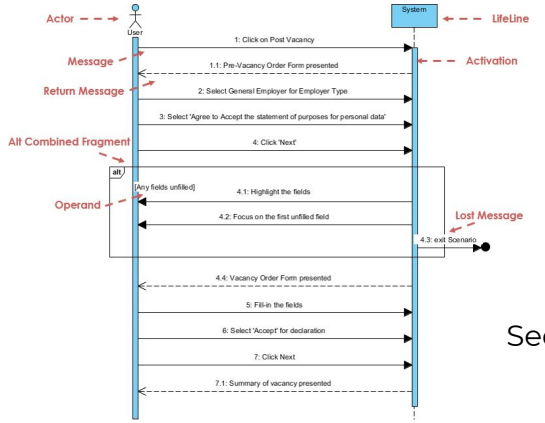
Use-Case



State Machine

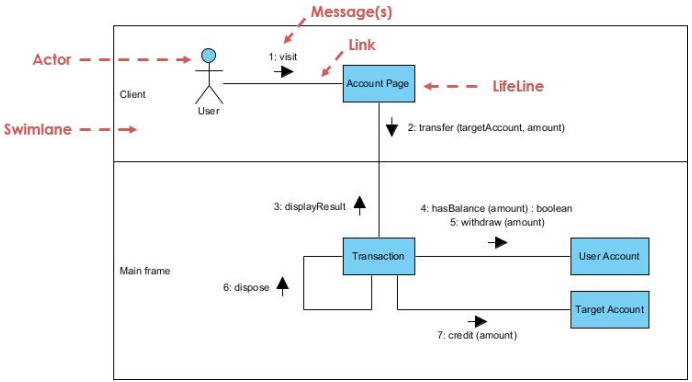


Activity

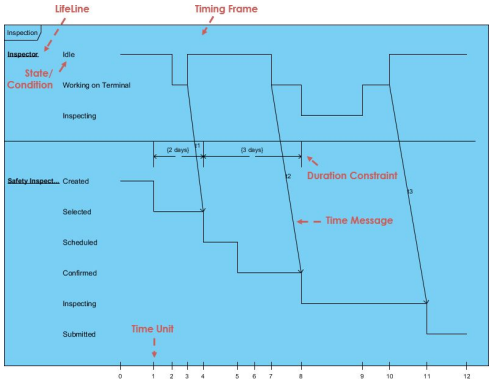


Sequence

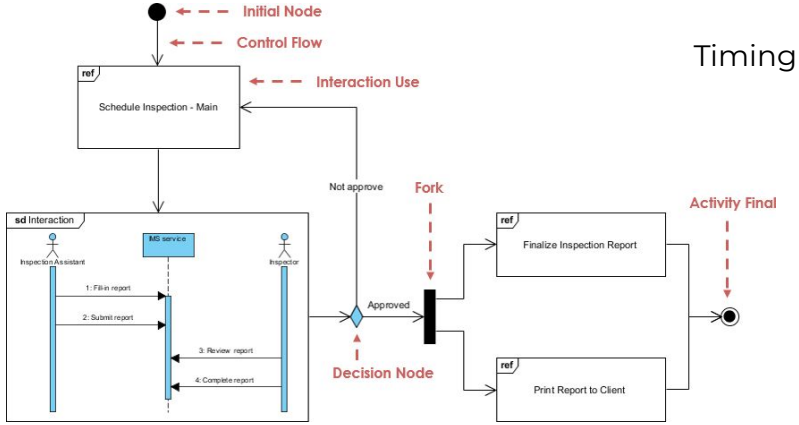
UML. Types of Diagrams



Communication



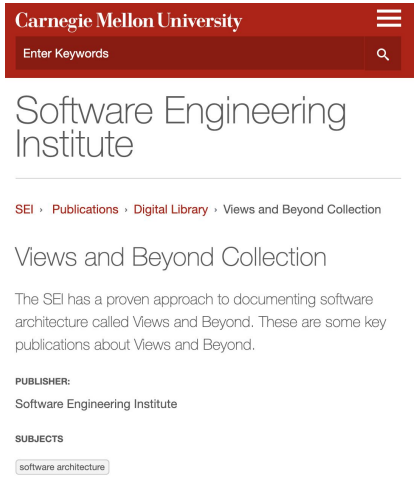
Timing

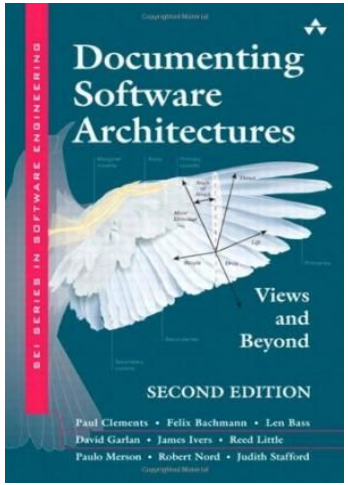


Interaction

SEI Views & Beyond.

SEI Views and Beyond.





https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=484159

- **Module**
- **Component-and-Connector**
- **Allocation**

SEI Views and Beyond. Module Views

- **Elements**: modules.
  - A module is a code unit that implements a set of responsibilities.
- **Relations**:
  - A is part of B.
  - A depends on B.
  - A is a B. (specialization/generalization)
- **Properties**:
  - name
  - responsibilities
  - visibility of the module and its interface.
- **Usage**:
  - Construction:
    - Blueprints for the code.
    - Modules are assigned to teams
    - Subsets and deployment packages are built using module styles.
  - Analysis:
    - Traceability and impact analysis
    - Project management, budgeting, planning, and tracking often use modules.
  - Education:
    - understanding project structure
- **Notations**
  - Informal:
    - box-and-line drawings (Nesting can represent “is part of” relations)
    - tables or lists (indenting can represent “is part of”)
  - Semi-formal:
    - UML **class** diagrams and **package** diagrams

Module styles

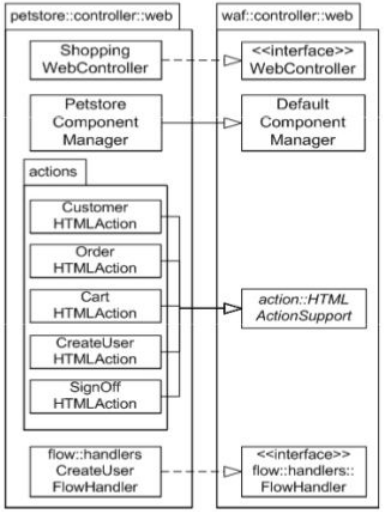
- Decomposition style
- Generalization style
- Uses style
- Layered style
- Data model style
- Aspects style

SEI Views and Beyond. Module Views

Decomposition View

- Elements: modules
- Relations: “is part of.”
- Topology: A child can have only one parent.

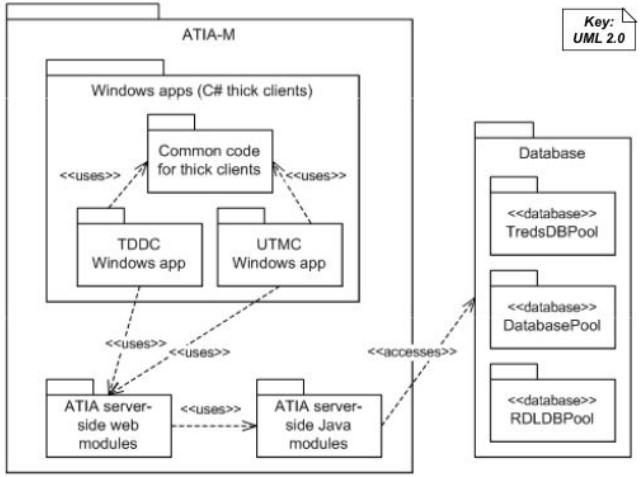
Generalization View



- Elements: modules
- Relations: generalization, an “is a” relation
- Properties: abstract?
- Topology: A module can have multiple parents. Cycles are prohibited.

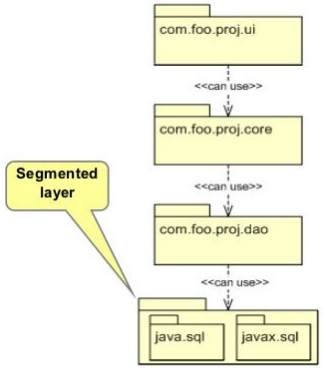
SEI Views and Beyond. Module Views

Uses View



- **Elements:** modules
- **Relations:** "uses," a specialization of "depends on".
- **Topology**: no constraints.

Layered View



- **Elements:** layers, a virtual machine
- **Relations:** "allowed to use," a specialization of the "depends on" relation; The relation is not necessarily transitive
- **Topology:**
  - Every piece of software is assigned to exactly one layer.
  - Software in a layer is allowed to use software in {any lower layer, next lower layer}.
  - Software in a layer {is, is not} allowed to use other software in the same layer.

SEI Views and Beyond. Component-and-Connector (C&C) Views

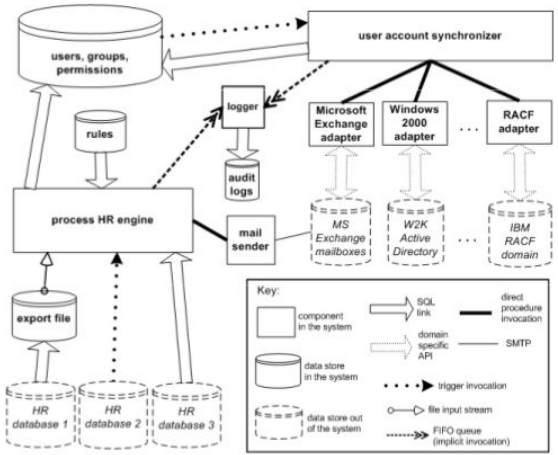
- **Elements**
  - software elements (as defined in module or C&C styles)
  - environment elements
- **Relations**
  - software elements are “allocated to” environment elements
- **Properties**
  - “requires” properties of software elements
  - “provides” properties of the environment elements.
  - If the properties are compatible, then the allocation is a sound one.
- **Notations**
  - Informal
    - box-and-line diagrams
    - tables
    - snapshots of tool interfaces that manage the allocations
  - Semi-formal:
    - UML can show various kinds of allocation, such as software to hardware or software to a file container.

Component-and-connector styles

- Shared-data style
- Pipe-and-filter style
- Service-oriented architecture (SOA) style
- Client-server style
- Publish-subscribe style

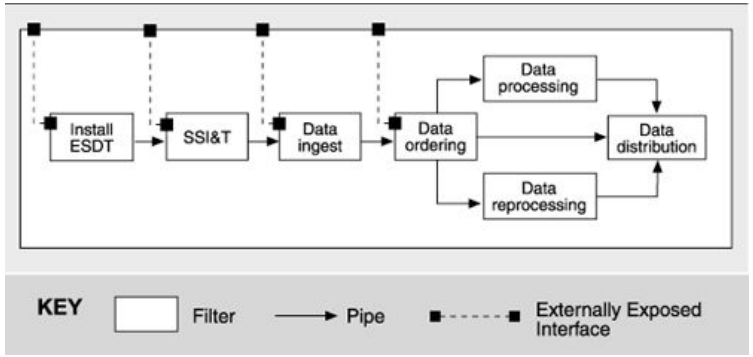
SEI Views and Beyond. Component-and-Connector (C&C) Views

Shared-Data View



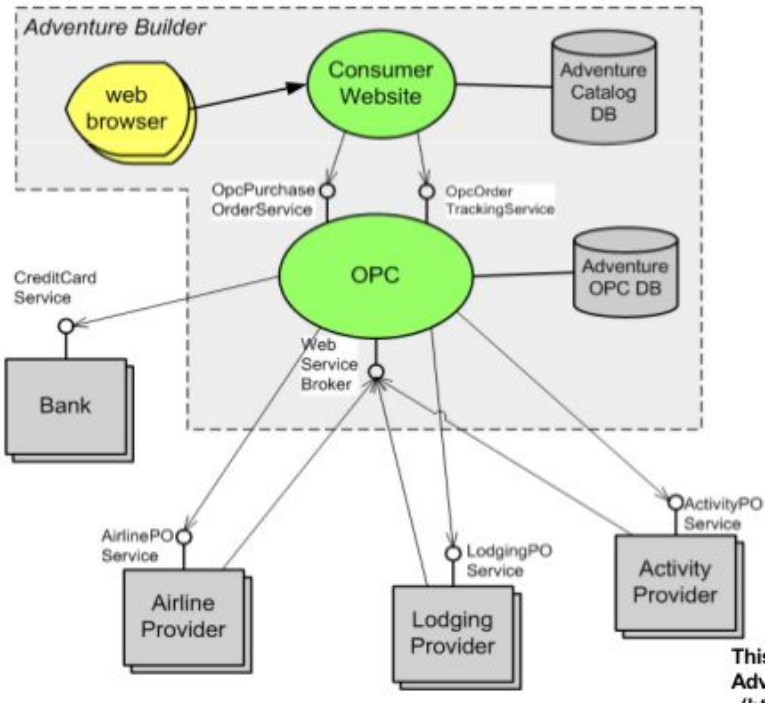
- **Elements** component types: data stores and accessors; connector types: data reading and writing
- **Relations**: attachment
- **Properties**: type of data, data-related performance properties, data distribution
- **Topology**: The data store is attached to the data accessors via connectors.

Pipe-and-Filter View



- **Elements**:
  - component type: filter, which transforms data
  - connector type: pipe, a unidirectional data conduit that preserves the order and value of data
- **Relations**: attachment of pipes to filters
- **Topology**: Pipes connect filters. Further specializations of the style may prohibit loops or branching

SEI Views and Beyond. Component-and-Connector (C&C) Views



Service-Oriented Architecture (SOA) View

- **Elements:**
  - component types: service users, service providers, ESB, registry of services
  - connector types: service calls
- **Relations:** attachment of a service call to a service endpoint
- **Properties**
  - Security (e.g., authorization),
  - cost,
  - reliability,
  - performance
  - other qualities of service can be associated with services.
- **Topology**
  - Service users are connected to service providers.
  - Special components may intermediate the interaction between service users and service providers:
    - Registry of services: naming and location of services
    - ESB: routing, data and format transformation, technology adapters
  - A service user may also be a service provider.

SEI Views and Beyond. Allocation Views

- **Elements**
  - software elements (as defined in module or C&C styles)
  - environment elements
- **Relations**
  - software elements are “allocated to” environment elements
- **Properties**
  - “requires” properties of software elements
  - “provides” properties of the environment elements.
  - If the properties are compatible, then the allocation is a sound one.
- **Notations**
  - Informal
    - box-and-line diagrams
    - tables
    - snapshots of tool interfaces that manage the allocations
  - Semi-formal:
    - UML can show various kinds of allocation, such as software to hardware or software to a file container.

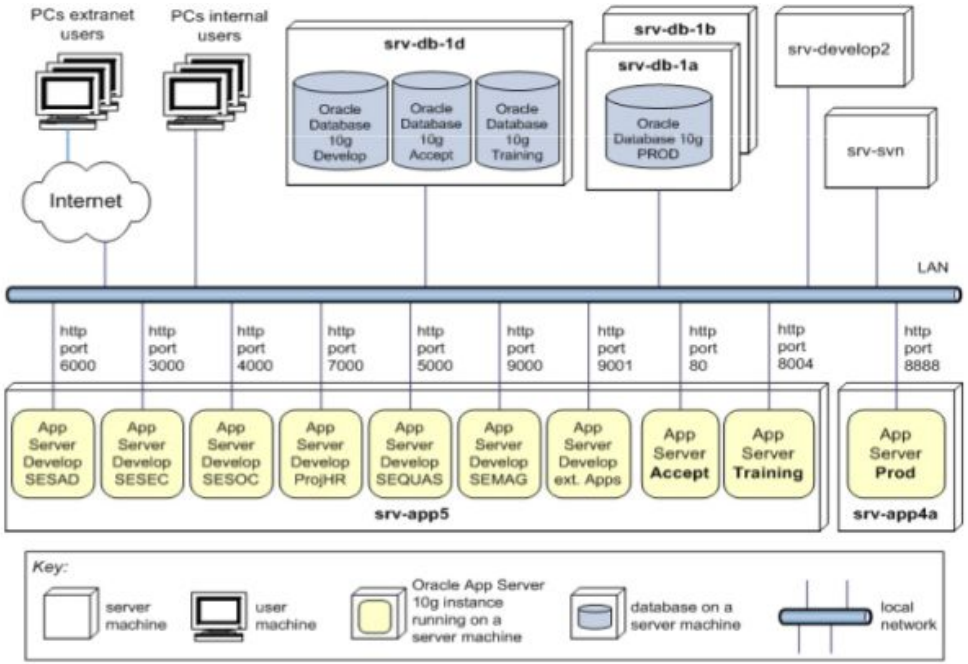
Allocation styles

- Deployment style
- Install style
- Work assignment style

SEI Views and Beyond. Allocation Views

Deployment View

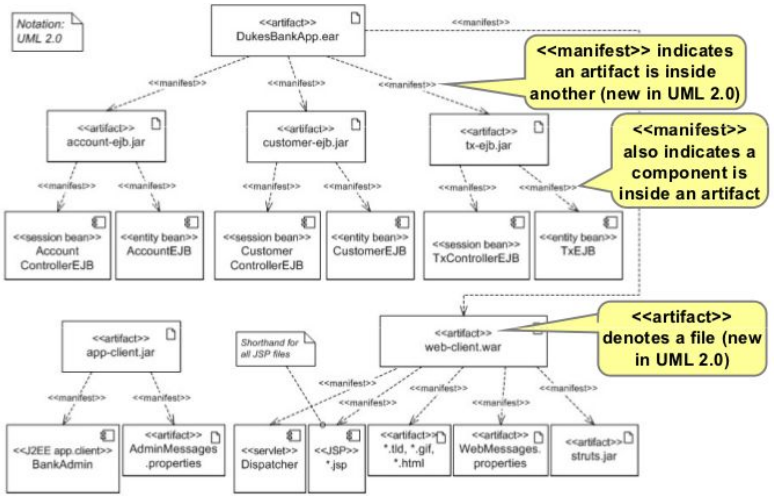
- **Elements**
  - software element—usually processes from C&C views
  - environmental element—computing hardware
- **Relations**
  - "allocated to"—physical elements on which software resides
  - "migrates to," "copy migrates to," and/or "execution migrates to" with dynamic allocation
- **Properties**
  - requires property for software elements
    - significant features required from hardware
  - provides property of environment elements
    - significant features provided by hardware
- **Topology**
  - unrestricted



SEI Views and Beyond. Allocation Views

Implementation View

- **Elements**
  - software element—a module or component
  - environmental element—configuration item; for example, a file or directory
- **Relations**
  - containment—One configuration item is contained in another.
  - “allocated to”—allocation of a module to a configuration item
- **Properties**
  - requires property for software element
    - usually requirements on the development or production environments; for example, Java
  - provides property of environment elements
    - characteristics provided by development or production environments
- **Topology**
  - configuration items are hierarchical, following “is contained in”

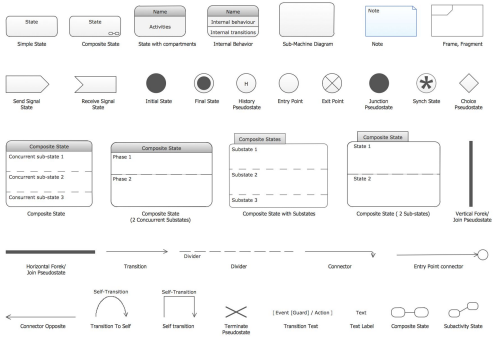


SEI Views and Beyond. Notations for Architecture Views

- **Informal Notation**
  - Views are depicted (often graphically) using general-purpose diagramming and editing tools
  - The semantics of the description are characterized in natural language
  - They cannot be formally analyzed

- **Semi-Formal**
  - Standardized notation that prescribes graphical elements and rules of construction
  - Lacks a complete semantic treatment of the meaning of those elements
  - Rudimentary analysis can be applied
  - UML is a semi-formal notation in this sense.

- **Formal**
  - Views are described in a notation that has a precise (usually mathematically based) semantics.
  - Formal analysis of both syntax and semantics is possible.
  - Architecture description languages (ADLs)
  - Support automation through associated tools.



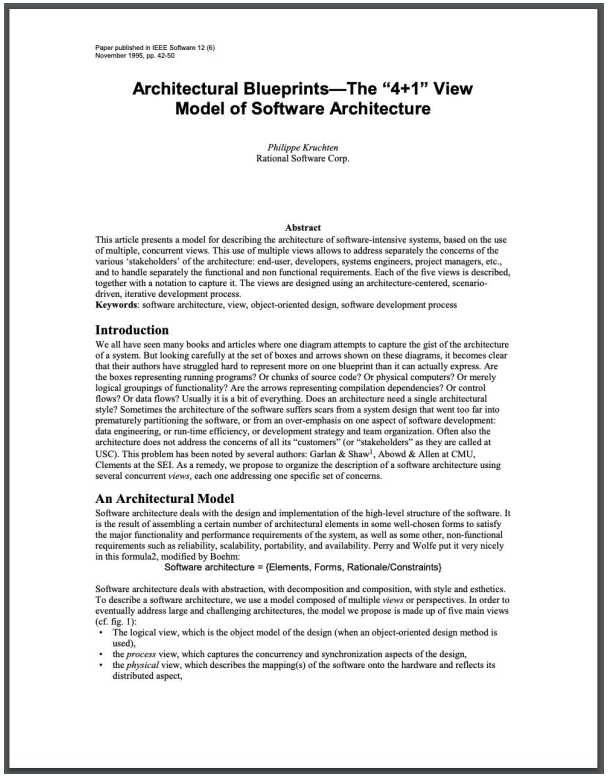
SEI Views and Beyond.

Stakeholder	Module				C&C	Allocation	
	Decomposition	Uses	Class	Layer	*Various*	Deployment	Implem.
Project Manager	s	s		s		d	
Member of Development Team	d	d	d	d	d	s	s
Testers and Integrators		d	d		s	s	s
Maintainers	d	d	d	d	d	s	s
Product Line Application Builder		d	s	o	s	s	s
Customer					s	o	
End User					s	s	
Analyst	d	d	s	d	s	d	
Infrastructure Support	s	s		s		s	d
New Stakeholder	x	x	x	x	x	x	x
Current and Future Architect	d	d	d	d	d	d	s

d = detailed information, s = some details, o = overview information, x = anything

Kruchten’s 4+1 View Model.

4+1. Use Case Diagrams



Paper published in IEEE Software 12 (6)
November 1995, pp. 42-50

Architectural Blueprints—The "4+1" View Model of Software Architecture

Philippe Kruchten
Rational Software Corp.

Abstract

This article presents a model for describing the architecture of software-intensive systems, based on the use of multiple, concurrent views. This use of multiple views allows to address separately the concerns of the various 'stakeholders' of the architecture: end-user, developers, systems engineers, project managers, etc., and to handle separately the functional and non functional requirements. Each of the five views is described, together with a notation to capture it. The views are designed using an architecture-centered, scenario-driven, iterative development process.
Keywords: software architecture, view, object-oriented design, software development process

Introduction

We all have seen many books and articles where one diagram attempts to capture the gist of the architecture of a system. But looking carefully at the set of boxes and arrows shown on these diagrams, it becomes clear that their authors have struggled hard to represent more on one blueprint than it can actually express. Are the boxes representing running programs? Or chunks of source code? Or physical computers? Or merely logical groupings of functionality? Are the arrows representing compilation dependencies? Or control flows? Or data flows? Usually it is a bit of everything. Does an architecture need a single architectural style? Sometimes the architecture of the software suffers scars from a system design that went too far into prematurely partitioning the software, or from an over-emphasis on one aspect of software development: data engineering, or run-time efficiency, or development strategy and team organization. Often also the architecture does not address the concerns of all its "customers" (or "stakeholders" as they are called at USC). This problem has been noted by several authors: Garlan & Shaw[1], Abowd & Allen at CMU, Clements at the SEI. As a remedy, we propose to organize the description of a software architecture using several concurrent *views*, each one addressing one specific set of concerns.

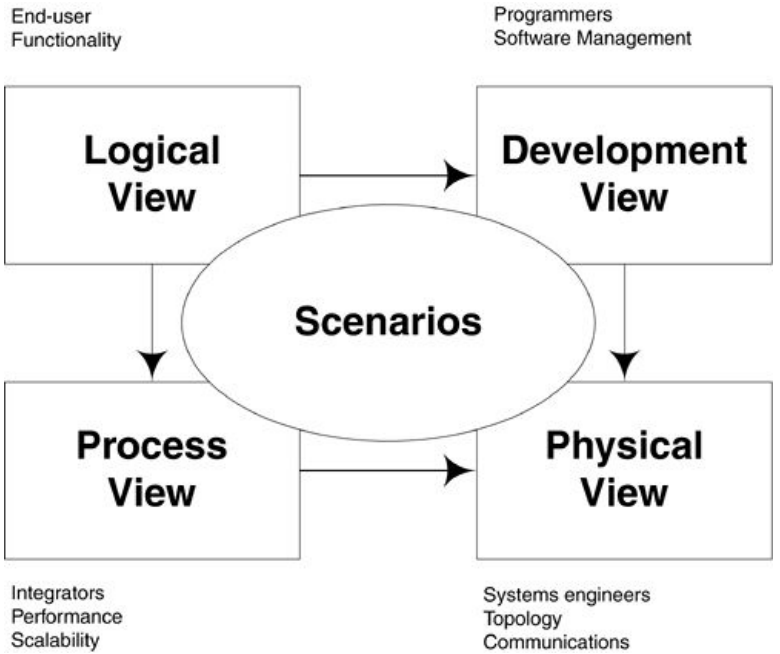
An Architectural Model

Software architecture deals with the design and implementation of the high-level structure of the software. It is the result of assembling a certain number of architectural elements in some well-chosen forms to satisfy the major functionality and performance requirements of the system, as well as some other, non-functional requirements such as reliability, scalability, portability, and availability. Perry and Wolfe put it very nicely in this formula2, modified by Boehm:

Software architecture = {Elements, Forms, Rationale/Constraints}

Software architecture deals with abstraction, with decomposition and composition, with style and esthetics. To describe a software architecture, we use a model composed of multiple *views* or perspectives. In order to eventually address large and challenging architectures, the model we propose is made up of five main views (cf. fig. 1):

- The logical view, which is the object model of the design (when an object-oriented design method is used),
- the *process* view, which captures the concurrency and synchronization aspects of the design,
- the *physical* view, which describes the mapping(s) of the software onto the hardware and reflects its distributed aspect,



https://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf

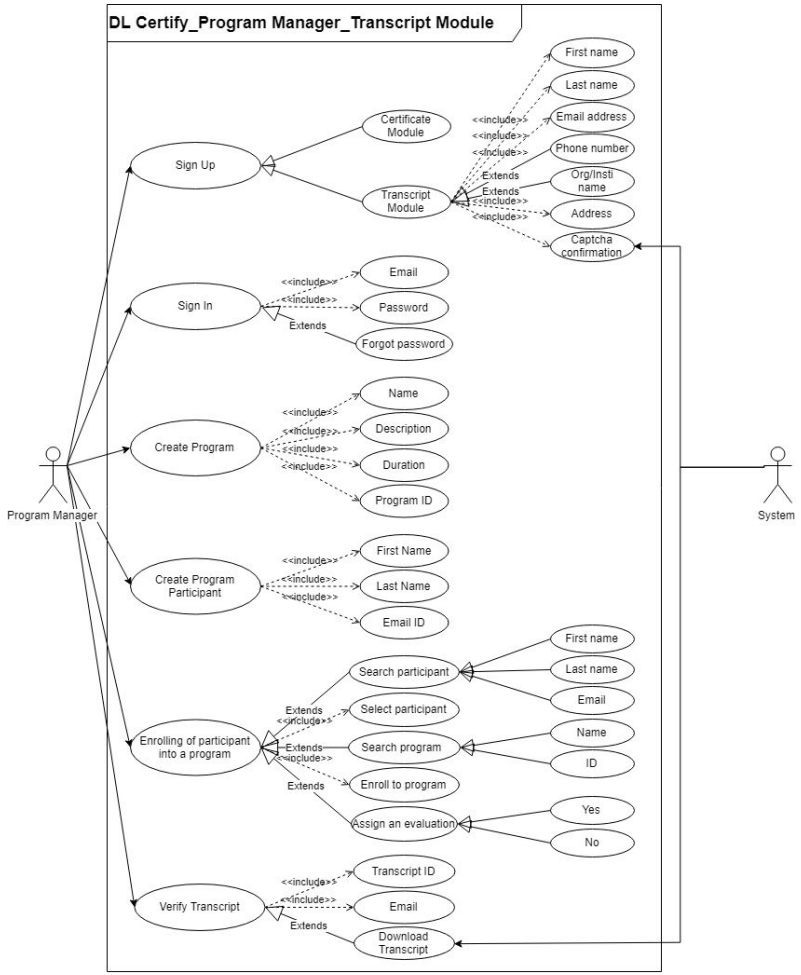
4+1. Use Case Diagrams

The following **4 elements** in the use case diagrams.

1. The Actor (Primary actor and Secondary actor) — The user (living or non-living) who interacts with the system.
2. The System Boundary
3. The Use Case (Essential use case and Supporting use case) — The services that the system performs.
4. The lines represent the relationship, between the elements.

How to draw the Use Case Diagram (UCD)

- Identify the actions against the requirements gathered
- Identify the actors (primary and secondary actors)
- Identify which actor performs the above action
- Identify the essential and supporting use cases
- Draw the relationship between the identified actors and action



4+1. Use Case Diagrams

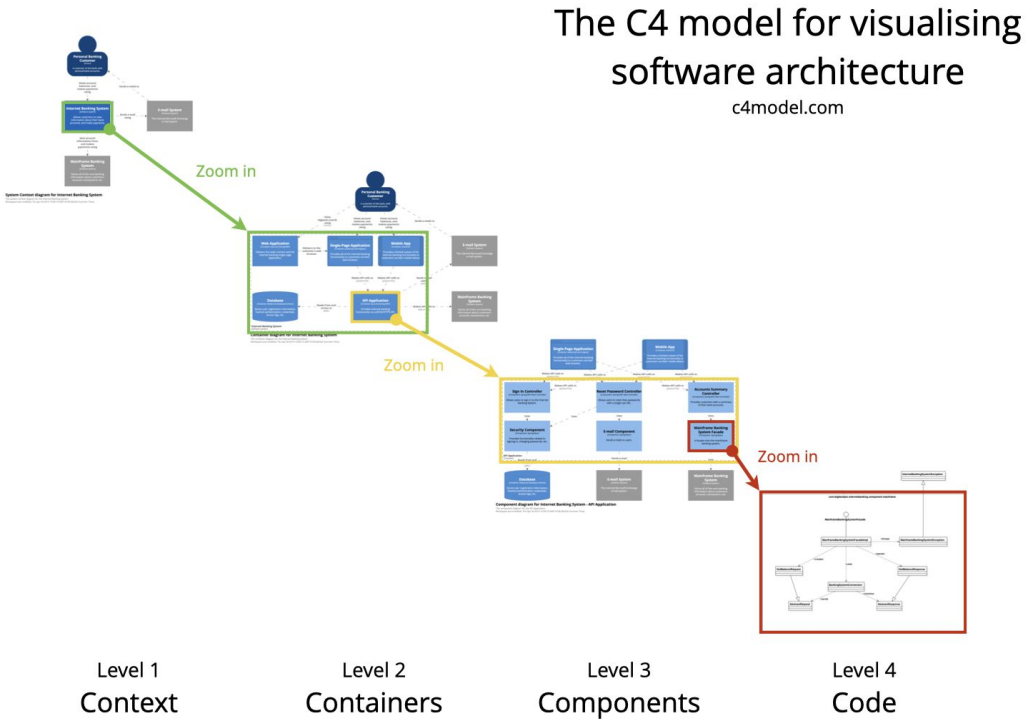
	Logical	Process	Development	Physical	Scenario
Description	Shows the component (Object) of system as well as their interaction	Shows the processes / Workflow rules of system and how those processes communicate, focuses on dynamic view of system	Gives building block views of system and describe static organization of the system modules	Shows the installation, configuration and deployment of software application	Shows the design is complete by performing validation and illustration
Viewer / Stake holder	End-User, Analysts and Designer	Integrators & developers	Programmer and software project managers	System engineer, operators, system administrators and system installers	All the views of their views and evaluators
Consider	Functional requirements	Non Functional Requirements	Software Module organization (Software management reuse, constraint of tools)	Nonfunctional requirement regarding to underlying hardware	System Consistency and validity
UML – Diagram	Class, State, Object, sequence, Communication Diagram	Activity Diagram	Component, Package diagram	Deployment diagram	Use case diagram

C4 Model.

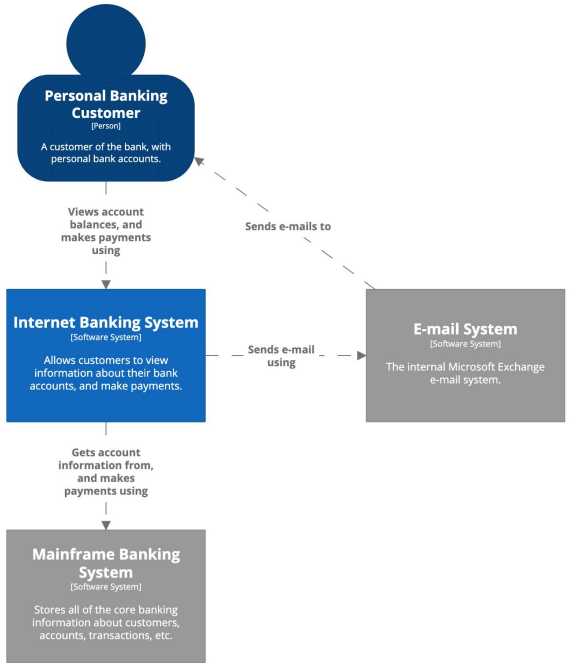
C4 Model.

C4 model is a lean graphical notation technique for modelling the architecture of software systems.

Different model parts focuses on different kinds of stakeholders.



C4 Model. Context



System Context diagram for Internet Banking System

The system context diagram for the Internet Banking System.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

Level 1: System Context diagram

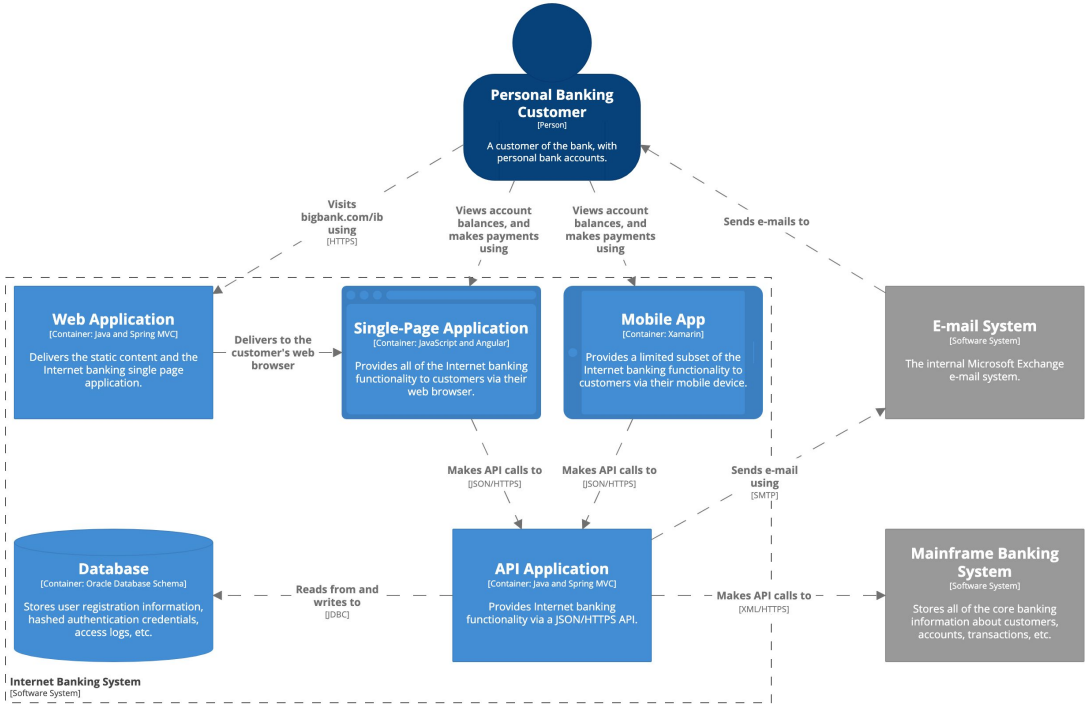
Scope: A single software system.

Primary elements: The software system in scope.

Supporting elements: People (e.g. users, actors, roles, or personas) and software systems (external dependencies) that are directly connected to the software system in scope. Typically these other software systems sit outside the scope or boundary of your own software system, and you don't have responsibility or ownership of them.

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

C4 Model. Container



Container diagram for Internet Banking System

The container diagram for the Internet Banking System.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

Level 2: Container diagram

Scope: A single software system.

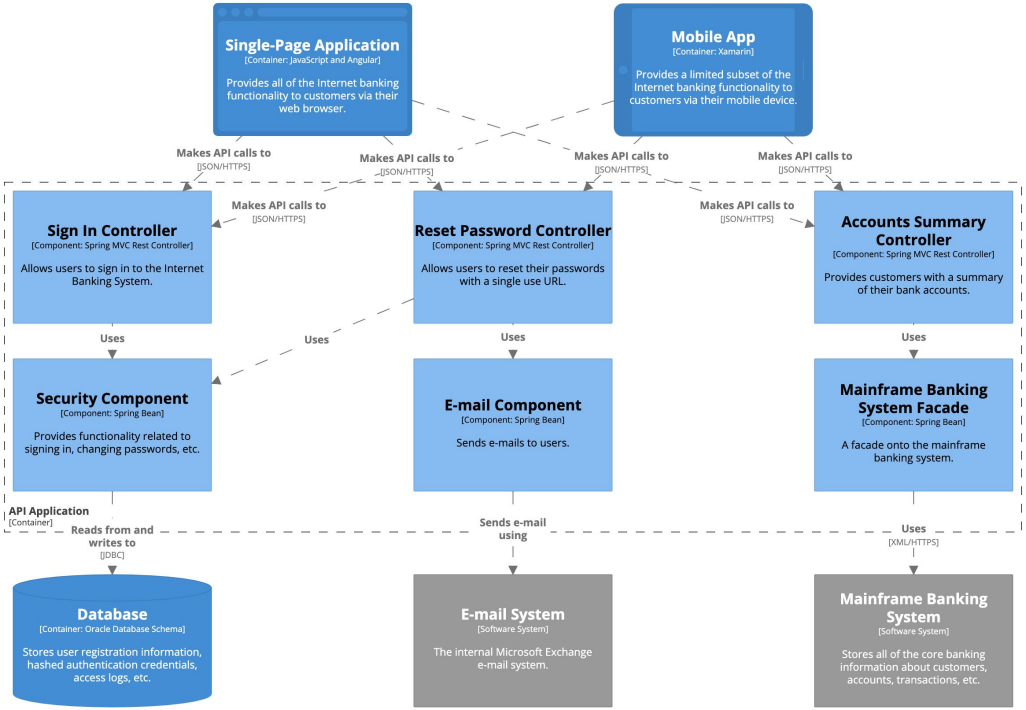
Primary elements: Containers within the software system in scope.

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.

C4 Model. Components



Component diagram for Internet Banking System - API Application

The component diagram for the API Application.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

Level 3: Component diagram

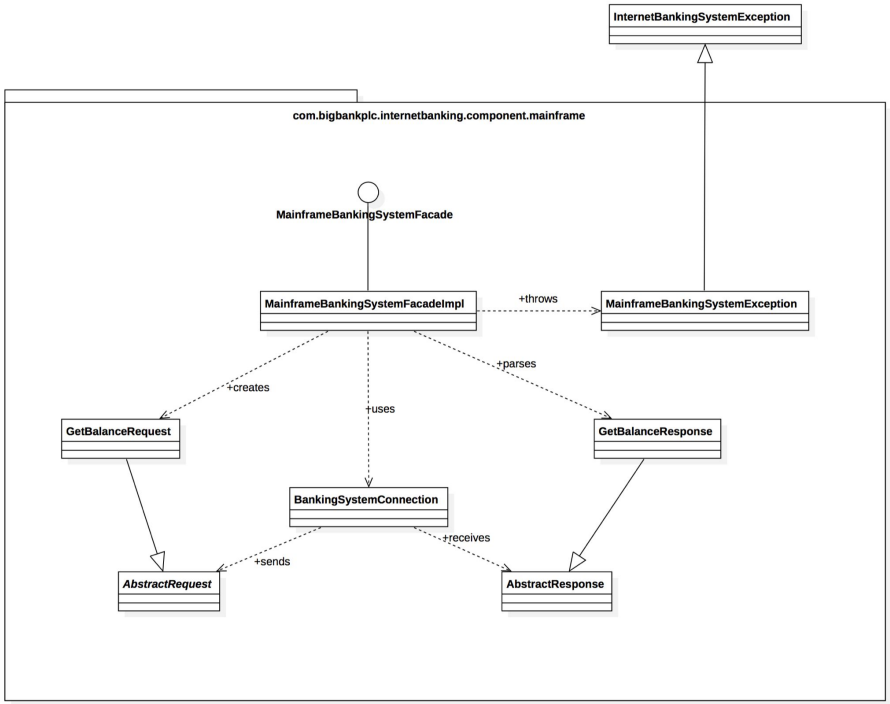
Scope: A single container.

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.

C4 Model. Code



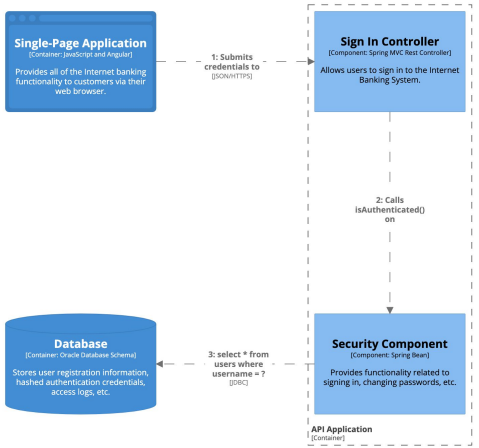
Level 4: Code

Scope: A single component.

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

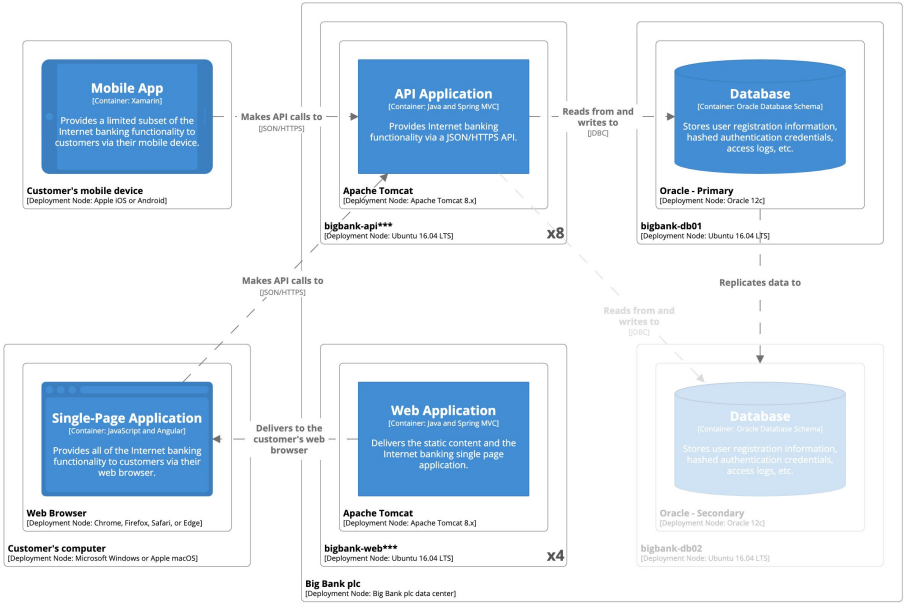
Intended audience: Software architects and developers.

C4 Model. Examples



Dynamic diagram for API Application

Summarises how the sign in feature works in the single-page application.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)



Deployment diagram for Internet Banking System - Live

An example live deployment scenario for the Internet Banking System.
Workspace last modified: Wed Feb 05 2020 09:33:36 GMT+0100 (Central European Standard Time)

C4 Model. Tools

Modelling tools (recommended - why?)

Structurizr

Structurizr is specifically designed to support the C4 model, and allows you to create diagrams as code (Java, .NET, TypeScript, PHP, Python, Go), diagrams as text (text-based DSL, YAML), or by using a web-based UI.

Ciaran Treanor has built a Structurizr DSL syntax highlighting extension for Visual Studio Code, while the open source Structurizr CLI can also output diagrams in PlantUML, Mermaid, and WebSequenceDiagrams formats.

Text-based · GUI-based · Cloud · On-premises · $ Freemium

Archi

Archi provides a way for you to create C4 model diagrams with ArchiMate. See C4 Model, Architecture Viewpoint and Archi 4.7 for more details..

GUI-based · On-premises · $ Free

Sparx Enterprise Architect

LieberLieber Software has built an extension for the C4 model, based upon the MDG Technology built into Sparx Enterprise Architect.

GUI-based · On-premises · $ Paid

MooD

MooD has support for the C4 model via a set of blueprints.

GUI-based · Cloud · On-premises · $ Paid

Diagramming tools

PlantUML

There are a number of extensions for PlantUML to asisst in the creation of C4 model diagrams:

C4-PlantUML by Ricardo Niepel
C4-PlantumlSkin by Savvas Kleanthous
c4builder by Victor Lupu
plantuml-libs by Thibault Morin

Text-based · Cloud · On-premises · $ Free

diagrams.net

diagrams.net includes support for the C4 model, and there are also a number of plugins that allow you to create diagrams using pre-built shapes:

c4-draw.io by Chris Kaminski
c4-draw.io by Tobias Hochgürtel
EasyC4 by Maciek Sliwinski

GUI-based · Cloud · On-premises · $ Free

OmniGraffle

Dennis Laumen has created a C4 model stencil for OmniGraffle, that allows you to create diagrams using pre-built shapes.

GUI-based · On-premises · $ Paid

Microsoft Visio

"pihalve" has created a C4 model template for Microsoft Visio, that allows you to create diagrams using pre-built shapes.

GUI-based · Cloud · On-premises · $ Paid

Architecture Decision Record.

Architecture Decision Record.

An **architecture decision record** (**ADR**) is a document that captures an important architectural decision made along with its context and consequences.

An **architecture decision** (**AD**) is a software design choice that addresses a significant requirement.

An **architecture decision log** (**ADL**) is the collection of all ADRs created and maintained for a particular project (or organization).

An **architecturally-significant requirement** (**ASR**) is a requirement that has a measurable effect on a software system's architecture.

All these are within the topic of **architecture knowledge management** (**AKM**).

Architecture Decision Record.

LICENSE.txt	add license
adr-001-use-adrs.md	Fix typo it's => its
adr-002-configuration.md	Fix YouTube link
adr-003-config-implementation.md	Fix ADR-003 typos
adr-004-module-loading.md	fix typos
adr-005-user-facing-config.md	fix typos
adr-006-core-runtime.md	Update adr-006-core-runtime.md
adr-007-configuration-updates.md	fix typos
adr-008-abstract-modules.md	fix typos
adr-009-datomic-config-ontology....	fix typos
adr-010-persistent-configs.md	adr-010-persistent-configs.md
adr-011-asset-pipeline.md	add asset pipeline ADR
adr-012-additional-validation.md	add validation ADR
adr-013-error-reporting.md	Merge pull request #16 from christianromney/patch-1
adr-014-project-templates.md	compare & constrast ADR 14 with Rails templates

ADR template by Michael Nygard

In each ADR file, write these sections:

Title

Status

What is the status, such as proposed, accepted, rejected, deprecated, superseded, etc.?

Context

What is the issue that we're seeing that is motivating this decision or change?

Decision

What is the change that we're proposing and/or doing?

Consequences

What becomes easier or more difficult to do because of this change?

https://github.com/arachne-framework/architecture/blob/master/adr-001-use-adrs.md

Architecture Document Templates.

Architecture Document Templates. SEI V&B

SEI "Views and Beyond" Architecture Documentation Template 05 February 2006

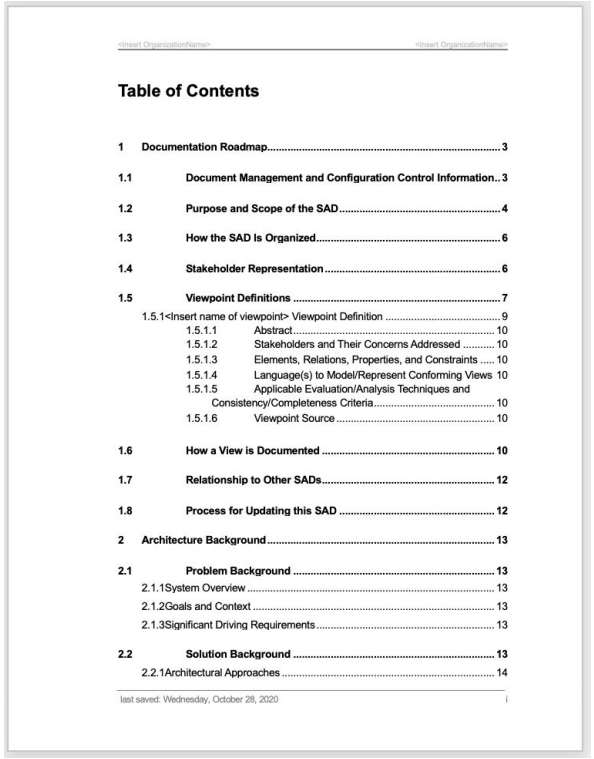
<Insert OrganizationName>

<Insert Project Name>
Software Architecture Document (SAD)

CONTENT OWNER: <Insert Name>

DOCUMENT NUMBER:	RELEASE/REVISION:	RELEASE/REVISION DATE:
•	•	•
•	•	•
•	•	•
•	•	•
•	•	•
•	•	•

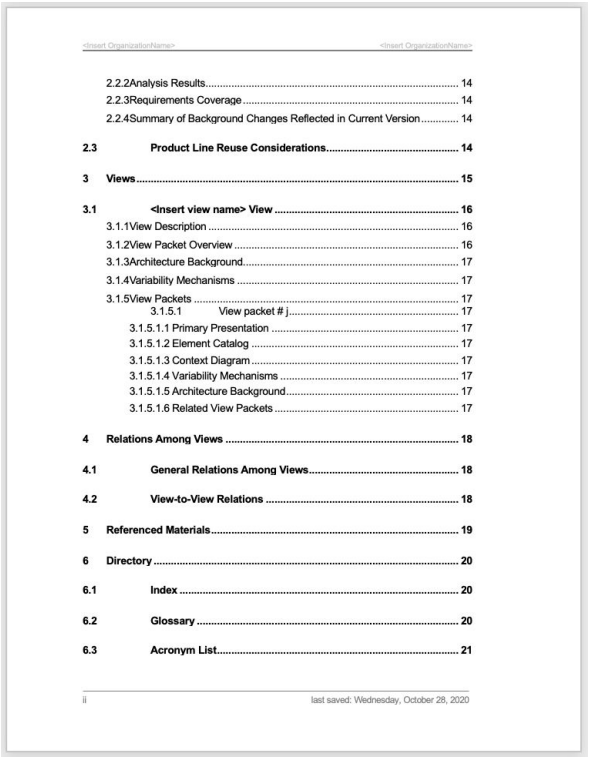
All future revisions to this document shall be approved by the content owner prior to release.



<Insert OrganizationName> <Insert OrganizationName>

Table of Contents

last saved: Wednesday, October 28, 2020 i



<Insert OrganizationName> <Insert OrganizationName>

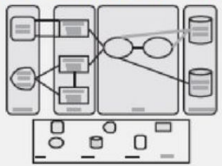
ii last saved: Wednesday, October 28, 2020

https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=519381

Architecture Document Templates. SEI V&B

Template for a View

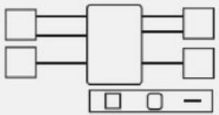
Section 1. Primary Presentation



Section 2. Element Catalog

Section 2.A. Elements and Their Properties
Section 2.B. Relations and Their Properties
Section 2.C. Element Interfaces
Section 2.D. Element Behavior

* **Section 3. Context Diagram**



* **Section 4. Variability Guide**

Section 5. Rationale

* Optional

Template for Interface Documentation

1. Interface identity (especially important for elements that have more than one interface)

2. Resources provided. For each resource:

- **syntax** of the resource: its signature and the information needed to use it in a syntactically correct manner
- **semantics** of the resource: what is the observable effect of using the resource?
  - assignment of values to parameters and return values
  - change in element state or other resources
  - events signaled or messages sent
  - events observable in the system's environment
  - whether the resource is atomic, interruptible, synchronous or asynchronous
  - usage restrictions on the resource
- **error handling:** error conditions and exceptions that can be raised by the resource

3. Data types and constants

- Describe any new data types and constants defined for this interface.

4. Error handling: error-handling behavior that is common to all or many resources.

5. Variability provided by the element's interface (if any) and how to exercise it

6. Quality attribute characteristics of the resources

- alternative: Specify this information with each resource.

7. Rationale and design issues for the interface

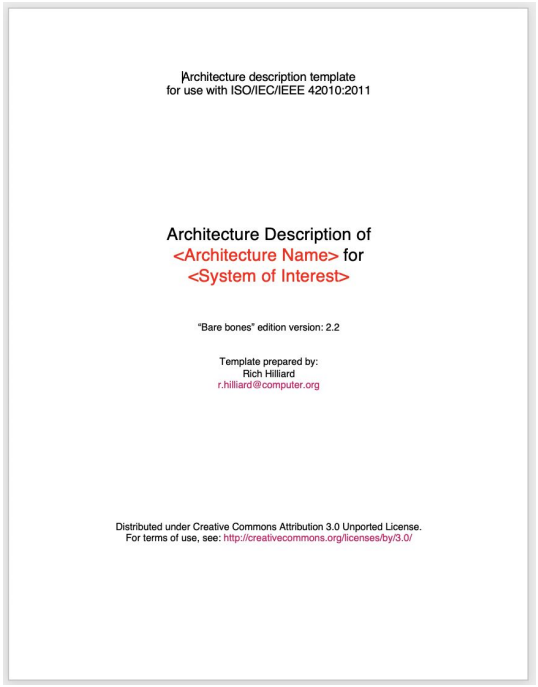
- the reasons behind the design decisions that led to the current interface
- rejected alternatives

8. Usage guide

- scenarios for using the resources
- examples of using the resources

https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=519381

Architecture Document Templates. ISO/IEC/IEEE 42010

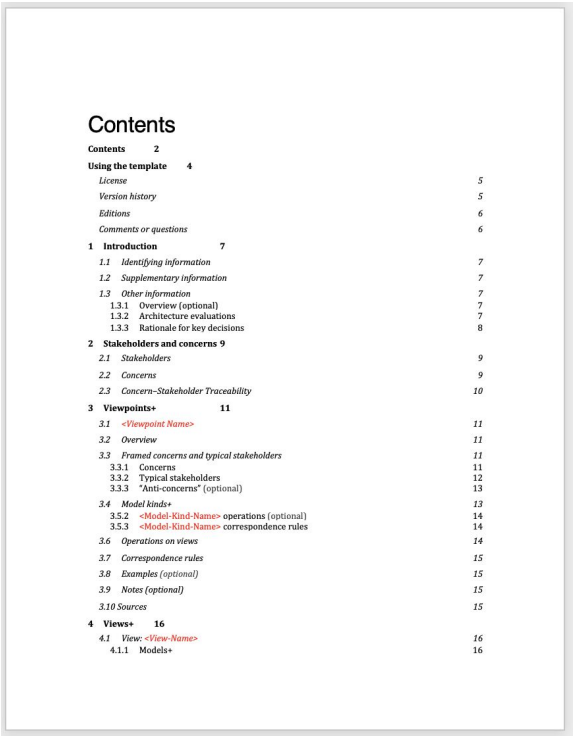


Architecture description template
for use with ISO/IEC/IEEE 42010:2011

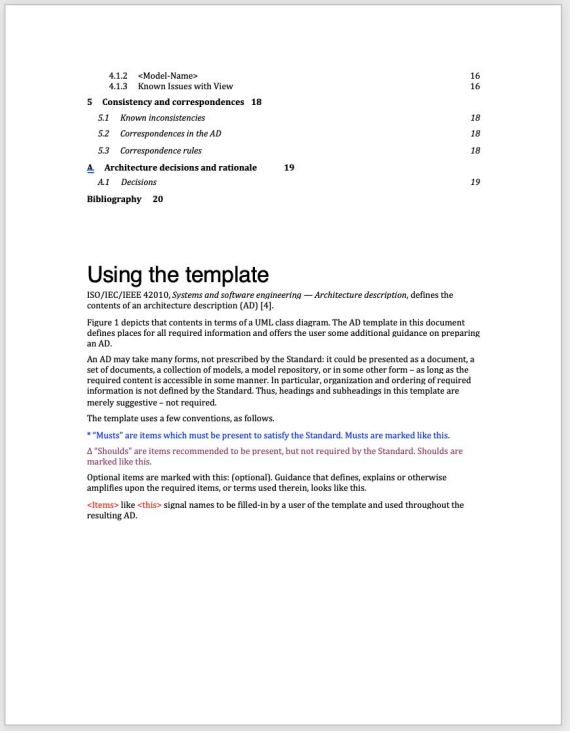
Architecture Description of
<Architecture Name> for
<System of Interest>

"Bare bones" edition version: 2.2

Template prepared by:
Rich Hilliard
r.hilliard@computer.org



Contents



Using the template

ISO/IEC/IEEE 42010, *Systems and software engineering — Architecture description*, defines the contents of an architecture description (AD) [4].

Figure 1 depicts that contents in terms of a UML class diagram. The AD template in this document defines places for all required information and offers the user some additional guidance on preparing an AD.

An AD may take many forms, not prescribed by the Standard: it could be presented as a document, a set of documents, a collection of models, a model repository, or in some other form – as long as the required content is accessible in some manner. In particular, organization and ordering of required information is not defined by the Standard. Thus, headings and subheadings in this template are merely suggestive – not required.

The template uses a few conventions, as follows.

* "Musts" are items which must be present to satisfy the Standard. Musts are marked like this.

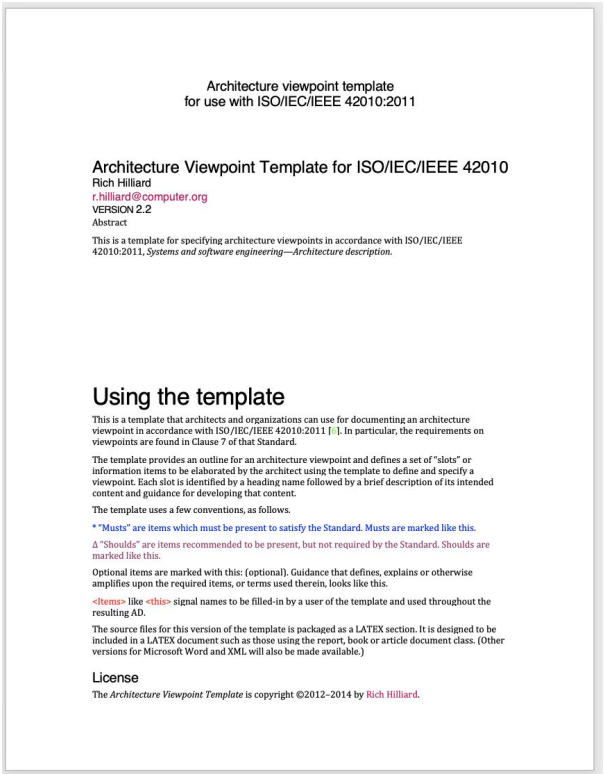
Δ "Shoulds" are items recommended to be present, but not required by the Standard. Shoulds are marked like this.

Optional items are marked with this: (optional). Guidance that defines, explains or otherwise amplifies upon the required items, or terms used therein, looks like this.

<Items> like <this> signal names to be filled-in by a user of the template and used throughout the resulting AD.

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

Architecture Document Templates. ISO/IEC/IEEE 42010



Architecture viewpoint template
for use with ISO/IEC/IEEE 42010:2011

Architecture Viewpoint Template for ISO/IEC/IEEE 42010

Rich Hilliard
r.hilliard@computer.org

VERSION 2.2

Abstract

This is a template for specifying architecture viewpoints in accordance with ISO/IEC/IEEE 42010:2011, *Systems and software engineering—Architecture description.*

Using the template

This is a template that architects and organizations can use for documenting an architecture viewpoint in accordance with ISO/IEC/IEEE 42010:2011 [#]. In particular, the requirements on viewpoints are found in Clause 7 of that Standard.

The template provides an outline for an architecture viewpoint and defines a set of "slots" or information items to be elaborated by the architect using the template to define and specify a viewpoint. Each slot is identified by a heading name followed by a brief description of its intended content and guidance for developing that content.

The template uses a few conventions, as follows.

* "Musts" are items which must be present to satisfy the Standard. Musts are marked like this.

Δ "Shoulds" are items recommended to be present, but not required by the Standard. Shoulds are marked like this.

Optional items are marked with this: (optional). Guidance that defines, explains or otherwise amplifies upon the required items, or terms used therein, looks like this.

<Items> like <this> signal names to be filled-in by a user of the template and used throughout the resulting AD.

The source files for this version of the template is packaged as a LATEX section. It is designed to be included in a LATEX document such as those using the report, book or article document class. (Other versions for Microsoft Word and XML will also be made available.)

License

Using the template
- License
- Version history
- Comments or questions

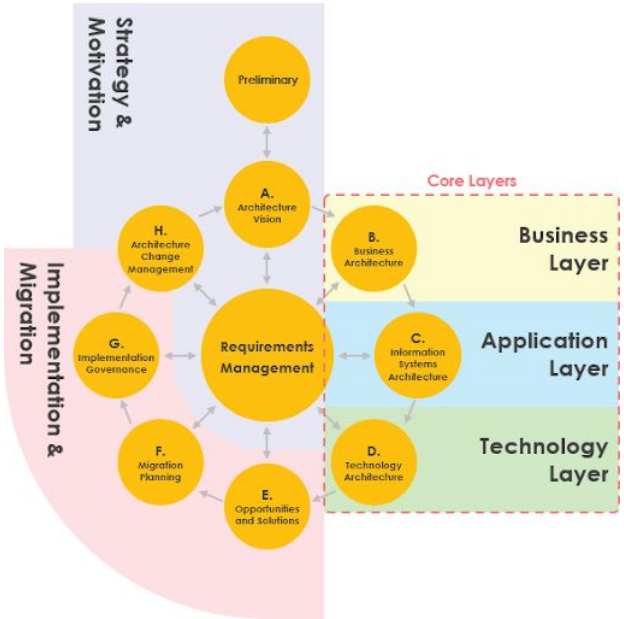
Template

1	<Viewpoint Name>
2	Overview
3	Concerns and stakeholders
3.1	Concerns
3.2	Typical stakeholders
3.3	“Anti-concerns” (optional)
4	Model kinds+
5	<Model Kind Name>
5.1	<Model Kind Name> conventions
5.2	<Model Kind Name> operations (optional)
5.3	<Model Kind Name> correspondence rules
6	Operations on views
7	Correspondence rules
8	Examples (optional)
9	Notes (optional)
10	Sources

References

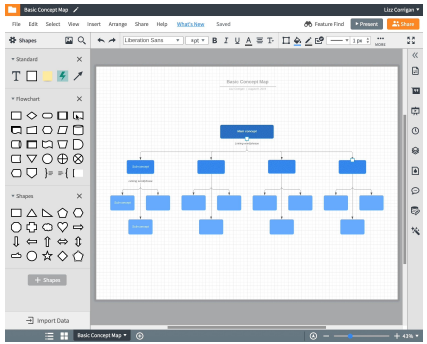
http://www.iso-architecture.org/ieee-1471/templates/

Architecture Document Templates. TOGAF

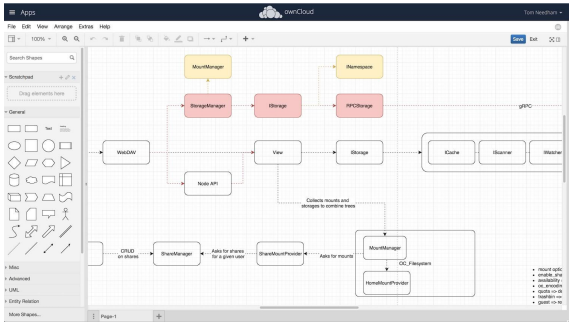


Architecture Toolset.

Architecture Toolset.



lucidchart



Draw.io

Visio

Practice II

10 minutes

Q&A session

5 minutes

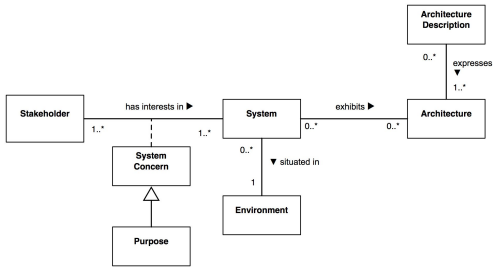
Summary.

5 minutes

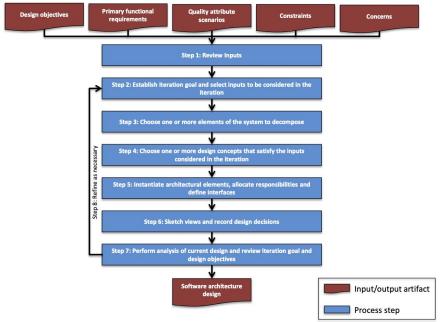
Summary.

- The **Software Architecture** is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.
- Architecture has **many context** and very **important**.
- Architect has complex role with common objective to **understand business**, **define Architecture Significant Requirement** which helps to **create Architecture**.

Summary.

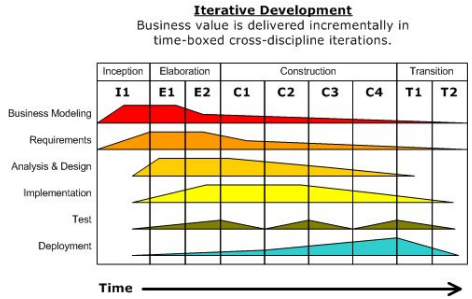


ISO/IEEE/IEC 42010

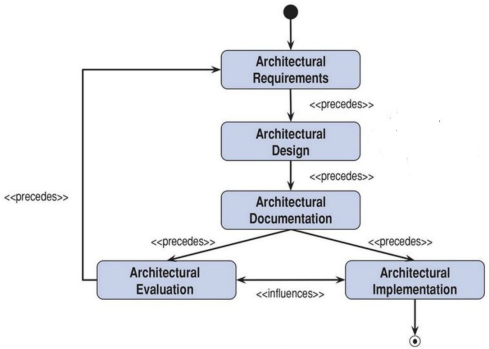


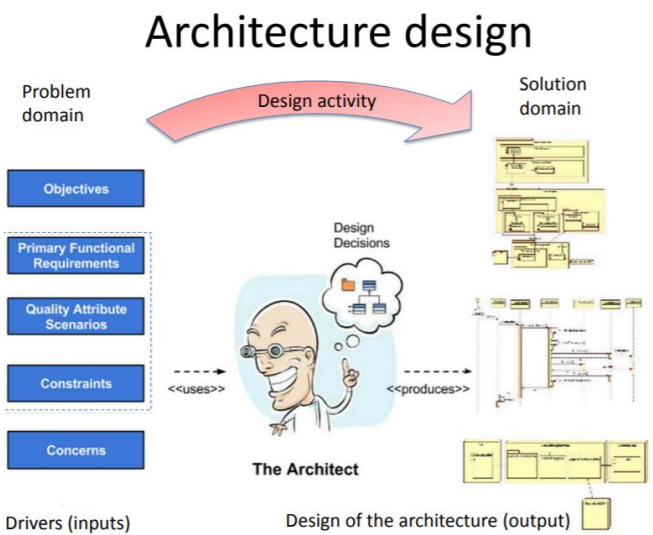
SEI Attribute Driven Design

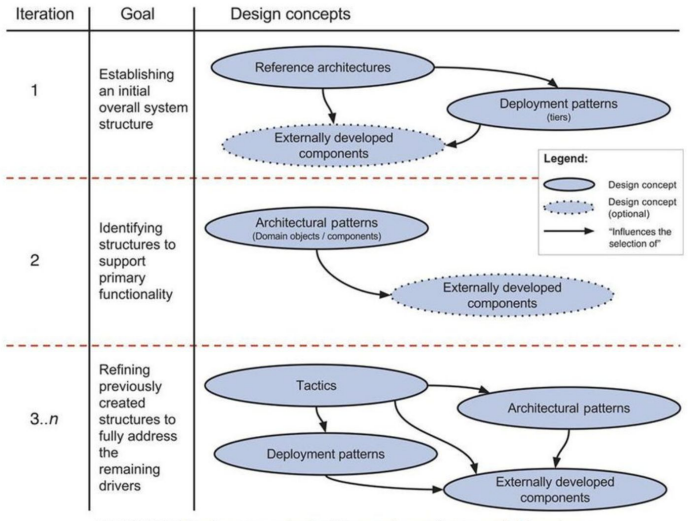
The TOGAF® Standard — Version 9.2



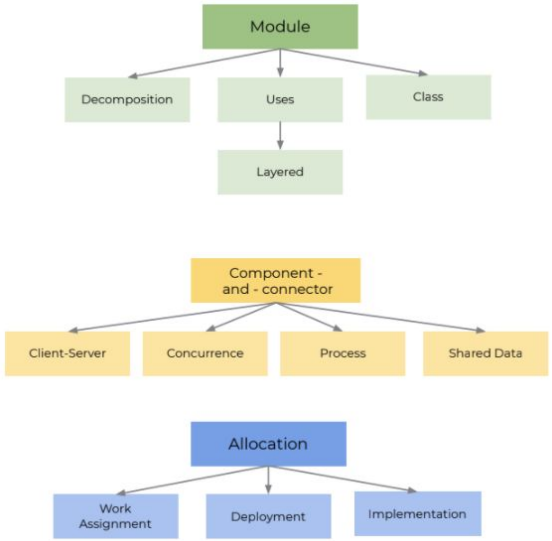
Summary.

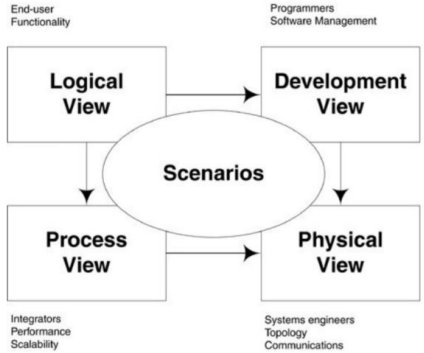

Architectural Requirements
<<precedes>>
Architectural Design
<<precedes>>
Architectural Documentation
<<precedes>>
<<precedes>>
Architectural Evaluation
<<influences>>
Architectural Implementation
<<precedes>>
<<precedes>>


Architecture design
Problem domain
Design activity
Solution domain
Objectives
Primary Functional Requirements
Quality Attribute Scenarios
Constraints
Concerns
<<uses>>
Design Decisions
<<produces>>
The Architect
Drivers (inputs)
Design of the architecture (output)

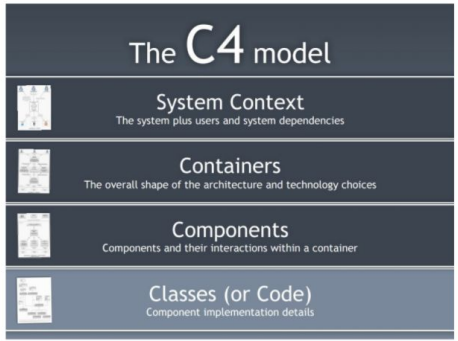

Iteration
Goal
Design concepts
1
Establishing an initial overall system structure
Reference architectures
Deployment patterns
(tiers)
Externally developed components
Legend:
Design concept
Design concept (optional)
"Influences the selection of"
2
Identifying structures to support primary functionality
Architectural patterns
(Domain objects / components)
Externally developed components
3..n
Refining previously created structures to fully address the remaining drivers
Tactics
Architectural patterns
Deployment patterns
Externally developed components

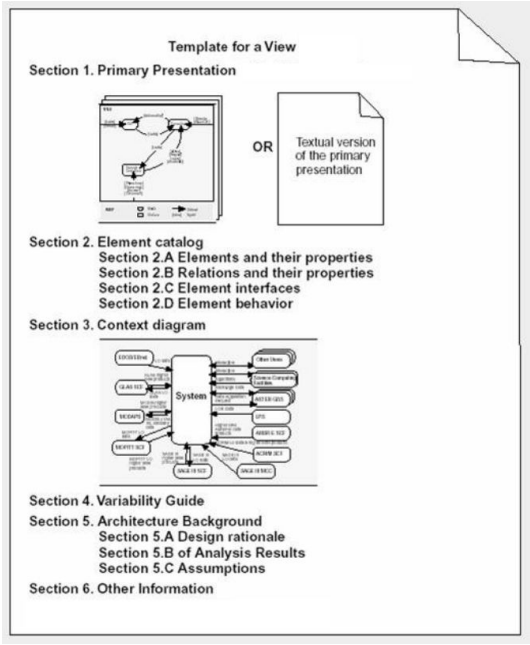
Summary.





4+1 View Model





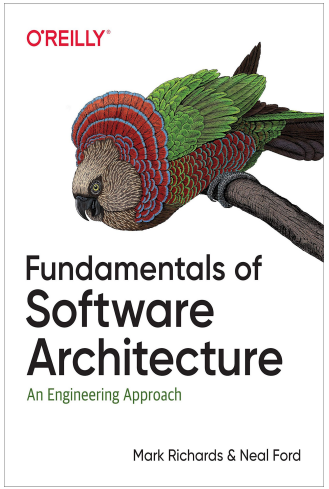
Learning Materials.

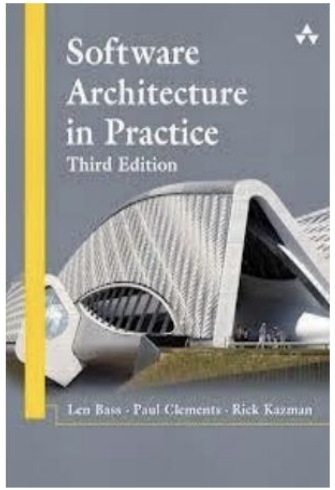
10 minutes

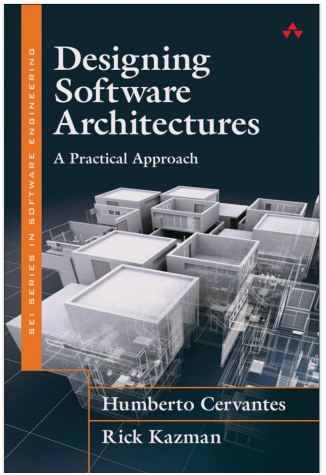
Resources.

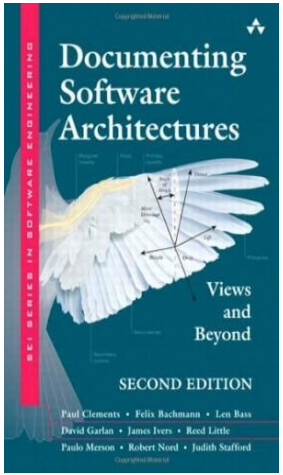
- https://www.developertoarchitect.com/
- O'Reilly course - https://learning.oreilly.com/videos/software-architecture-fundamentals/9781491998991
- https://martinfowler.com/architecture/
- SEI:
  - http://www.ece.ubc.ca/~matei/EECE417/BASS/index.html
  - Courses - https://www.sei.cmu.edu/education-outreach/courses/index.cfm
  - Views & Beyond Collection - https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=484159
  - SEI Example of wiki - https://wiki.sei.cmu.edu/confluence/display/SAD/Main+Page
- Types of diagrams - https://circle.visual-paradigm.com/diagram-examples/
- UML - https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/
- http://www.iso-architecture.org/ieee-1471/afs/
- Architectural Katas - https://archkatas.herokuapp.com/
- ADR's - https://adr.github.io/
- Body Of Knowledge:
  - SWEBOK - Software Engineering Body of Knowledge
  - PMBOK
  - BABOK
  - TBOK

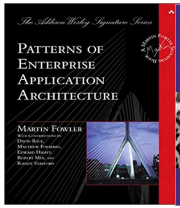
Books.


O'REILLY
Fundamentals of
Software
Architecture
An Engineering Approach
Mark Richards & Neal Ford


Software
Architecture
in Practice
Third Edition
Len Bass · Paul Clements · Rick Kazman


SEI SERIES IN SOFTWARE ENGINEERING
Designing
Software
Architectures
A Practical Approach
Humberto Cervantes
Rick Kazman

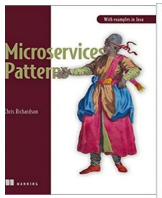

SEI SERIES IN SOFTWARE ENGINEERING
Documenting
Software
Architectures
Views
and
Beyond
SECOND EDITION
Paul Clements • Felix Bachmann • Len Bass
David Garlan • James Ivers • Reed Little
Paulo Merson • Robert Nord • Judith Stafford

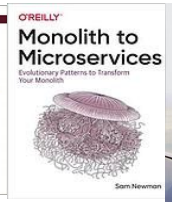

Patterns of
Enterprise
Application
Architecture
Martin Fowler


O'REILLY
Software
Architecture
Patterns
Mark Richards

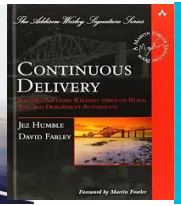

Clean Architecture
A Craftsman's Guide to
Software Structure and Design
Robert C. Martin

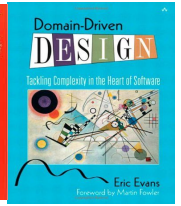

O'REILLY
Building
Microservices
Sam Newman


Microservices
Patterns


O'REILLY
Monolith to
Microservices
Evolutionary Patterns to Transform
Your Monolith
Sam Newman


Release It!
Second Edition
Design and Deploy
Production-Ready Software


Continuous
Delivery
Jez Humble
David Farley


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans


THANK YOU