CIS 422
Software Methodologies I

Chapter 11
Software Architecture

Professor: Juan J. Flores

jflore10@uoregon.edu

UNIVERSITY OF OREGON

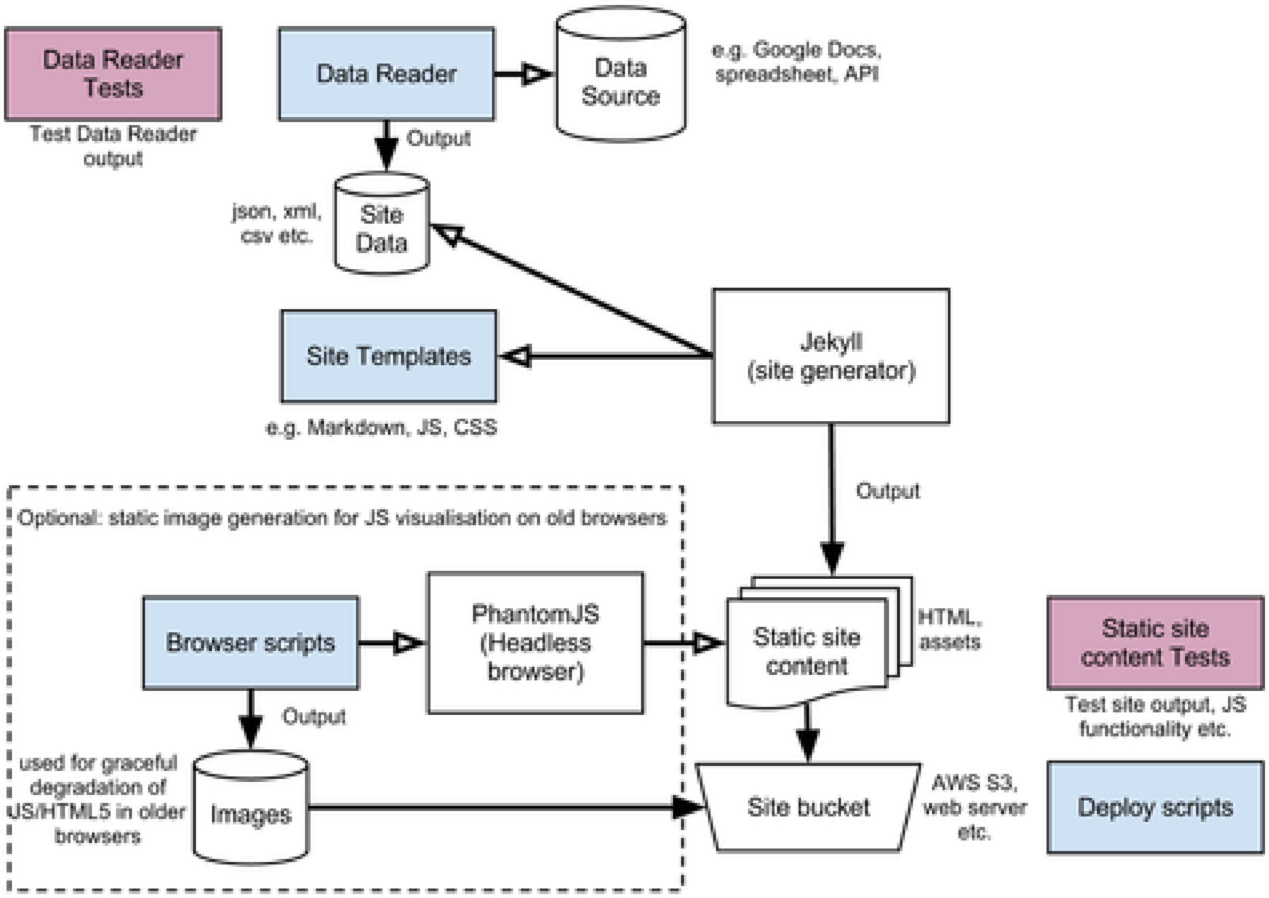
Meetings Interesting Ideas

- Static vs. Dynamic Architecture Models
- Why a tree, why bother?
- How to store a function in a tree node?
- User Interface
  - Direct manipulation
  - Command Line
  - Library
- You will be better prepared for Project 2 than you are for project 1

Static Models

- A static model of a system shows the system's structure.
- It emphasizes the parts that make up the system.
- Use static models to define class names, attributes, method signatures, and packages.
- UML diagrams that represent static models include ***class diagrams, object diagrams,*** and ***use case diagrams***.

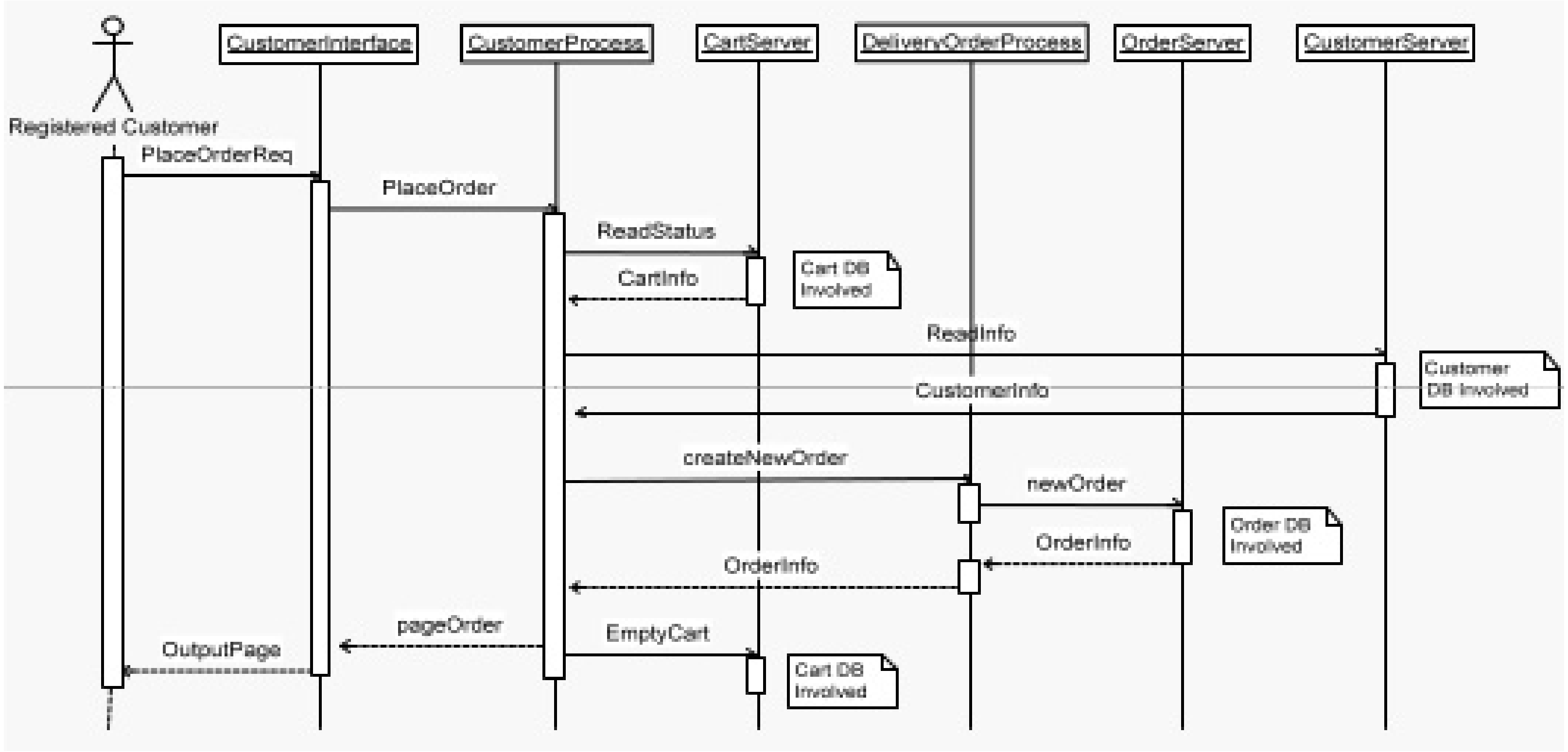
Static Models



Dynamic Models

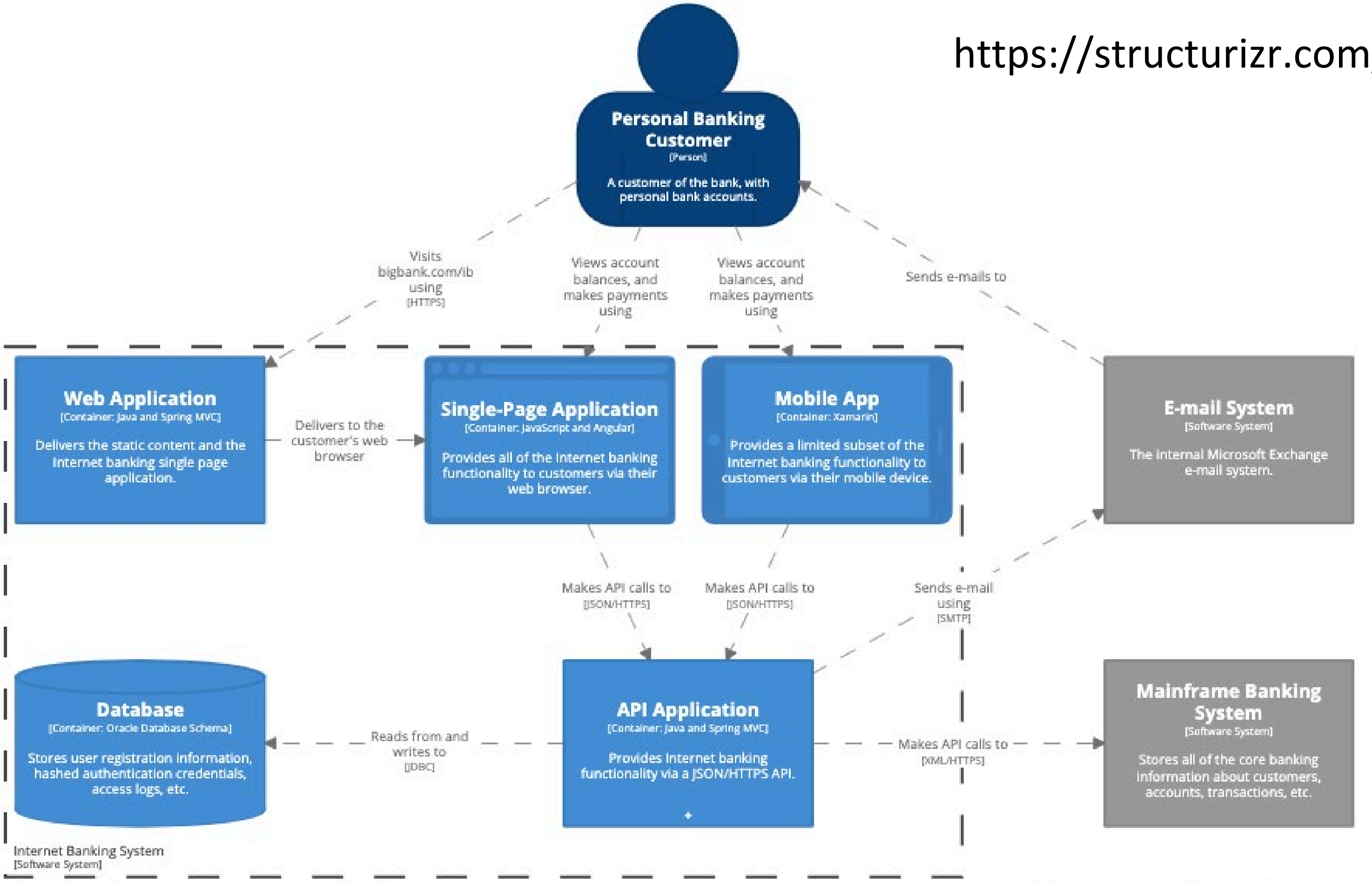
- A dynamic model of a system shows the system's behavior, for example, how the system behaves in response to external events.
- It lets you identify the objects needed and how those objects work together through methods and messages.
- Use dynamic models to design the logic and behavior of the system.
- UML diagrams that represent dynamic models include ***sequence diagrams, communication diagrams, state diagrams***, and ***activity diagrams.***

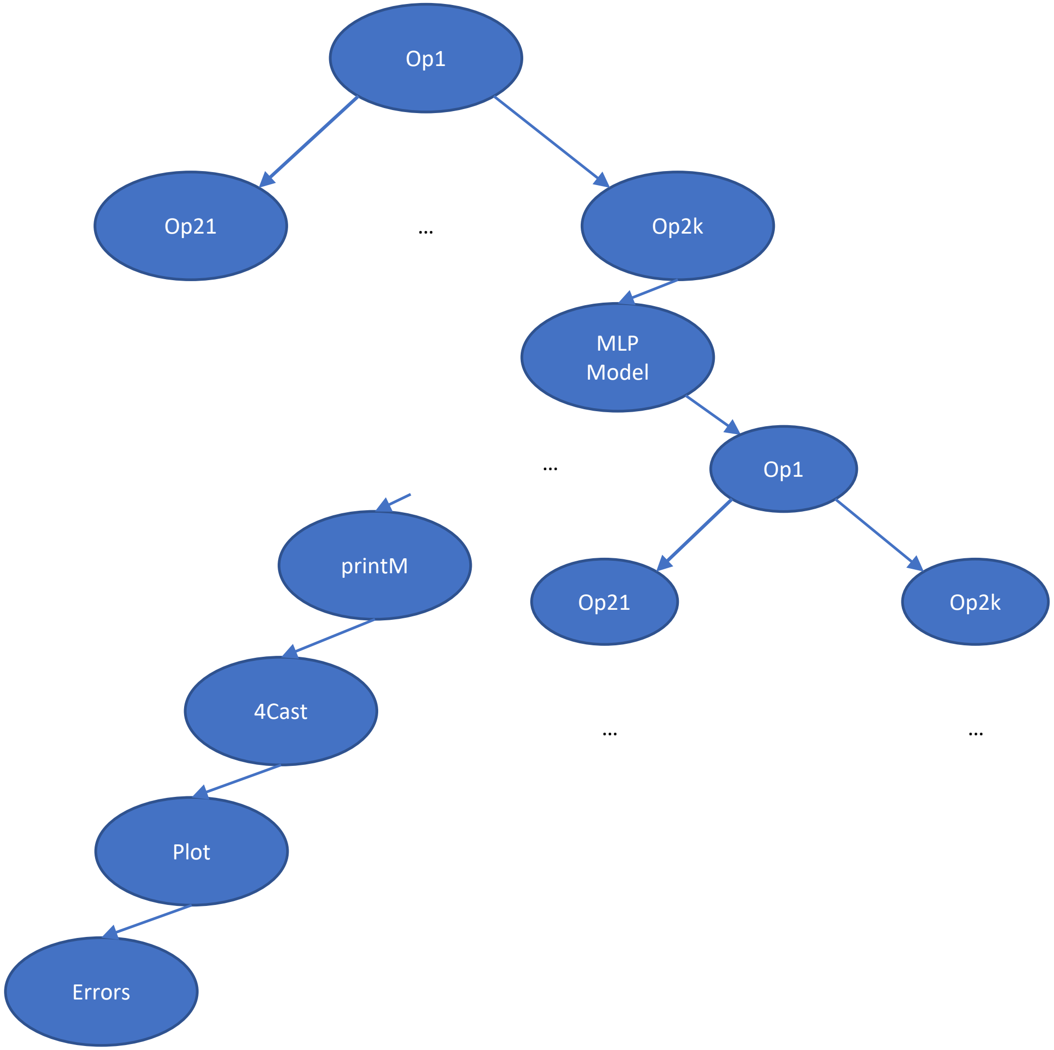
Dynamic Models

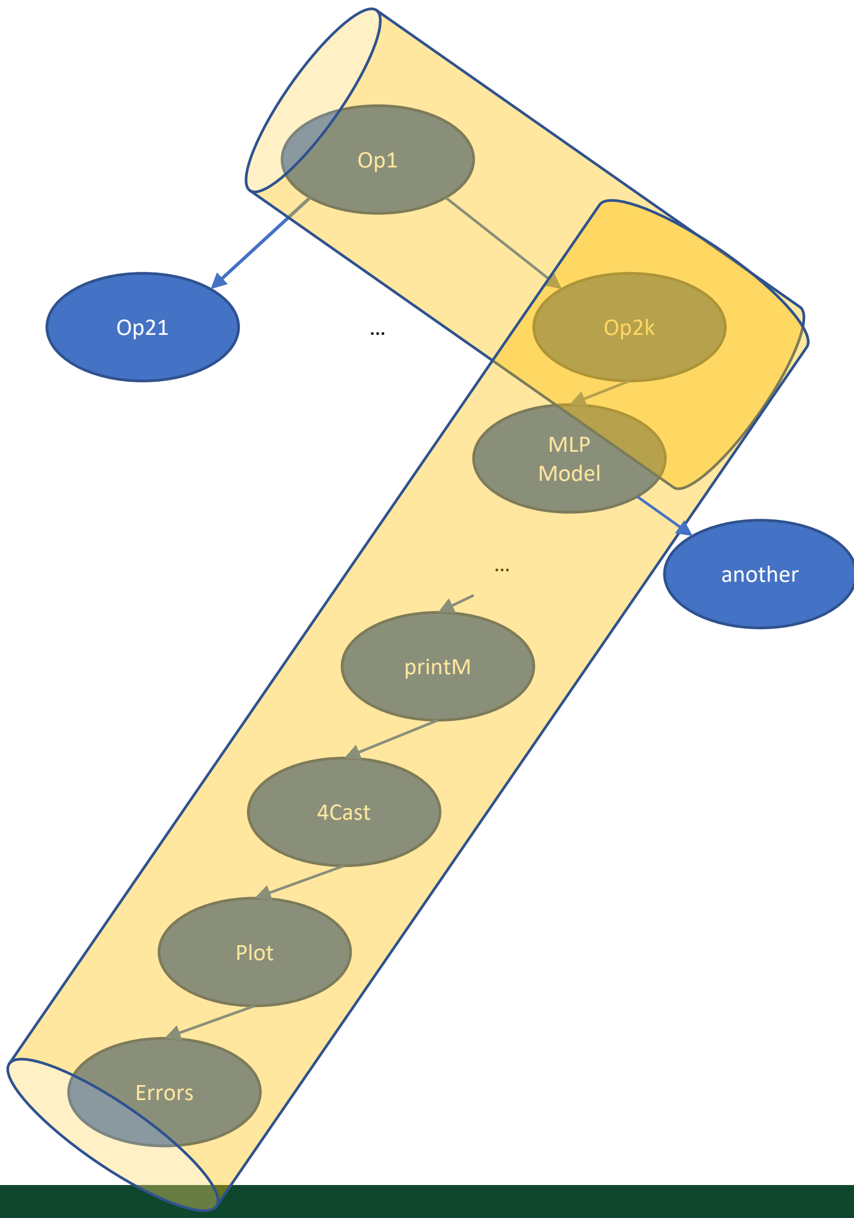


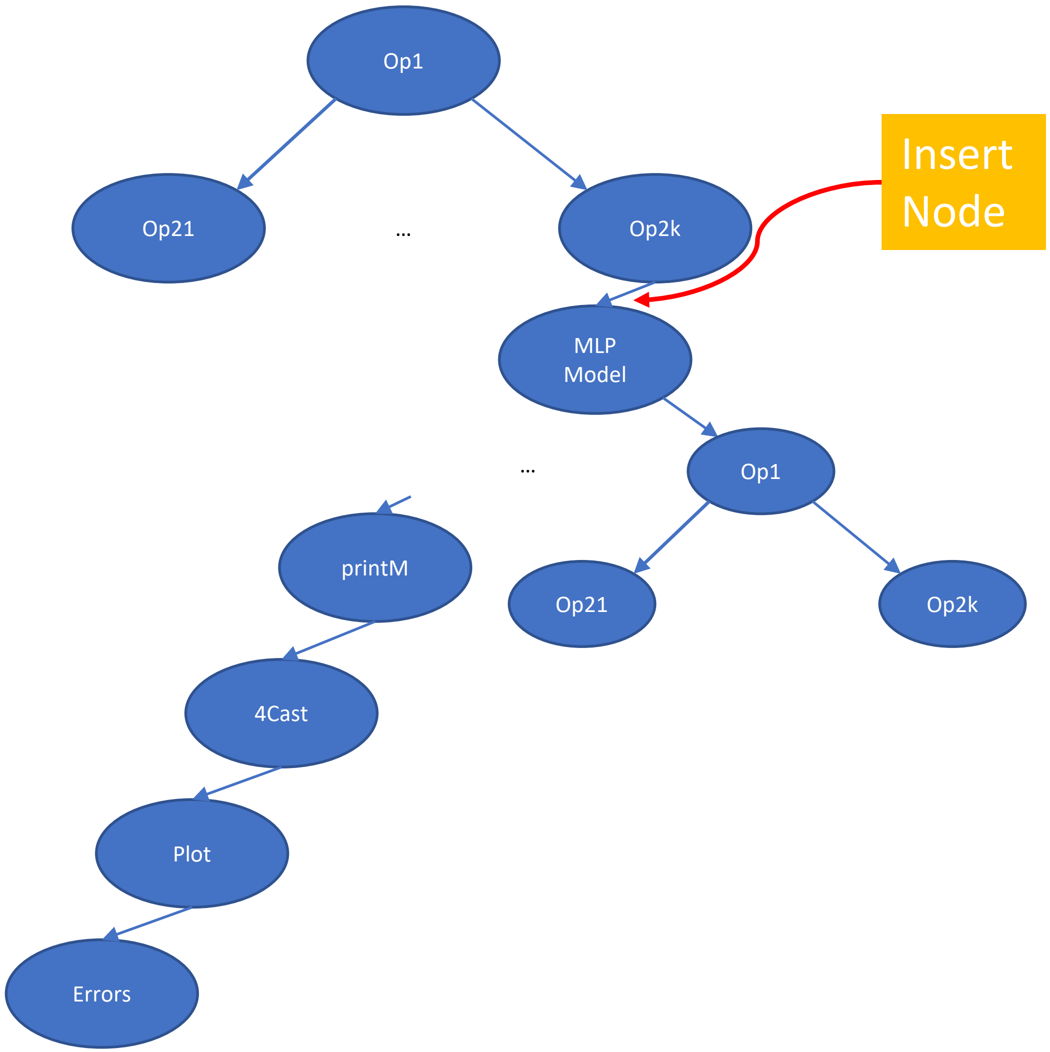
Dynamic Models

https://structurizr.com/




Op1
Op21
...
Op2k
MLP
Model
...
Op1
printM
Op21
Op2k
4Cast
...
...
Plot
Errors


Op1
Op21
...
Op2k
MLP Model
another
...
printM
4Cast
Plot
Errors

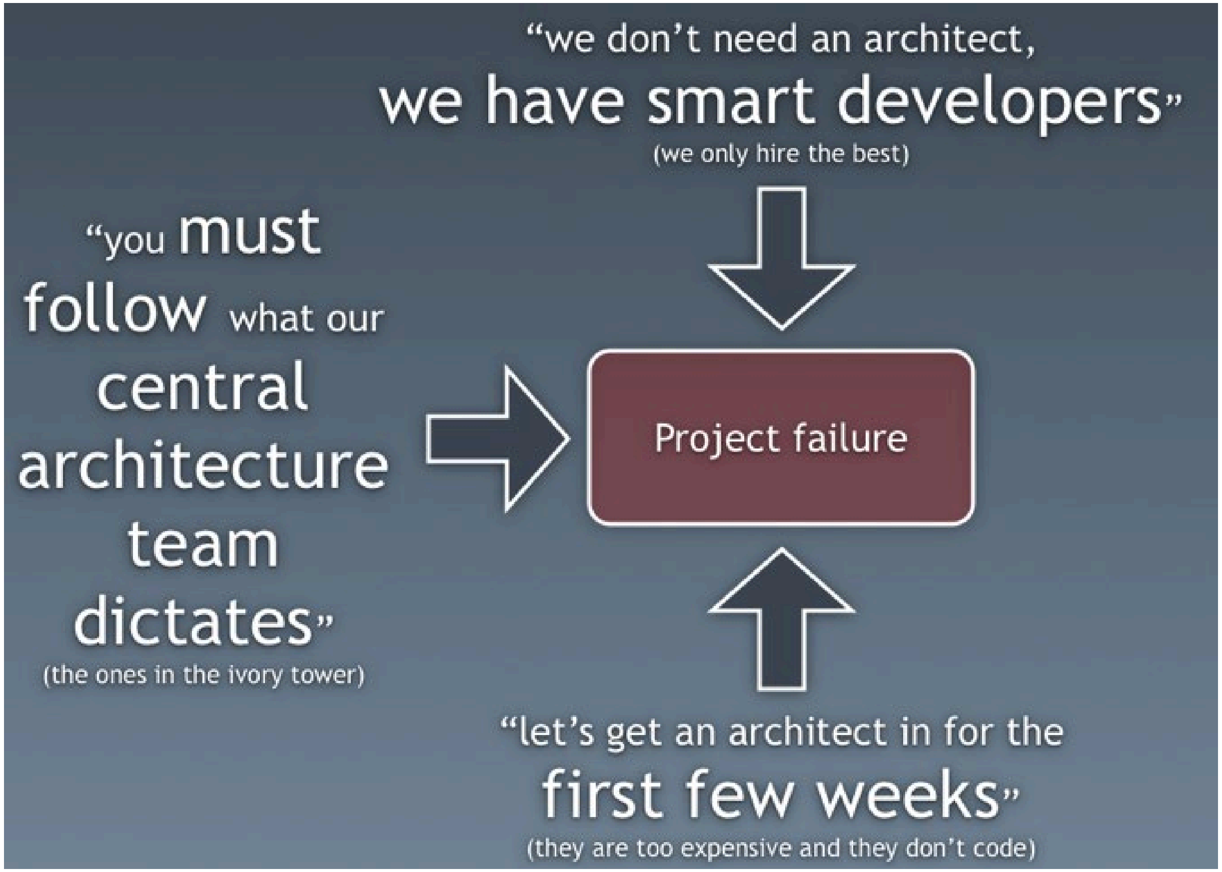

Op1
Op21
...
Op2k
Insert Node
MLP Model
...
Op1
printM
Op21
Op2k
4Cast
Plot
Errors

Overview

- Software Architecture. What is it, why bother?

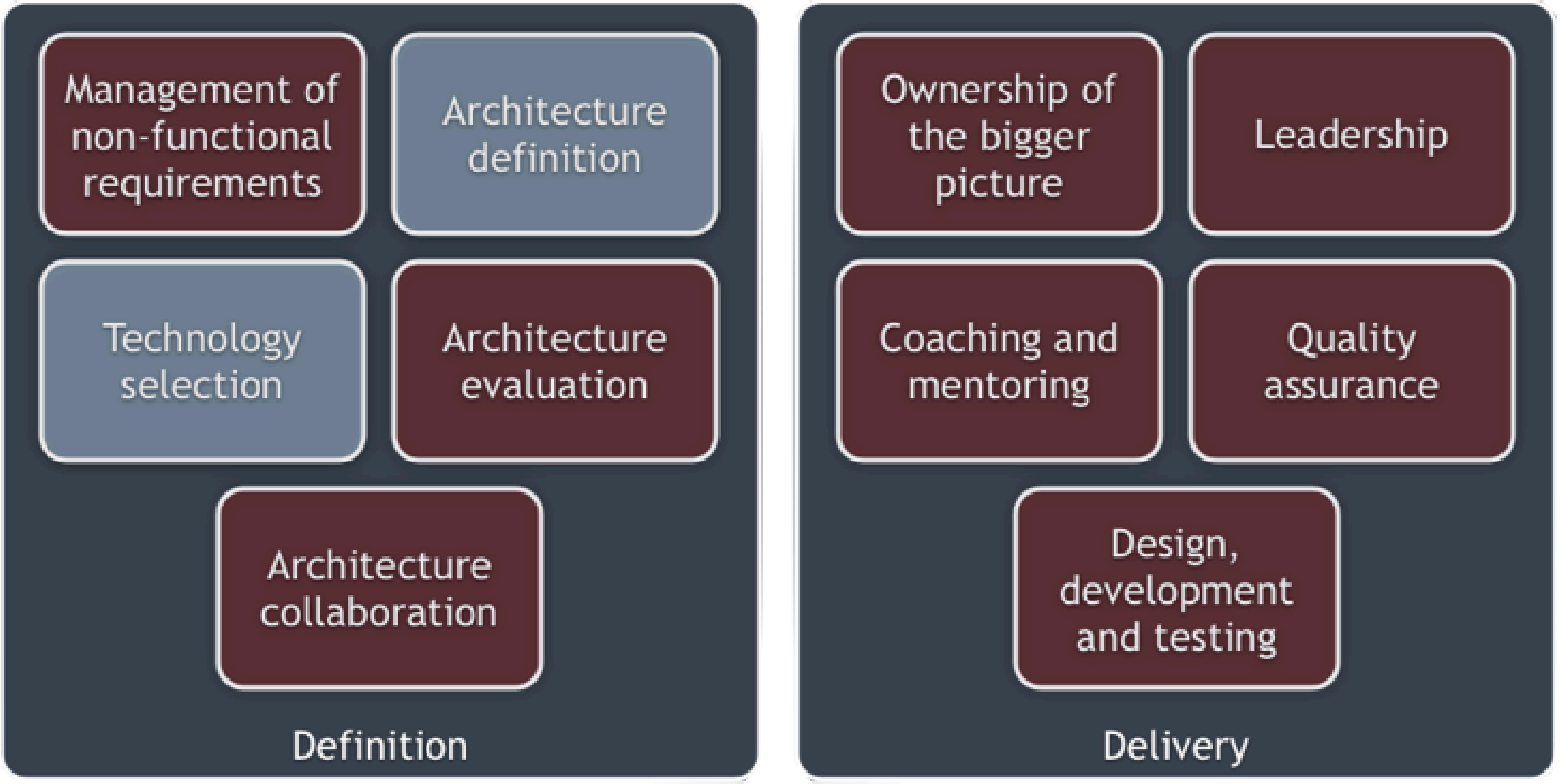
- Architecture Design
- Viewpoints and view models
- Architectural styles
- Architecture asssessment
- Role of the software architect

The Role of the Architect



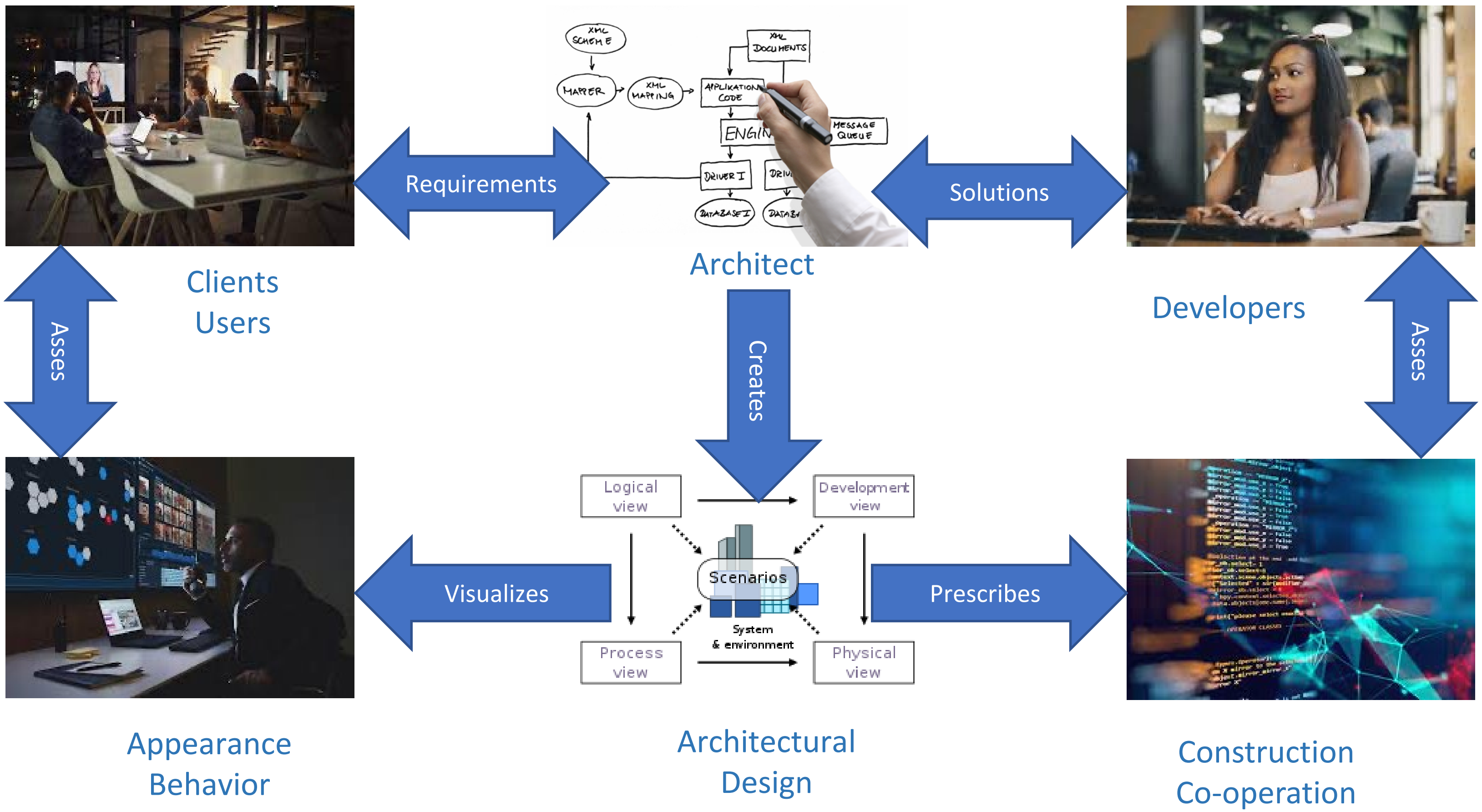
http://www.codingthearchitecture.com/presentations/sa2008-why-software-projects-fail/

The Role of the Architect

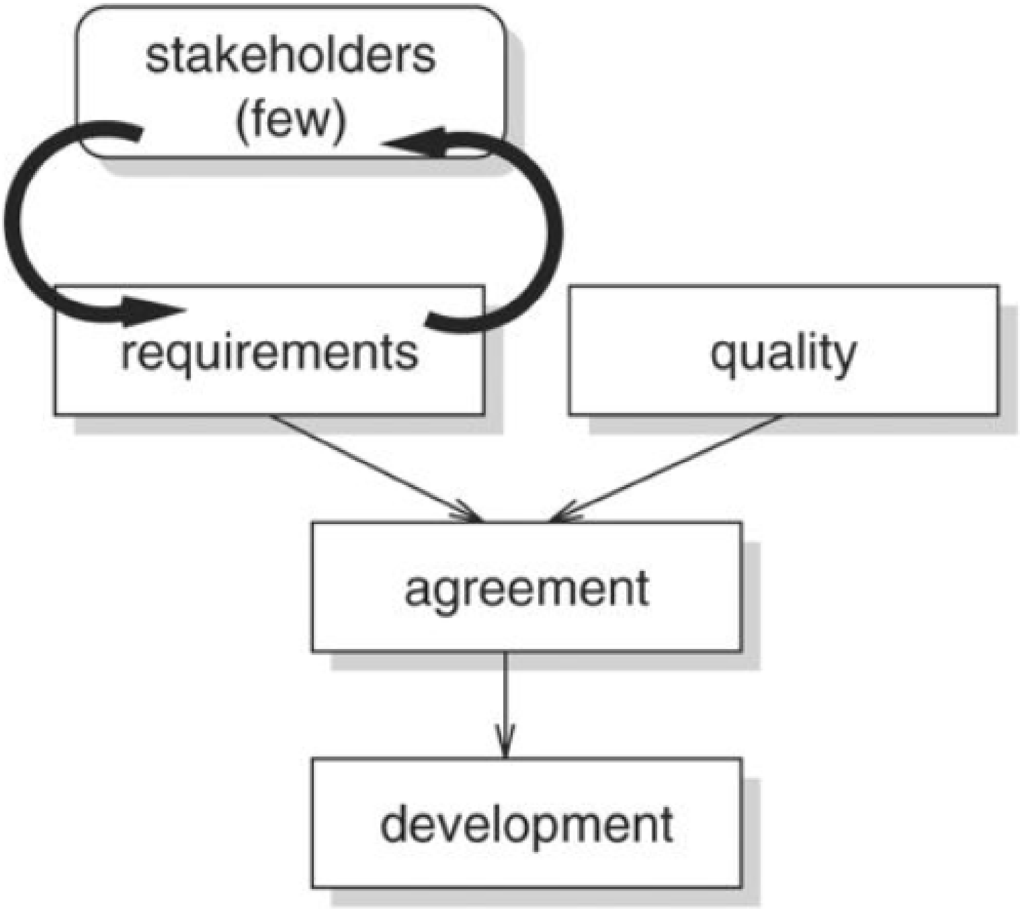


http://www.codingthearchitecture.com/presentations/sa2008-why-software-projects-fail/

The Role of the Architect



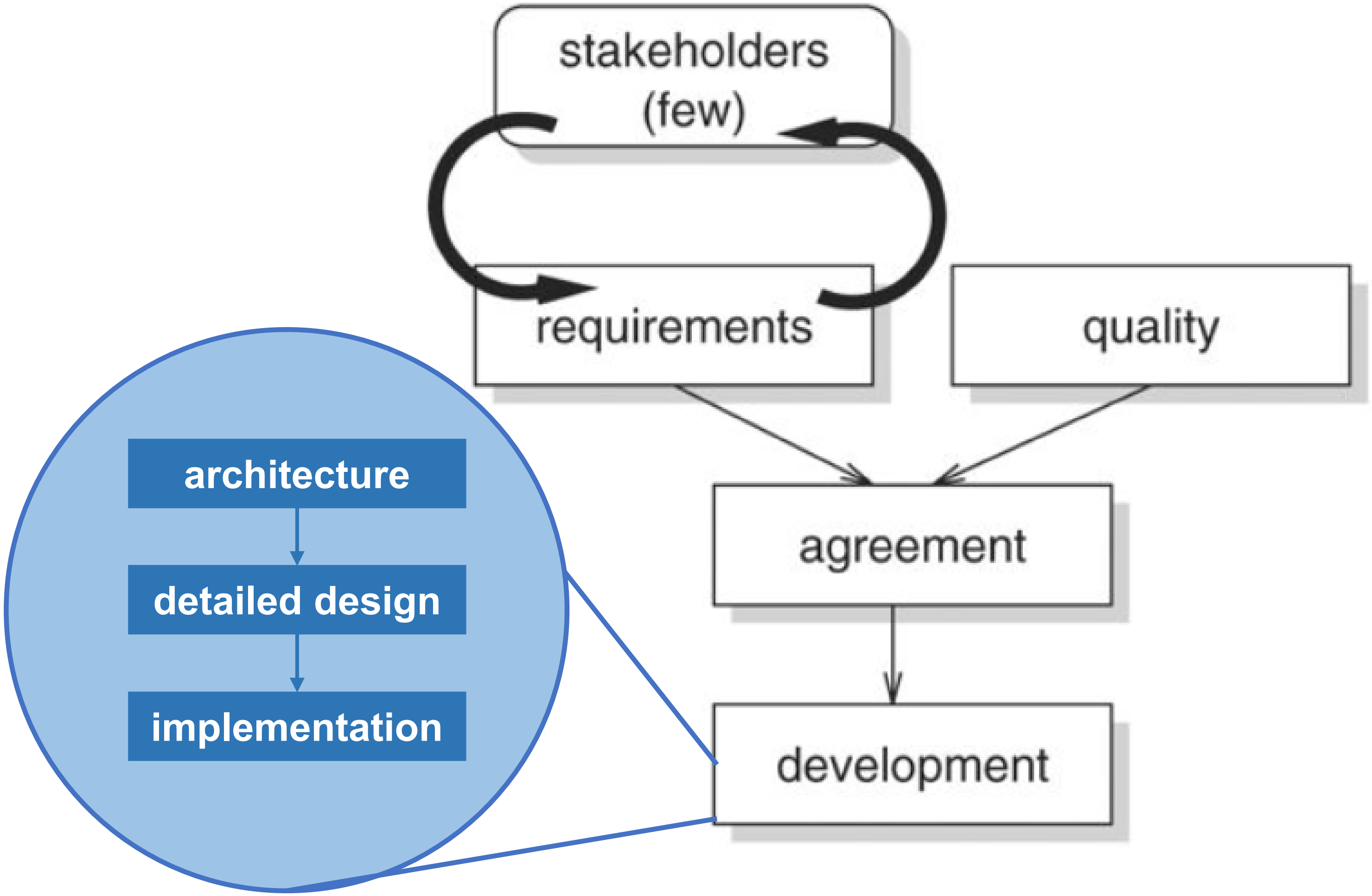
Pre-architecture life cycle



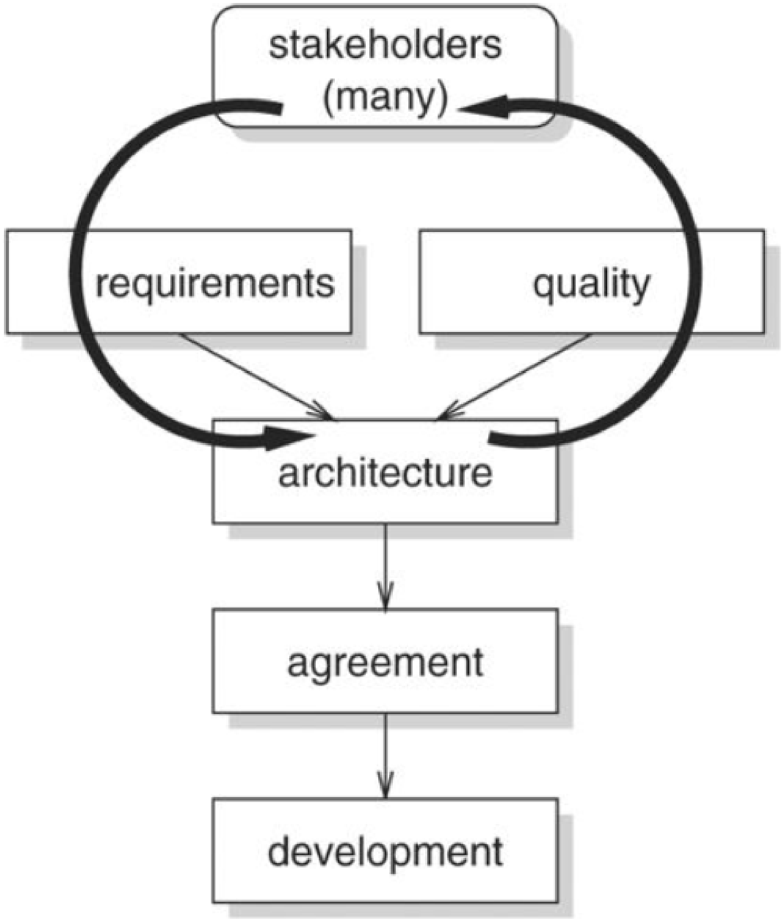
Characteristics

- Iteration mainly on functional requirements
- Few stakeholders involved
- No balancing of functional and quality requirements

Adding architecture, the easy way



Architecture in the life cycle



Characteristics

- Iteration on both functional and quality requirements
- Many stakeholders involved
- Balancing of functional and quality requirements

Why Is Architecture Important?

Architecture:

- is the vehicle for stakeholder communication
- manifests the earliest set of design decisions
  - Constraints on implementation
  - Dictates organizational structure
  - Inhibits or enable quality attributes
- is a transferable abstraction of a system
  - Product lines share a common architecture
  - Allows for template-based development
  - Basis for training

Software architecture, definition

The software architecture of a computing system is the structure or structures of the system –software elements and their externally visible properties– and the relationships among them.

- (Bass et al., 2003.)

Software Architecture

- Important issues raised in this definition:
  - multiple system structures
  - externally observable properties of components

- The definition does not include:
  - the process
  - rules and guidelines
  - architectural styles

Architectural Structures

- Module structure
- Conceptual, or logical structure
- Process, or coordination structure
- Physical structure
- Uses structure
- Calls structure
- Data flow
- Control flow
- Class structure

Software Architecture, definition rev.

Architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution

IEEE Standard

Recommended Practice for Architectural Descriptions, 2000.

Software Architecture

- Architecture is *conceptual*.
- Architecture is about *fundamental* things.
- Architecture exists in some *context*.

Other points of view

- Architecture is high-level design
- Architecture is overall structure of the system
- Architecture is the structure, including the principles and guidelines governing their design and evolution over time
- Architecture is components and connectors

Software Architecture & Quality

- Quality is central in software architecting
  - Software architecture is devised to gain insight in the qualities of a system as early as possible
- Some qualities are observable via execution
  - performance, security, availability, functionality, usability
- Some are not observable via execution
  - modifiability, portability, reusability, integrability, testability

Overview

- What is it, why bother?
- Architecture Design
- Viewpoints and view models
- Architectural styles
- Architecture asssessment
- Role of the software architect

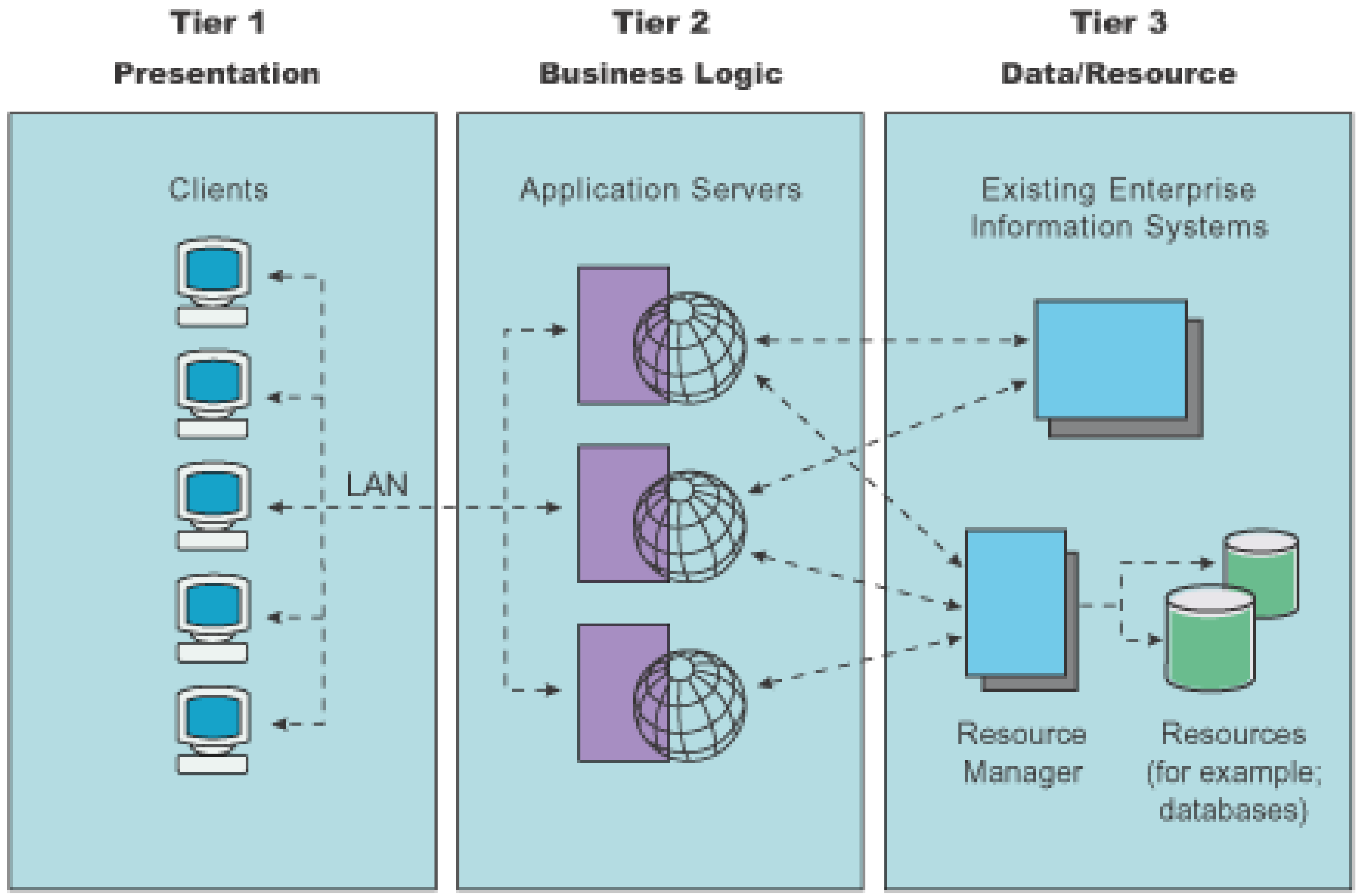
Attribute-Driven Design (Bass et al, Ch 7)

- Choose module to decompose
- Refine this module:
  - choose architectural drivers (quality is driving force)
  - choose pattern that satisfies drivers
  - apply pattern
- Repeat steps

Example ADD iterations

- Top-level
  - usability $\Rightarrow$ separate user interface $\Rightarrow$ See three-tier architecture
- Lower-level, within user interface
  - security $\Rightarrow$ authenticate users
- Lower-level, within data layer
  - availability $\Rightarrow$ active redundancy

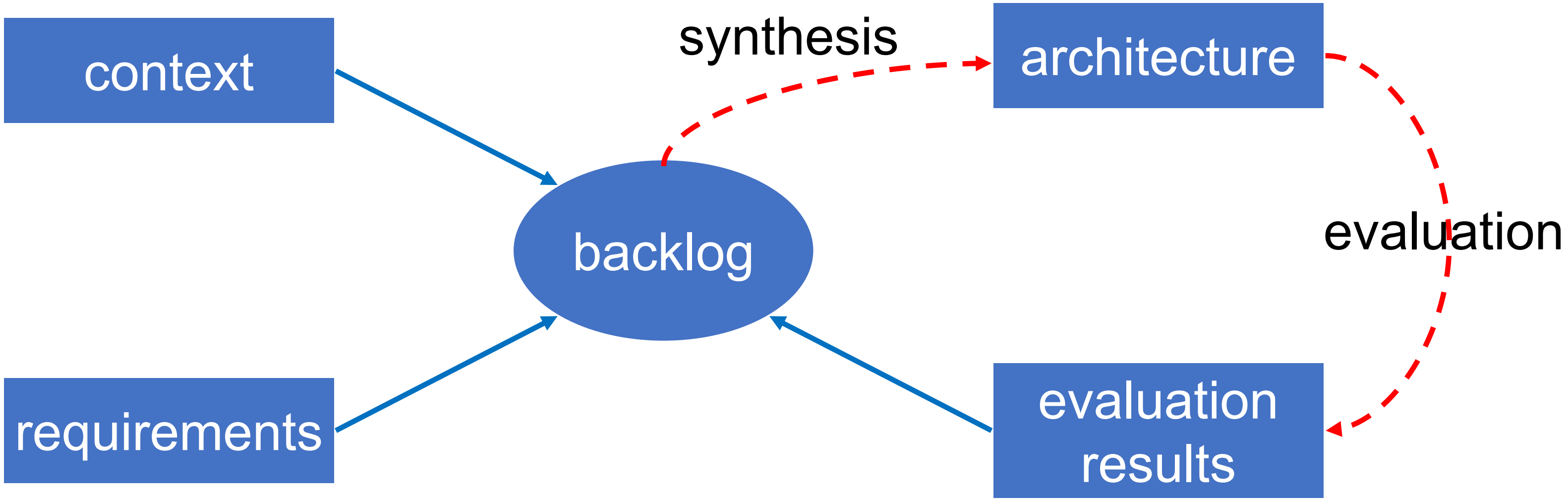
Example ADD iterations



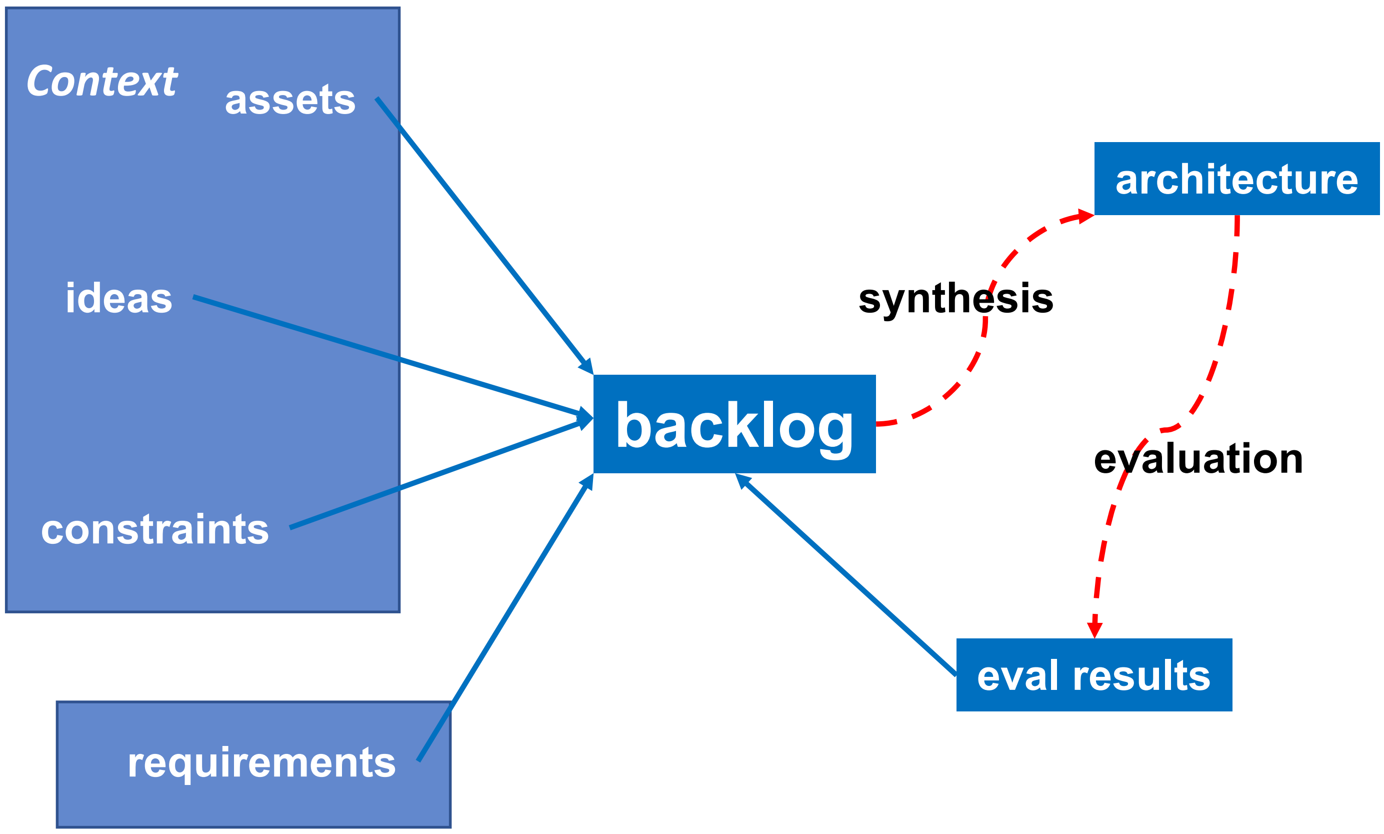
Generalized model

- Understand problem
- Solve it
- Evaluate solution

Global workflow in architecture design



Generalized Model

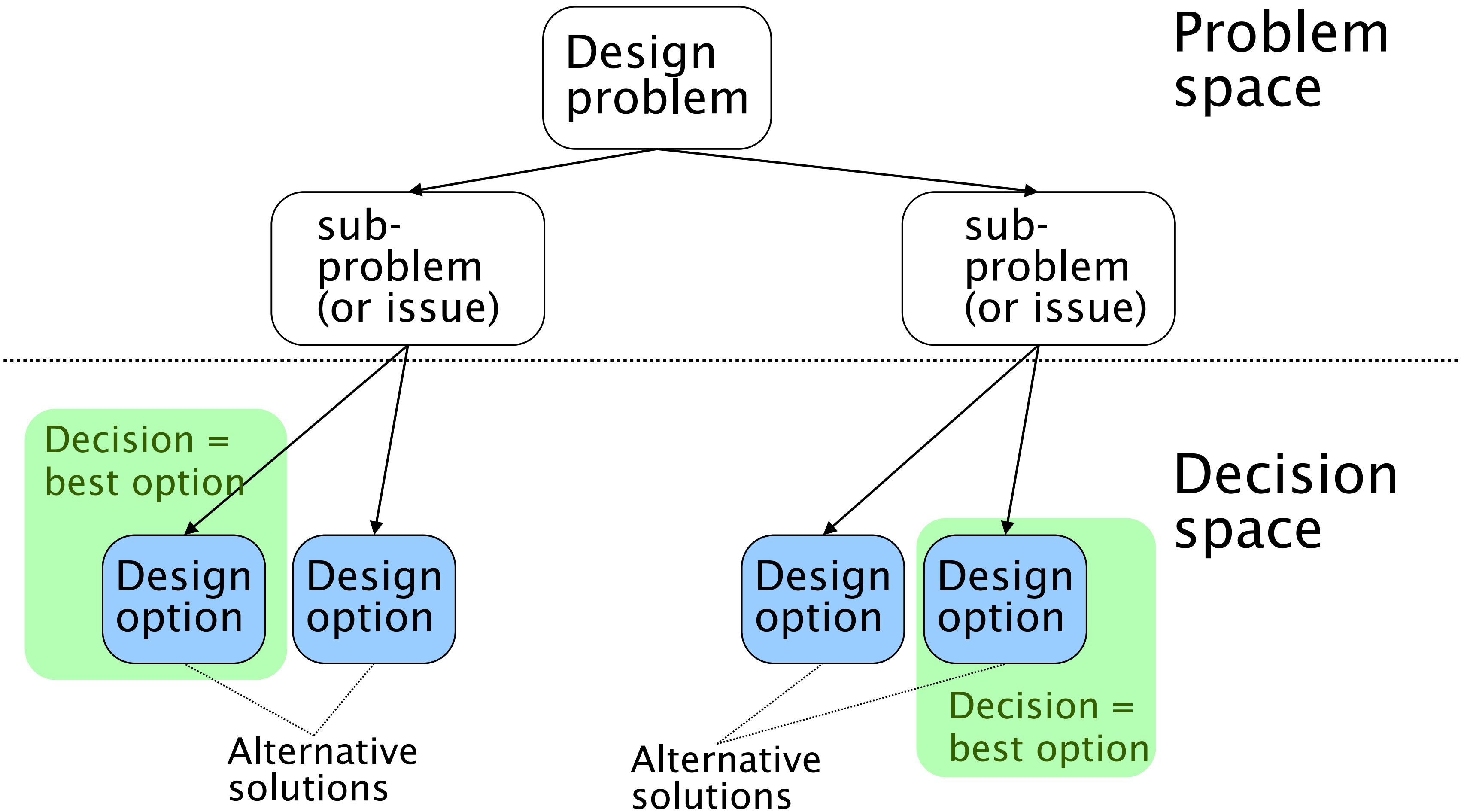


Design issues, options and decisions

Design issues

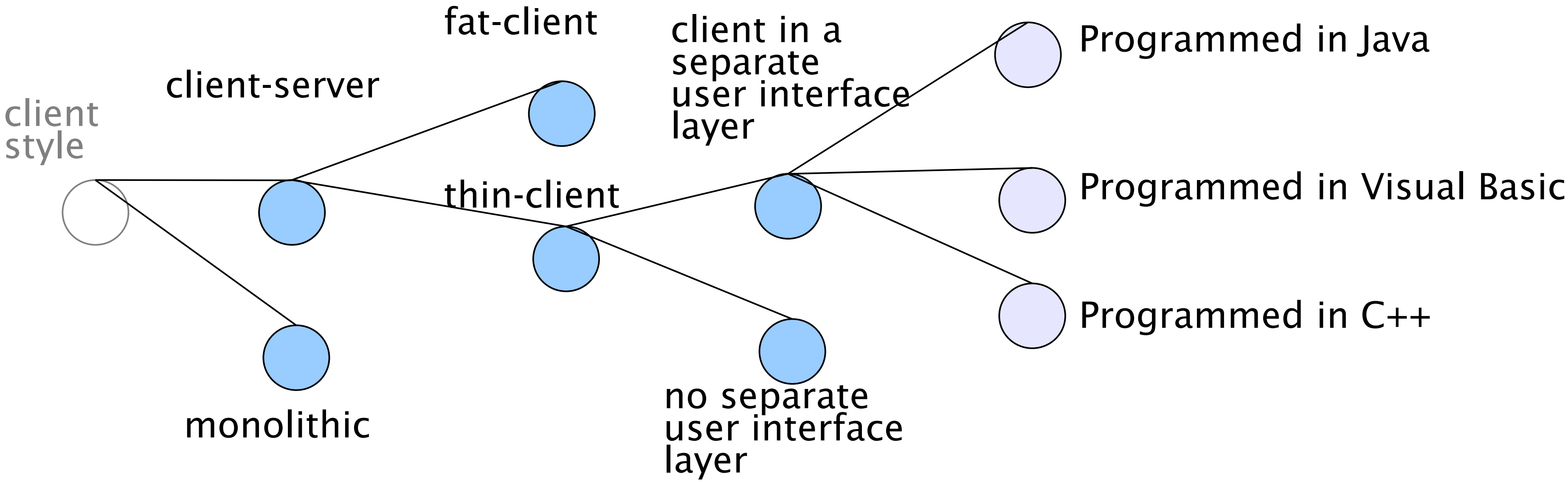
- Sub-problems of the overall design problem
- Issues have alternative solutions (aka *design options*)
- The architect makes a *design decision* to resolve each issue
  - This process involves choosing the best option from among the alternatives

Decision Making



Decision space

The space of possible designs that can be achieved by choosing different sets of alternatives.



More than just IT

- Technical and non-techical issues and options are intertwined
  - Architects deciding on the type of database

  versus
  - Management deciding on new strategic partnership

  or
  - Management deciding on budget

Types of Decisions

- Implicit, undocumented
  - Unaware, tacit, of course knowledge
- Explicit, undocumented
  - Vaporizes over time
- Explicit, explicitly undocumented
  - Tactical, personal reasons
- Explicit, documented
  - Preferred, exceptional situation

Documenting Design Decisions

- Prevents repeating (expensive) past steps
- Explains why this is a good architecture
- Emphasizes qualities and criticality for requirements/goals
- Provides context and background

Uses of design decisions

- Identify key decisions for a stakeholder
  - Make the key decisions quickly available. E.g., introducing new people and make them up to date.
  - …, Get a rationale, Validate decisions against reqs.
- Evaluate impact
  - What elements are impacted if we want to change an element (decisions, design, issues)?
  - Cleanup the architecture, identify important architectural drivers

Documenting design decisions

Element	Description
Issues	Design issues being addressed by this decision
Decision	The decision taken
Status	The status of the decision, e.g. pending, approved
Assumptions	The underlying assumptions about the environment in which the decision is taken
Alternatives	Alternatives considered for this decision
Rationale	An explanation of why the decision was chosen
Implications	Implications of this decision, such as the need for further decisions or requirements
Notes	Any additional information one might want to capture

Overview

- What is it, why bother?
- Architecture Design

- Viewpoints and view models

- Architectural styles
- Architecture asssessment
- Role of the software architect

Software design in UML

- UML Diagrams
  - Class diagrams
  - Case diagrams
  - Sequence diagrams
  - State Diagrams
  - …
- Who can read those diagrams?
- Which type of questions do they answer?
- Do they provide enough information?

Who can read those diagrams?

- Designer, programmer, tester, maintainer, etc.

- Client
- User

Which type of questions do they answer?

- How much will it cost?
- How secure will the system be?
- Will it perform?
- How about maintenance cost?
- What if requirement A is replaced by requirement B?

Analogy with building architecture

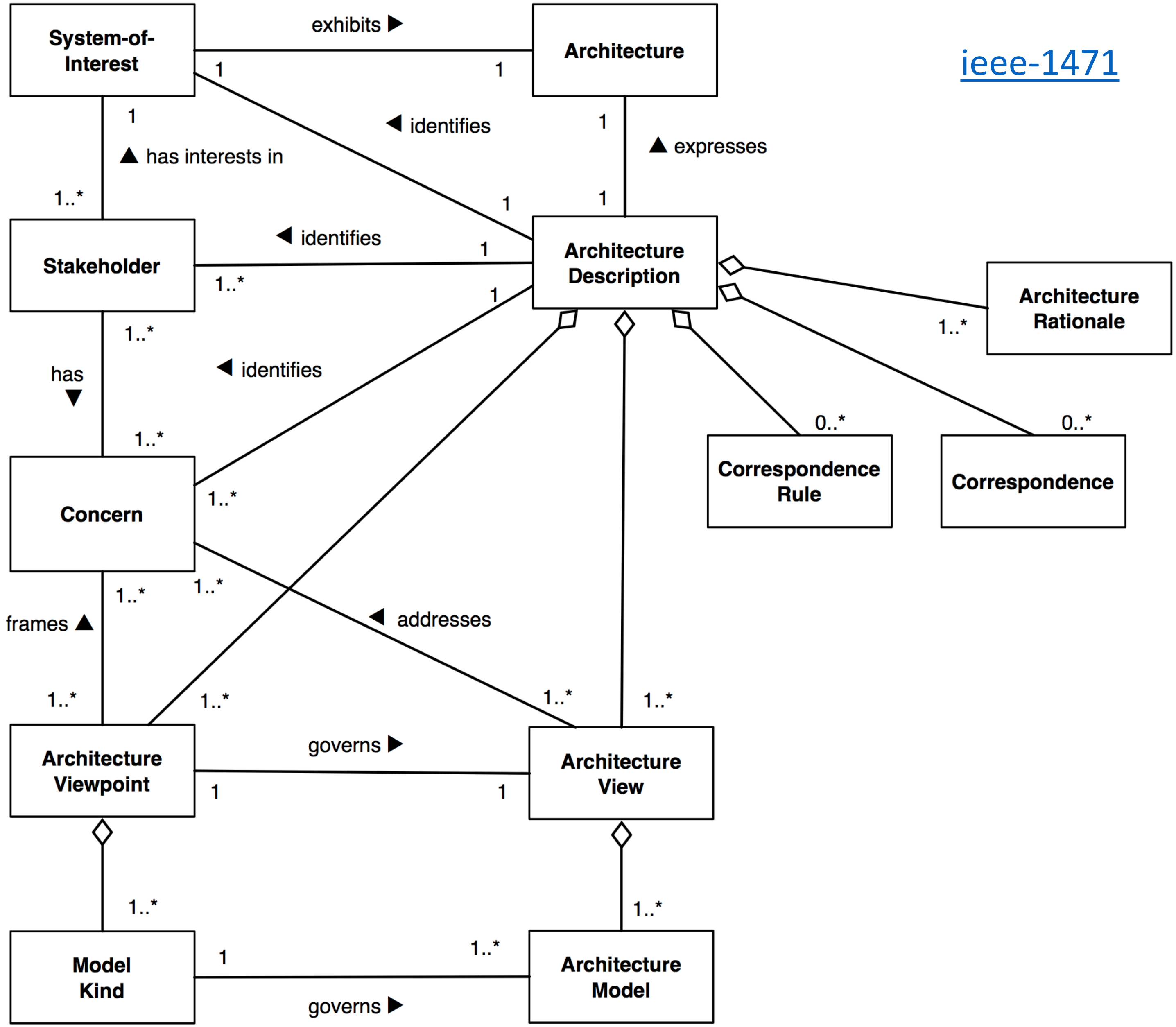
- Overall picture of building (client)
- Front view (client, “beauty” committee)
- Separate drawing for water supply (plumber)
- Separate drawing for electrical wiring (electrician)
- Etc.

Architecture presentations in practice

- By and large two flavors:
  - PowerPoint slides – for managers, users, consultants, etc.
  - UML diagrams, for technicians
- Different representations
- For different people
- For different purposes
- Architectural representations
  - descriptive
  - prescriptive

IEEE model for architectural descriptions

ieee-1471



Some terms (from IEEE standard)

- System stakeholder
  - an individual, team, or organization with interests or concerns related to a system
- View
  - system representation from the perspective of a set of concerns
- Viewpoint
  - purposes and audience for a view and the techniques or methods employed in constructing a view

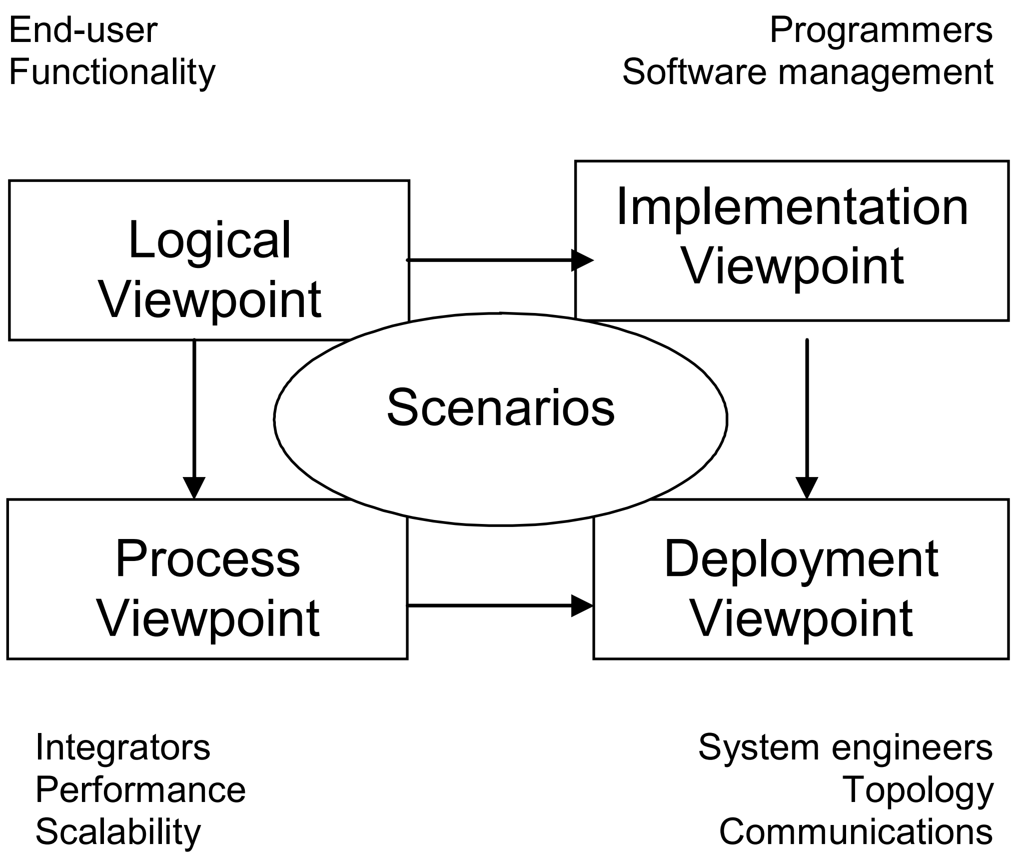
Stakeholders

- Architect
- Requirements engineer
- Designer
- Implementor
- Tester
- Integrator
- Maintainer
- Manager
- Quality assurance people
- Client

Viewpoint specification

- Viewpoint name
- Stakeholders addressed
- Concerns addressed
- Language, modeling techniques

Kruchten's 4+1 view model



4 + 1: Logical Viewpoint

- Describes the system in terms of design elements (components) and their interactions
- Supports the functional requirements, i.e., the services the system should provide to its end users.
- Typically, it shows the key abstractions (e.g., classes and interactions amongst them).

4 + 1: Implementation Viewpoint

- The implementation viewpoint focuses on the organization of the actual software modules in the software-development environment
- Provides a view of the system in terms of modules or packages and layers
- The software is packaged in small chunks (program libraries or subsystems)

4 + 1: Process Viewpoint

- Addresses dynamic concurrent aspects of the system (tasks, threads, processes and their interactions)

- It takes into account some nonfunctional requirements, such as performance, system availability, concurrency and distribution, system integrity, and fault-tolerance.

4 + 1: Deployment Viewpoint

- Allocates elements from the logical, process, and implementation viewpoints (networks, processes, tasks, and objects) to the various nodes.
- It takes into account the system's nonfunctional requirements such as system availability, reliability (fault-tolerance), performance (throughput), and scalability
- Only needed if the system is distributed

4 + 1: Scenario Viewpoint

- Small subset of important use cases to show that the elements of the four viewpoints work together seamlessly

- This viewpoint is redundant but plays two critical roles:
  - help designers discover architectural elements during design
  - validates and illustrates the architecture design

Architectural views

- View = representation of a structure
- Module views
  - Module is unit of implementation
  - Decomposition, uses, layered, class
- Component and connector (C & C) views
  - These are execution elements
  - Process (communication), concurrency, shared data (repository), client-server models
- Allocation views
  - Relationship between software elements and environment
  - Work assignment, deployment, implementation

Module views

- Decomposition
  - units are related by "is a submodule of"
  - larger modules are composed of smaller ones
- Uses
  - calls, passes information to, etc.
  - important for modifiability
- Layered is special case of uses
  - layer *n* can only use modules from layers <*n*
  - layer *n* can only use modules from layer = *n-1*
- Class
  - generalization
  - relation "inherits from"

Component and connector views

- Process
  - units are processes
  - connection means communication or synchronization
- Concurrency:
  - determine opportunities for parallelism
  - connector = logical thread
- Shared data
  - shows how data is produced and consumed
- Client-server
  - cooperating clients and servers

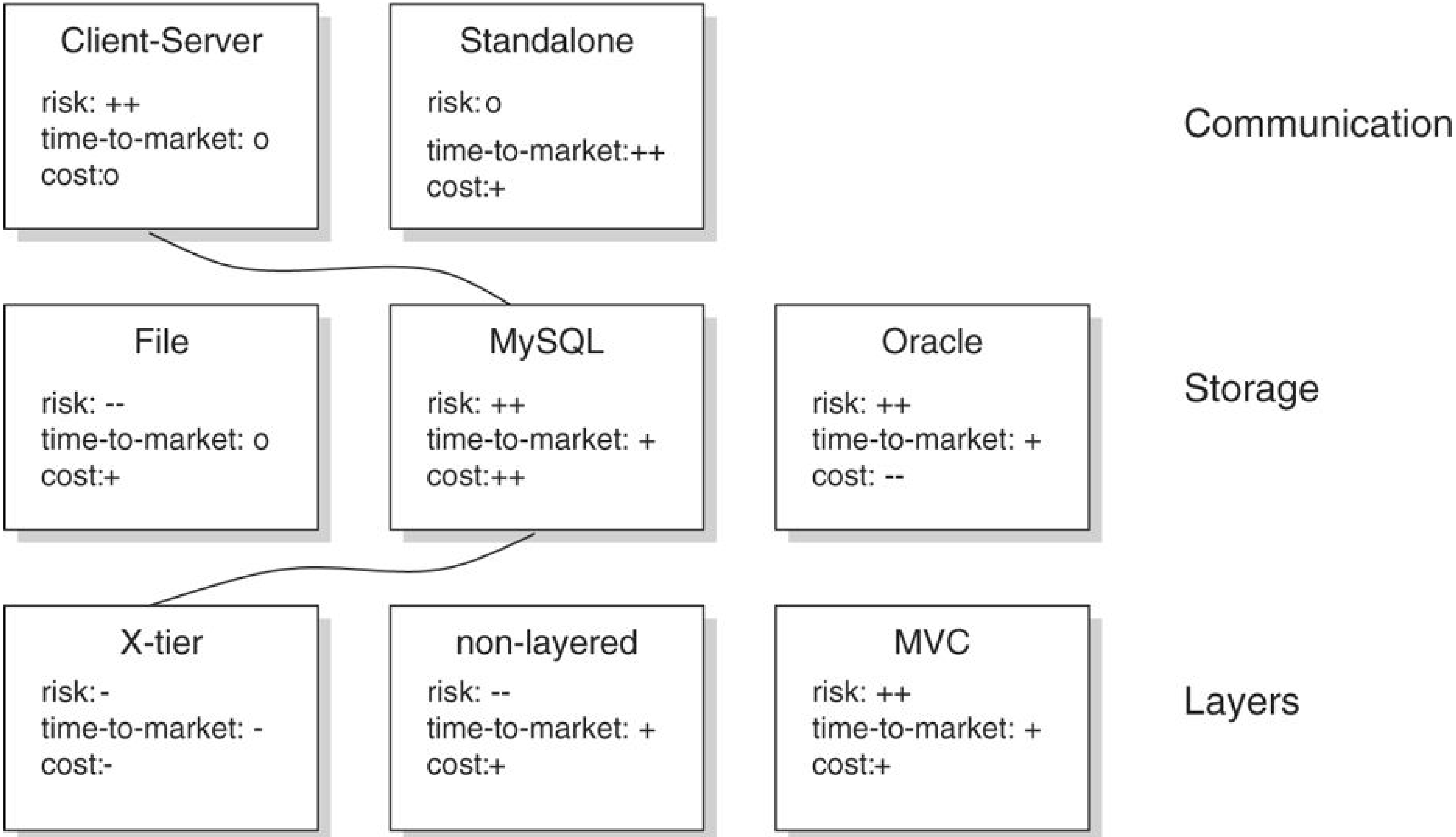
Allocation views

- Deployment
  - software is assigned to hardware
- Implementation
  - software is mapped onto file structures
- Work assignment
  - who is doing what

How to decide on which viewpoints

- What are the stakeholders and their concerns?
- Which viewpoints address these concerns?
- Prioritize and possibly combine viewpoints

Business viewpoint



Overview

- What is it, why bother?
- Architecture Design
- Viewpoints and view models

- Architectural styles

- Architecture asssessment
- Role of the software architect

Architectural styles

- A description of component and connector types
- A pattern of runtime control and/or data transfer

- Examples:
  - main program with subroutines
  - data abstraction
  - implicit invocation
  - pipes and filters
  - repository (blackboard)
  - layers of abstraction

Alexander’s patterns

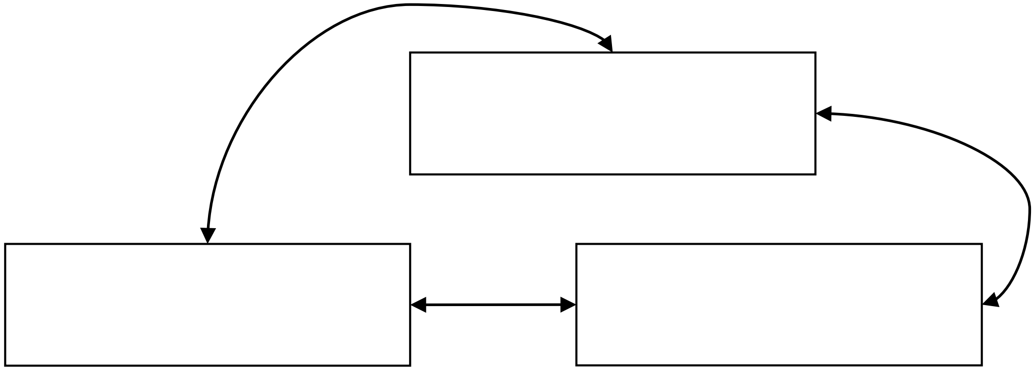
- Evidence shows that high buildings make people crazy

- High buildings have no advantage
  - are not cheaper
  - do not help to create open space
  - make life difficult for children
  - wreck open spaces near them
  - can actually damage people’s minds and feelings

- Keep the majority of buildings four stories high or less

- Certain buildings need to exceed this limit, but they should never be buildings for human habitation

Components and Connectors



- Components are connected by connectors
- Architecture's building blocks
- No standard notation has emerged yet

Types of components

- *Computational*
  - some kind of computation
  - e.g. function, filter
- *Memory*
  - maintains a collection of persistent data
  - E.g. data base, file system, symbol table
- *Manager*
  - contains state + operations
  - state is retained between invocations of operations
  - e.g. adt, server
- *Controller*
  - governs time sequence of events
  - e.g. control module, scheduler

Types of connectors

- Procedure call
  - local or RPC
- Data flow
  - e.g. pipes
- implicit invocation
- Message passing
- Shared data
  - e.g. blackboard or shared data base
- Instantiation

Architectural Styles Framework

- *Problem*
  - type of problem that the style addresses
  - characteristics of the reqs's guide the designer in his choice for a particular style
- *Context*
  - characteristics of the environment that constrain the designer
  - req's imposed by the style
- *Solution*
  - components
  - connectors
  - control structure
- *Variants*
- *Examples*

Main-program-with-subroutines style

Problem:

- hierarchy of functions
- result of functional decomposition
- single thread of control

Context:

- language with nested procedures

Solution:

- system model: modules in a hierarchy
- components: modules with local/global data
- connectors: procedure call
- control structure: single thread, centralized control: main program pulls the strings

Variants:

- OO versus non-OO

Abstract-data-type style

Problem:
- identify and protect related bodies of information
- data representations likely to change

Context:
- OO-methods guide the design
- OO-languages provide the class-concept

Solution:
- system model: component has its own local data
- connectors: procedure call (message)
- control structure: usually single thread

Variants:
- caused by language facilities

Implicit-invocation style

Problem:
- loosely coupled collection of components
- useful for applications which must be reconfigurable

Context:
- requires event handler, through OS or language.

Solution:
- system model: independent, reactive processes, invoked when an event is raised
- components: processes that signal events and react to events
- connectors: automatic invocation
- control structure: decentralized control

Variants:
- tool-integration frameworks
- languages with special features

Pipes-and-filters style

Problem:

- independent, sequential transformations on ordered data
- usually incremental ASCII pipes.

Context:

- series of incremental transformations
- OS-functions transfer data between processes
- error-handling difficult

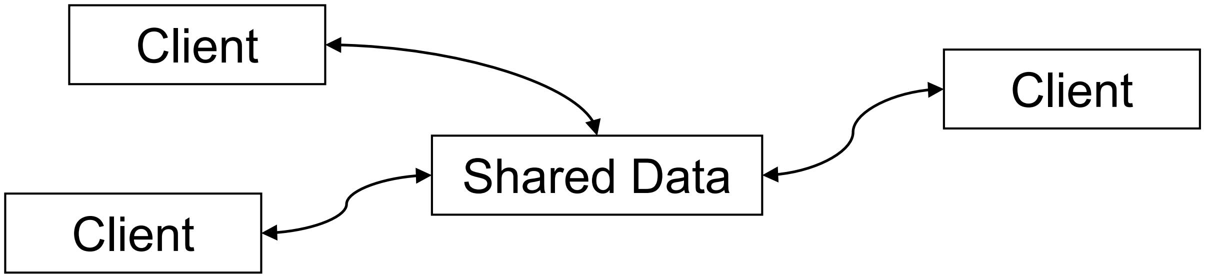
Solution:

- system model: continuous data flow; components incrementally transform data
- components: filters for local processing
- connectors: data streams (usually plain ASCII)
- control structure: data flow between components; component has own flow

Variants:

- From pure filters with little internal state to batch processes

Repository style



Problem:

- manage richly structured information
- data is long-lived

Context:

- shared data to be acted upon by multiple clients

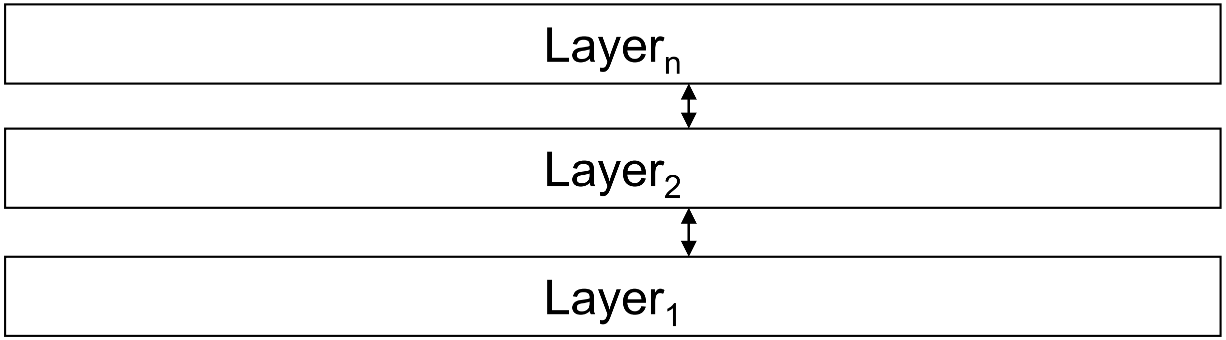
Solution:

- system model: centralized body of information. Independent computational elements.
- components: one memory, many computational
- connectors: direct access or procedure call
- control structure: varies, may depend on input or state of computation

Variants:

- traditional data base systems
- compilers
- whiteboard systems

Layered style



Problem:

- distinct, hierarchical classes of services
- concentric circles of functionality

Context:

- a large system that requires decomposition
- e.g., virtual machines, OSI model

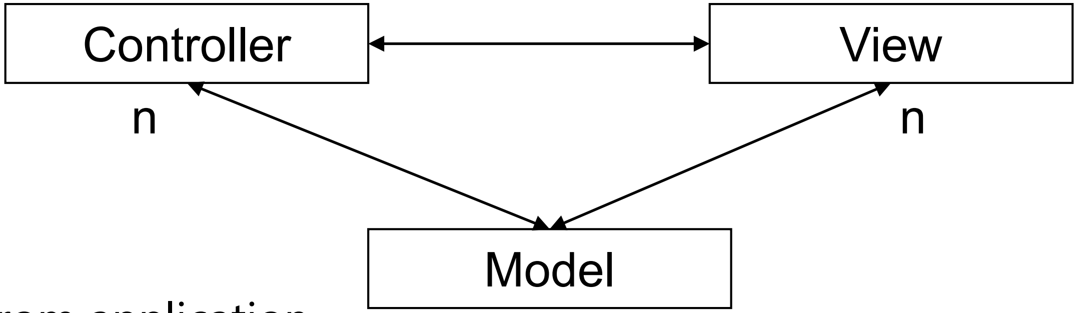
Solution:

- system model: hierarchy of layers, often limited visibility
- components: collections of procedures (module)
- connectors: (limited) procedure calls
- control structure: single or multiple threads

Variants:

- relaxed layering (level n calls any level k<=n)

Model-View-Controller (MVC) style



Problem:

- separation of UI from application
- expected UI adaptations

Context:

- interactive applications with a flexible UI

Solution:

- system model: UI (View and Controller Components) is decoupled from the application (Model component)
- components: collections of procedures (module)
- connectors: procedure calls
- control structure: single thread

Variants:

- document-View

Overview

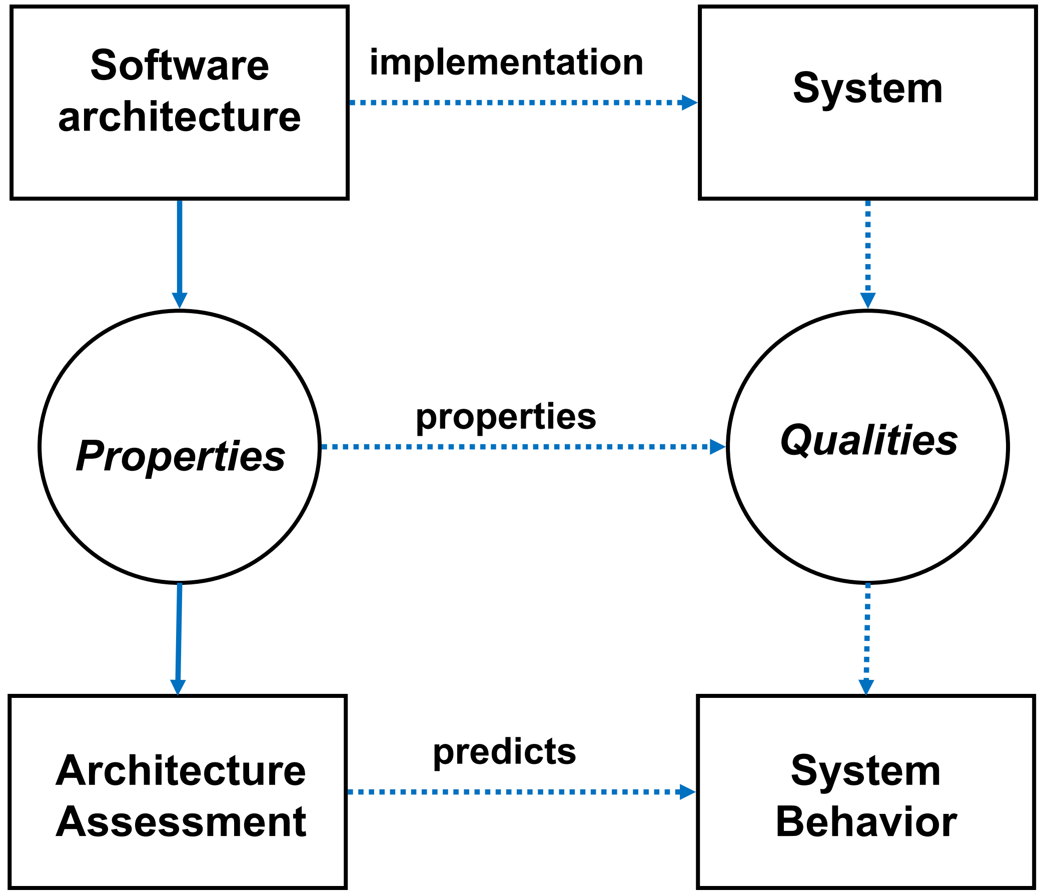
- What is it, why bother?
- Architecture Design
- Viewpoints and view models
- Architectural styles
- Architecture asssessment
- Role of the software architect

Architecture evaluation/analysis

- Assess whether architecture meets certain quality goals
  - maintainability
  - modifiability
  - reliability
  - performance

- Mind
  - architecture is assessed
  - we hope the results will hold for a system yet to be built

Software Architecture Analysis



Analysis techniques

- Questioning techniques
  - how does the system react to various situations?
  - often make use of scenarios
- Measuring techniques
  - rely on quantitative measures
  - architecture metrics
  - simulation
  - etc.

Scenarios in Architecture Analysis

- Different types of scenarios
  - e.g. use-cases
  - likely changes
  - stress situations
  - risks
  - far-into-the-future scenarios
- Which stakeholders to ask for scenarios?
- When do you have enough scenarios?

Preconditions for successful assessment

- Clear goals and requirements for the architecture
- Controlled scope
- Cost-effectiveness
- Key personnel availability
- Competent evaluation team
- Managed expectations

Architecture Tradeoff Analysis Method

ATAM

- Reveals how well architecture satisfies quality goals
- How well quality attributes interact
  - i.e. how they trade off
- Elicits business goals for system and its architecture
- Uses those goals and stakeholder participation to focus attention to key portions of the architecture

Benefits

- Financial gains
- Forced preparation
- Captured rationale
- Early detection of problems
- Validation of requirements
- Improved architecture

Participants in ATAM

- Evaluation team
- Decision makers
- Architecture stakeholders

Phases in ATAM

- 0: partnership, preparation
  (informally)

- 1: evaluation
  (evaluation team + decision makers, one day)

- 2: evaluation
  (evaluation team + decision makers + stakeholders, two days)

- 3: follow up
  (evaluation team + client)

Steps in ATAM (phases 1 and 2)

- Present method
- Present business drivers (by project manager of system)
- Present architecture (by lead architect)
- Identify architectural approaches/styles
- Generate quality attribute utility tree
  - + priority
  - how difficult
- Analyze architectural approaches

- Brainstorm and prioritize scenarios
- Analyze architectural approaches
- Present results

Example Utility tree

- Utility
  - Performance
    - Transaction response time (H, M)
    - Throughput — 150 transactions/sec
  - Usability
    - Training
    - Normal operations
  - Maintainability — vendor releases new Database version

Outputs of ATAM

- Concise presentation of the architecture
- Articulation of business goals
- Quality requirements expressed as set of scenarios
- Mapping of architectural decisions to quality requirements
- Set of sensitivity points and tradeoff points
- Set of risks, nonrisks, risk themes

Important concepts in ATAM

- Sensitivity point
  - decision/property that is critical for certain quality attribute
- Tradeoff point
  - decision/property that affects more than one quality attribute
- Risk
  - decision/property that is a potential problem

- These concepts overlap

Overview

- What is it, why bother?
- Architecture Design
- Viewpoints and view models
- Architectural styles
- Architecture asssessment

- Role of the software architect

Role of the software architect

- Key technical consultant
- *Make decisions*
- Coach of development team
- Coordinate design
- Implement key parts
- Advocate software architecture

Summary

- New and immature field
- Proliferation of terms: architecture - design pattern - framework - idiom
- Architectural styles and design pattern
- *describe* (how things are done)
- *prescribe* (how things should be done)
- stakeholder communication
- *early* evaluation of a design
- transferable abstraction