Introduction to Software Architecture

Imed Hammouda

Chalmers | University of Gothenburg

Who am I?

- Associate Professor of Software Engineering, previously in Tampere, Finland

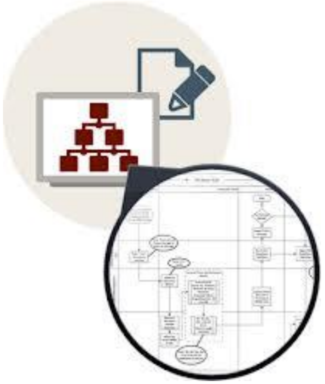
- Research interests
  - Software Architecture, Open Source, Software Ecosystems, Software Development Methods and Tools, Variability Management
    - Developing and supporting open software architectures
    - Studying socio-technical dependencies in software development
    - Software ecosystems

- Coordinates:

  Imed.hammouda@cse.gu.se, hammouda@chalmers.se
    - Room 416, floor 4, Jupiter building, Campus Lindholmen
    - Phone +46 31 772 60 40

CONTENTS :

- What is software architecture?
- Architectural drivers
- Addressing architectural drivers
- Architectural views
- Example system



What is Software Architecture?

- Software Architecture is the **global organization** of a software system, including
  - the division of software into **subsystems/components**,
  - **policies** according to which these subsystems **interact**,
  - the definition of their **interfaces**.

T. C. Lethbridge & R. Laganière

What is Software Architecture?

- "The software architecture of a program or computing system is the structure or **structures** of the system, which comprise software **components**, the **externally visible properties** of those components, and the **relationships** among them."

Len Bass

What is Software Architecture?

- “fundamental **concepts** or **properties** of a system in its **environment** embodied in its **elements**, **relationships**, and in the **principles** of its **design** and **evolution**.”

ISO/IEC/IEEE 42010
http://www.iso-architecture.org/ieee-1471/defining-architecture.html

Architectural Information Increases

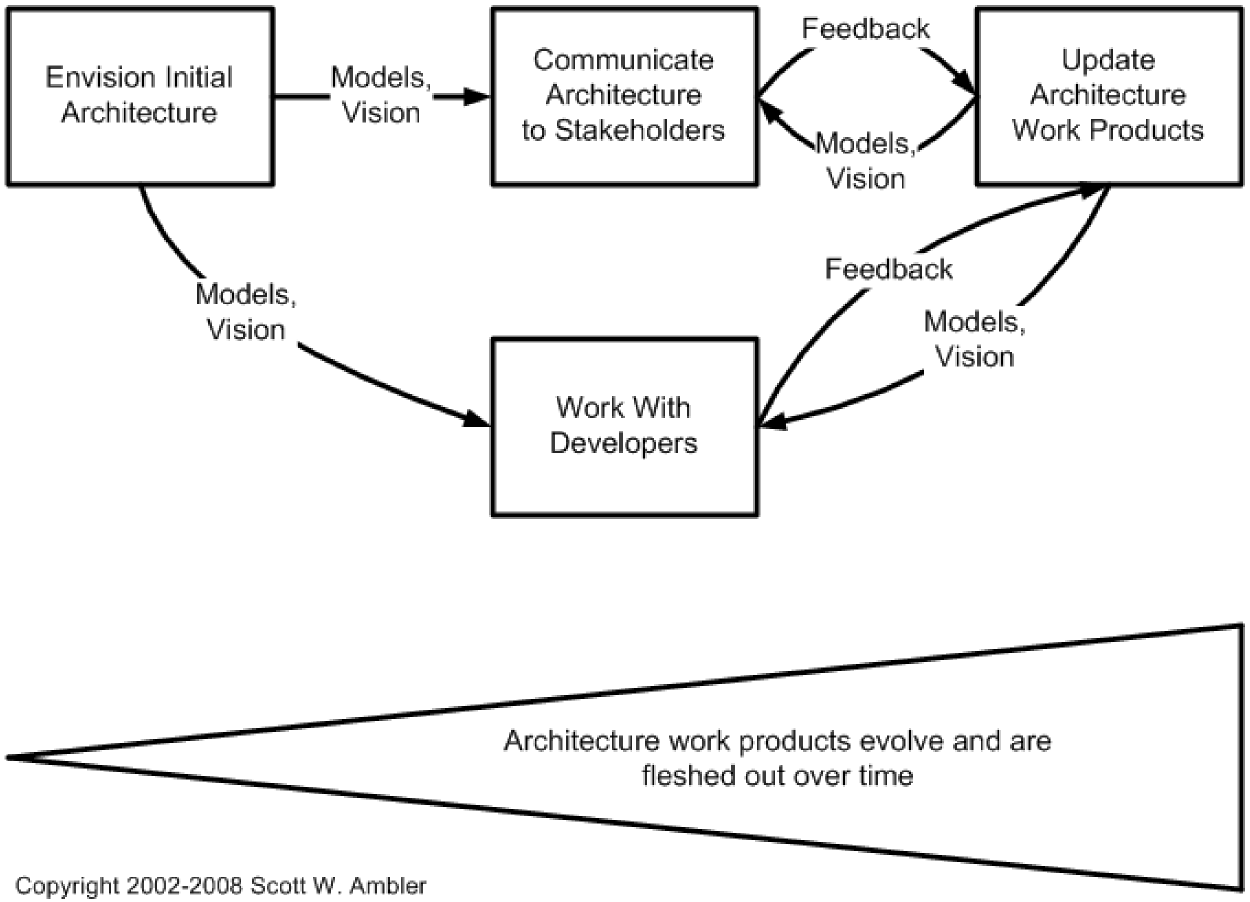
Architecture is **everything** that a (group of) **person(s)** needs to let a large **team** successfully develop a (family of) **products**

Rob van Ommering, Philips Natlab

- Add your own definition:
  http://www.sei.cmu.edu/architecture/start/glossary/community.cfm

The Role of an Architect

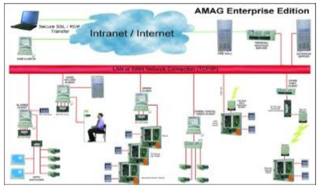
- Central activities:
  - Design
  - Document
  - Assess
  - Recover
  - Maintain



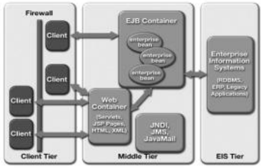
Copyright 2002-2008 Scott W. Ambler

Levels of Architecture*

Enterprise architecture

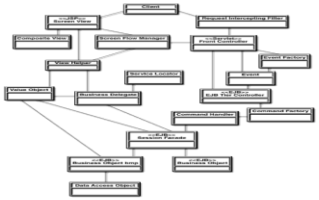



System architecture



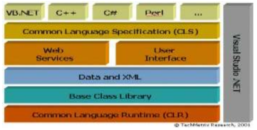
Subsystem

Application architecture



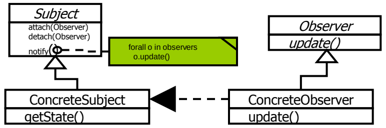
Application

Macro-architecture



Frameworks

Micro-architecture

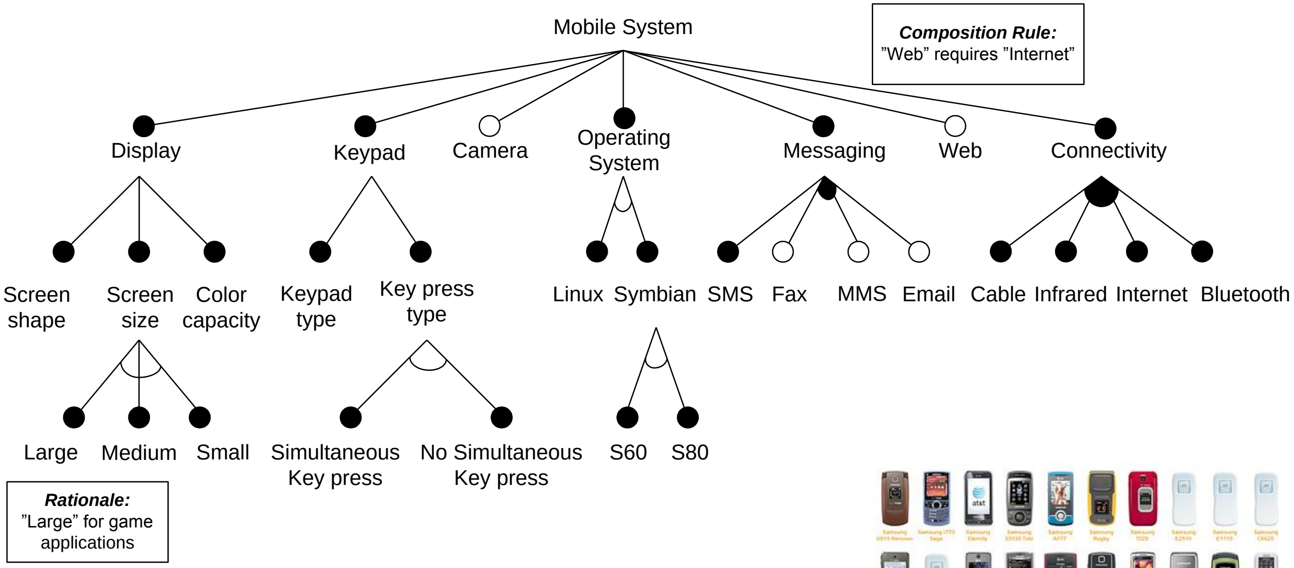


Design patterns

* Mowbray and Malveau

Types of Architecture

Single Product



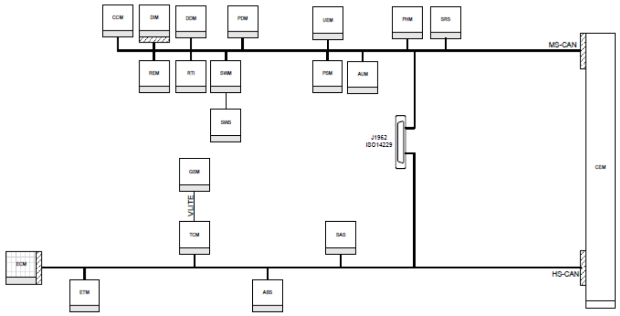
Composition Rule:
”Web” requires ”Internet”

Rationale:
”Large” for game applications

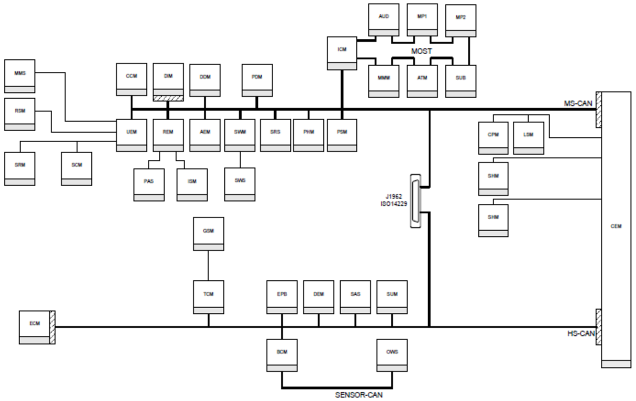
Product family

Increasing Amount of Software in Systems

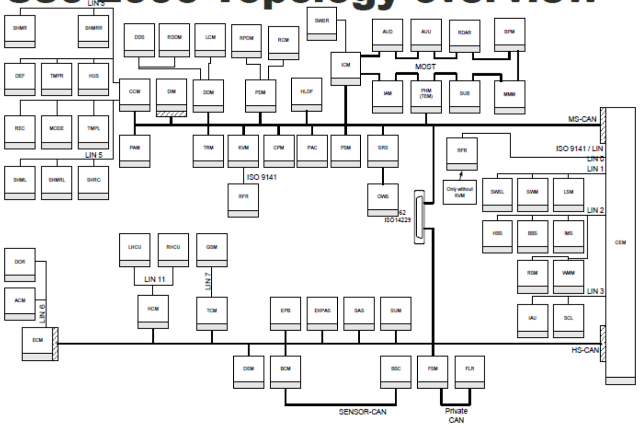
S80 1998 Topology overview



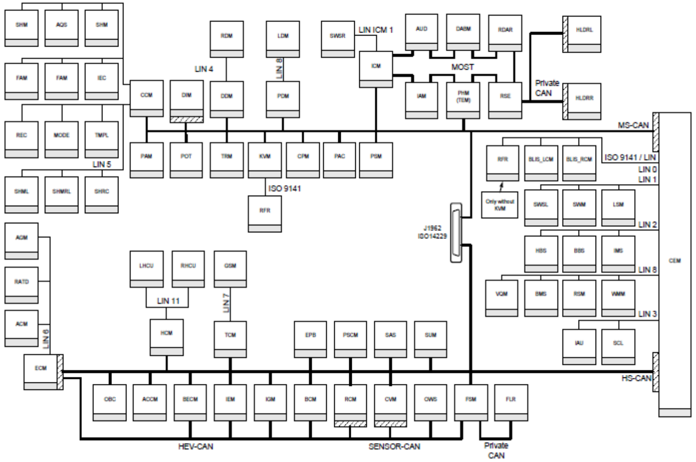
XC90 2002 Topology overview



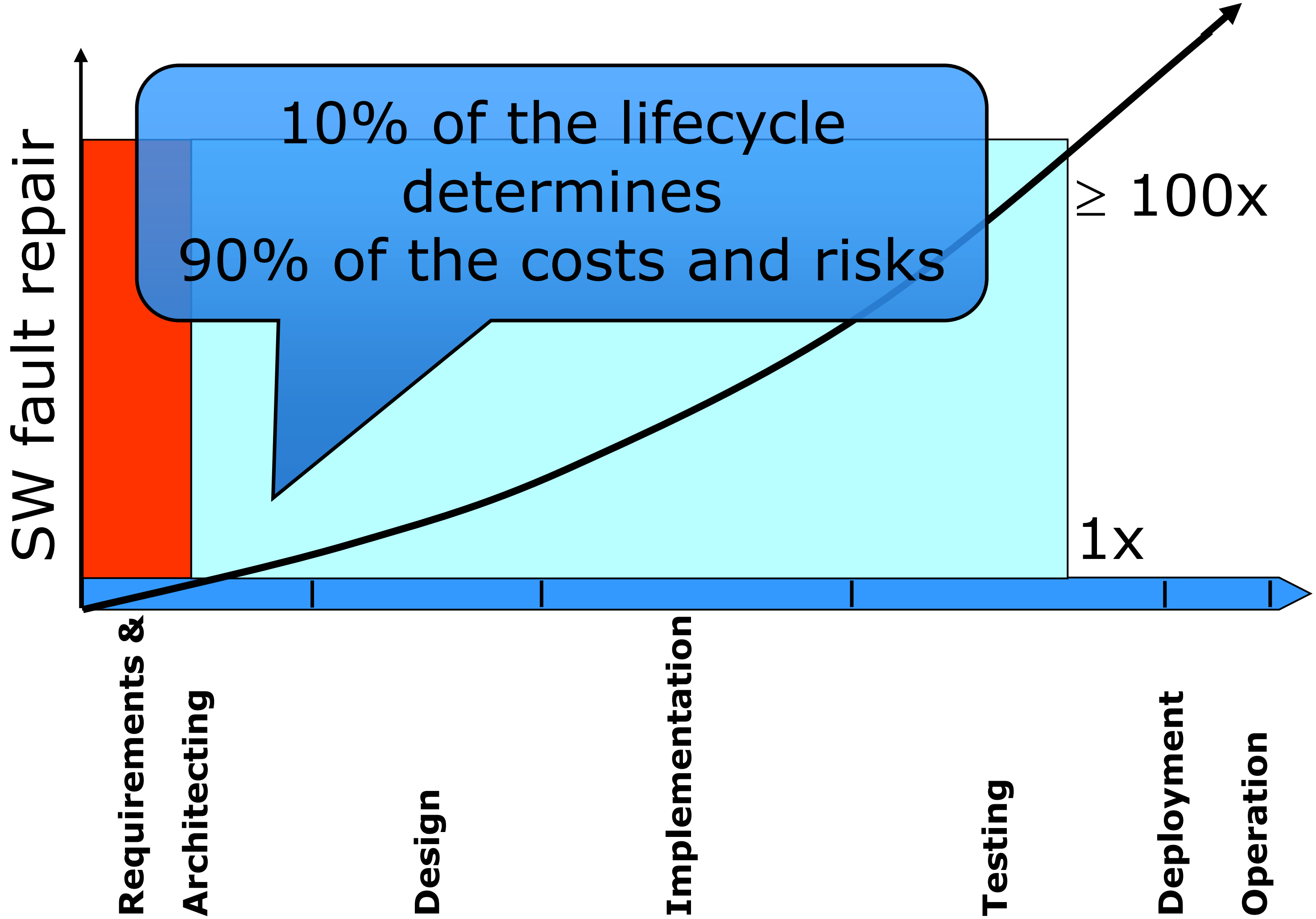
S80 2006 Topology overview



V60 PHEV Topology overview



The Importance of Architecture



10% of the lifecycle
determines
90% of the costs and risks

The Importance of Architecture

- “If a project has not achieved a system architecture, including its rationale, the **project should not proceed** to full-scale system development. Specifying the architecture as a deliverable enables its use throughout the **development** and **maintenance** process”

Barry Boehm

WHY?

The Importance of Architecture

- Software architecture:
  - provides a **communication** among stakeholders
  - captures **early design** decisions
  - acts as a **transferable abstraction** of a system
  - defines **constraints** on implementation
  - dictates **organizational** structure
  - inhibits or enables a system’s **quality attributes**
  - is analyzable and a vehicle for **predicting** system qualities
  - makes it easier to reason about and **manage change**
  - helps in evolutionary **prototyping**
  - enables more accurate **cost** and **schedule** estimates

WHY?

What Drives Software Architecture?

- System requirements:
  - **Functional needs** (what the system should do)
  - **Quality needs** (properties that the system must possess such as availability, performance, security,…
- **Design constraints**
  - Development process
  - Technical constraints
  - Business constraints
  - Contractual requirements
  - Legal obligations
  - Economic factors

What Drives Software Architecture?*

Plain Old Telephone System

- Feature:
  - Call subscriber
- Good qualities:
  - Works during power shortage
  - Reliable
  - Emergency calls get location information
- Architecture:

Centralized hardware switch

Skype

- Feature:
  - Call subscriber
- Good qualities:
  - Scales without central hardware changes
  - Easy to add new features
  - Easy deployment
- Architecture:

Peer-to-peer software

*George Fairbanks

Architectural Drivers

- **Architectural drivers** are the design **forces** that will influence the early design decisions the architects make

- Architectural drivers are not all of the requirements for a system, but they are an early attempt to identify and capture those requirements, that are **most influential** to the architect making early design decisions.

→ **Architecturally Significant Requirements**

Archietcts pay more attention to **qualities** that arrise from architecture choices

Functional Requirements

- **Functional requirements** specify what the software needs to do. They relate to the **actions** that the product must carry out in order to satisfy the fundamental reasons for its existence.

- **Business Level**: defines the objective/goal of the project and the measurable business benefits for doing it.

- **User Level**: user requirements are written from the user's point-of-view.

- **System Level**: defines what the system must do to process input and provide the desired output.

Functional Requirements

- **MoSCoW** Method:
  - **M - MUST**: Describes a requirement that must be satisfied in the final solution for the solution to be considered a success.
  - **S - SHOULD**: Represents a high-priority item that should be included in the solution if it is possible. This is often a critical requirement but one which can be satisfied in other ways if strictly necessary.
  - **C - COULD**: Describes a requirement which is considered desirable but not necessary. This will be included if time and resources permit.
  - **W - WON'T**: Represents a requirement that stakeholders have agreed will not be implemented in a given release, but may be considered for the future.

Functionality and Software Architecture

- It is the ability of the system or application to **satisfy the purpose** for which it was designed.
- It drives the initial **decomposition** of the system.
- It is the **basis** upon which all other quality attributes are specified.
- It is **related to quality attributes** like validity, correctness, interoperability, and security.

Functional requirements often get the most focus in a development project. But systems are often redesigned, **not** because of functional requirements.

Quality Attributes

- A **quality attribute** is a **measurable** or **testable property** of a system that is used to indicate **how well** the system satisfies the needs of its stakeholders.*

- A **quality requirement** is a specification of the **acceptable values** of a quality attribute that must be present in the system.

- Quality attributes should be:
  - Not subjective
  - In sufficient detail
  - Of a value and context (e.g: 180 seconds)

*Bass Clements Kazman

Quality Attributes*

Attribute	Description
Performance	How fast does it respond or execute?
Availability	Is it available when an where I need to use it?
Safety	How well does it protect against damage?
Usability	How easy it is for people to learn and use?
Interoperability	How easily does it interconnect with other systems?
Integrity	Does it protect against unauthorized access and data loss?
Installability	How easy is it to correctly install the product?
Robustness	How well does it respond to unexpected operating conditions?
Reliability	How long does it run before experiencing a failure?
Recoverability	How quickly can the user recover from a failure?

*EnfocusSolutions

Quality Attributes

Attribute	Description
Recoverability	How quickly can the user recover from a failure?
Efficiency	How well does it utilize processor capacity, disk space, memory, bandwidth, and other resources?
Flexibility	How easily can it be updated with new functionality?
Maintainability	How easy is it to correct defects or make changes?
Portability	How easily can it be made to work on other platforms?
Reusability	How easily can we use components in other systems?
Scalability	How easily can I add more users, servers, or other extensions?
Supportability	How easy will it be support after installation?
Testability	Can I verify that it eas implemented correctly?

ISO/IEC 25010:2011 Quality Model

Functional suitability	**Compatibility**	**Reliability**	**Maintainability**
Functional completeness Functional correctness Functional appropriateness	Co-existence Interoperability	Maturity Availability Fault tolerance Recoverability	Modularity Reusability Analysability Modifiability Testability
Performance efficiency	**Usability**	**Security**	**Portability**
Time behaviour Resource utilization Capacity	Appropriateness recognizability Learnability Operability User error protection User interface aesthetics Accessibility	Confidentiality Integrity Non-repudiation Accountability Authenticity	Adaptability Installability Replaceability

ISO/IEC 25010:2011 Quality in Use Characteristics

Effectiveness	**Efficiency**
Satisfaction Usefulness Trust Pleasure Comfort	**Freedom from risk** Economic risk mitigation Health and safety risk mitigation Environmental risk mitigation
Context coverage Context completeness Flexibility	

Quality Attributes & Software Architecture

- Different architectural styles address **different sets of quality** attributes and to varying degrees
  - **Architecture decides** range of quality possibilities
- The specification of quality attributes affects the architectural style of the system
  - **Architectures are evaluated** w.r.t quality attributes
- **Not all quality attributes** are addressed by the architectural design, e.g. some aspects of usability (e.g. layouts) and some aspects of performance (e.g. algorithms)
- **Impossible to maximize** all quality attributes at once
  - Tradeoff: More of this → less of that
  - Performance versus security
  - Everything versus time-to-market (or cost)

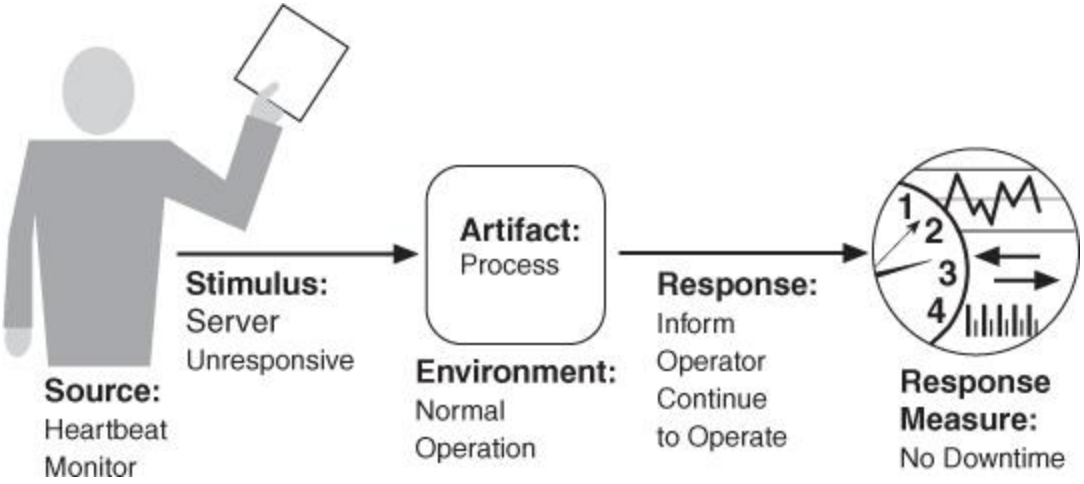
Specifying Quality Requirements

- A Quality Attribute Scenario is a quality attribute specific requirement.
  - **Stimulus** – a condition that needs to be considered
  - **Source** of stimulus (e.g., human, computer system, etc.)
  - **Environment** - what are the conditions when the stimulus occurs?
  - **Artifact** – what elements of the system are stimulated.
  - **Response** – the activity undertaken after arrival of the stimulus.
  - **Response measure** – when the response occurs it should be measurable so that the requirement can be tested.

Specifying Quality Requirements

Reliability	
Stimulus	Database unresponsive/crash
Source of Stimulus	System
Environment	Runtime
Artifact	Database
Response	Detect the failure, inform the system, use redundant database server
Response Measure	Degraded mode not to exceed 5 mins

Availability Concrete Scenario



Source: Heartbeat Monitor

Stimulus: Server Unresponsive

Artifact: Process

Environment: Normal Operation

Response: Inform Operator Continue to Operate

Response Measure: No Downtime

Making Design Decisions

- To make each design decision, the software engineer uses knowledge of:
  - the application domain
  - the requirements
  - the design as created so far
  - the technology available
  - software design principles and 'best practices'
  - what has worked well in the past

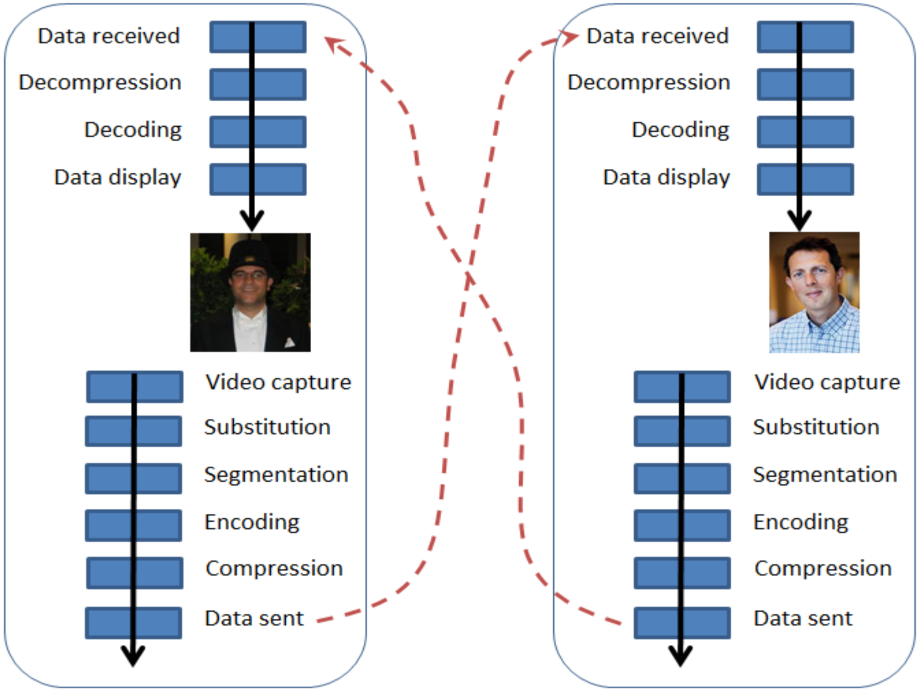
"The architecture = f(requirements, design methods, experience, knowledge, patterns, intuition, ...)"

Making Design Decisions

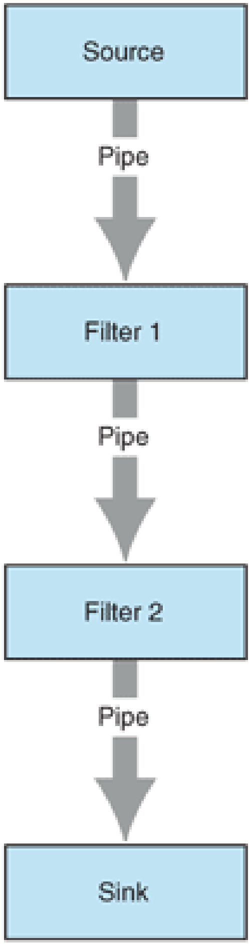
- **Quality attributes** can be addressed through different (architectural) tactics:
  - **Architectural styles**: for example pipes-and-filters, MVC, blackboard, publish-subscribe,…
  - **Design patterns**: for example Abstract Factory, Adapter, Command, Observer,…
  - **Tactics**: A design decision: for example heartbeat, limit access
  - **Converting** quality requirement to functionality: for example exception handling, logging

- Quality requirements can **be distributed** at the system level to the subsystems or components that make up the system.

Example Architectural Style

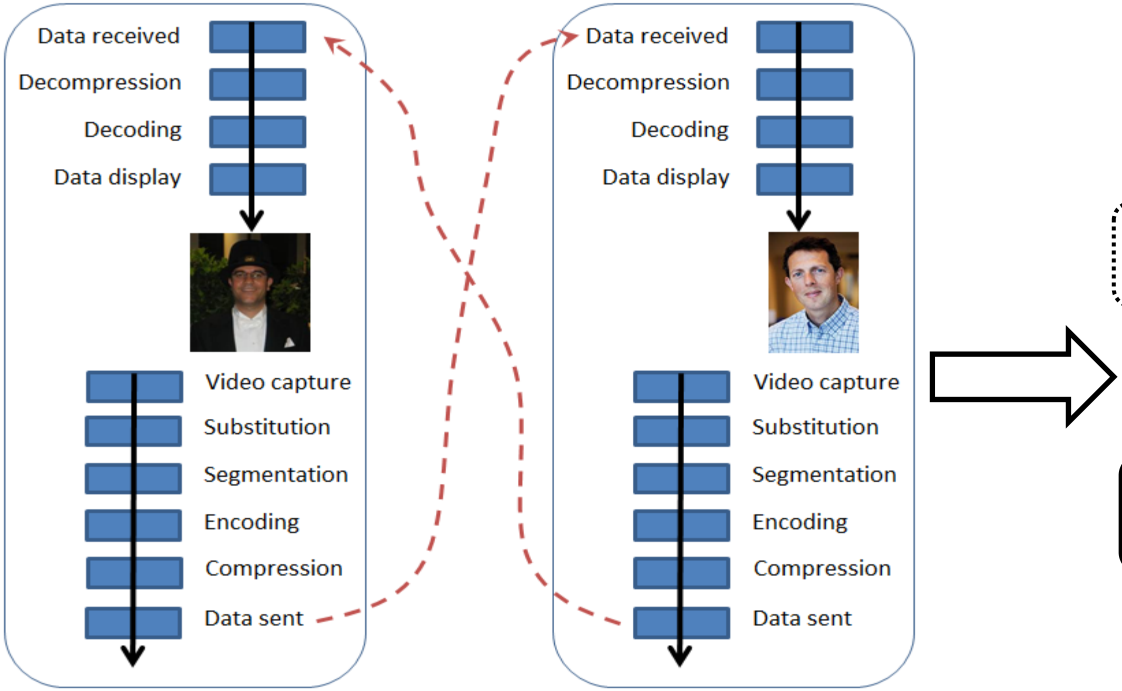


Remote Interaction System

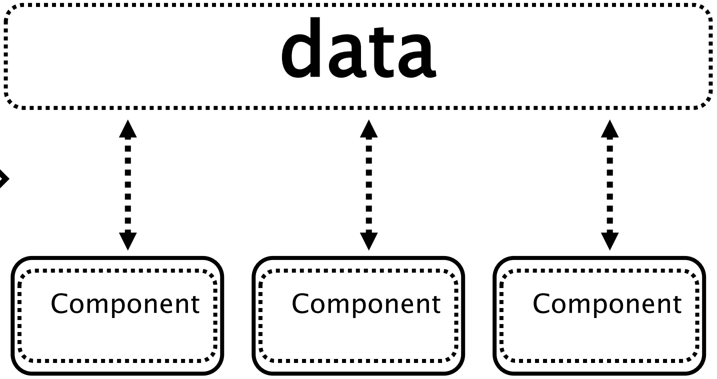


Pipes and Filters

Or it could have been…

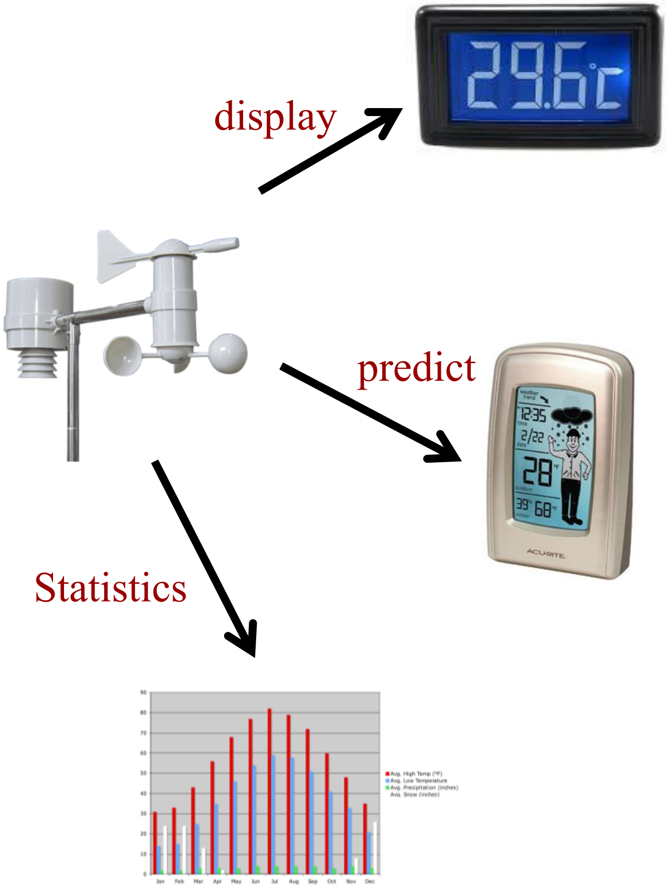


Remote Interaction System



Blackboard

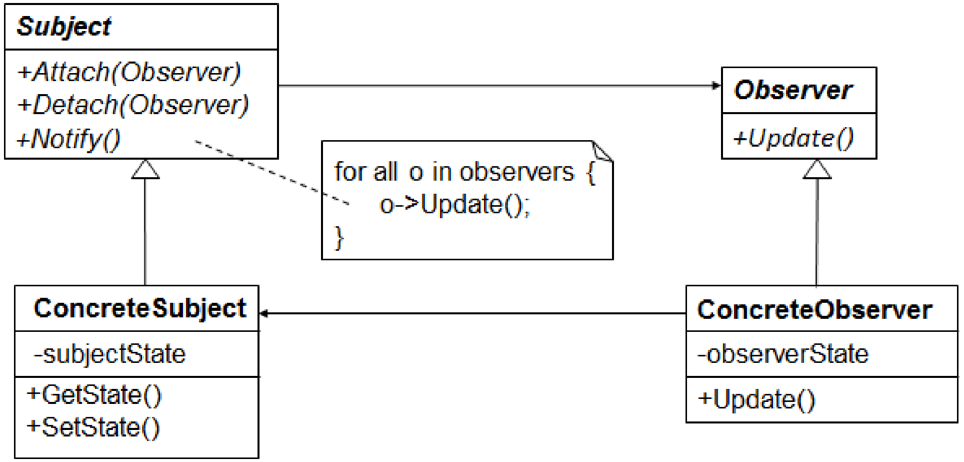
Example Design Pattern



display

predict

Statistics



Subject

+Attach(Observer)
+Detach(Observer)
+Notify()

for all o in observers {
o->Update();
}

Observer

+Update()

ConcreteSubject

-subjectState

+GetState()
+SetState()

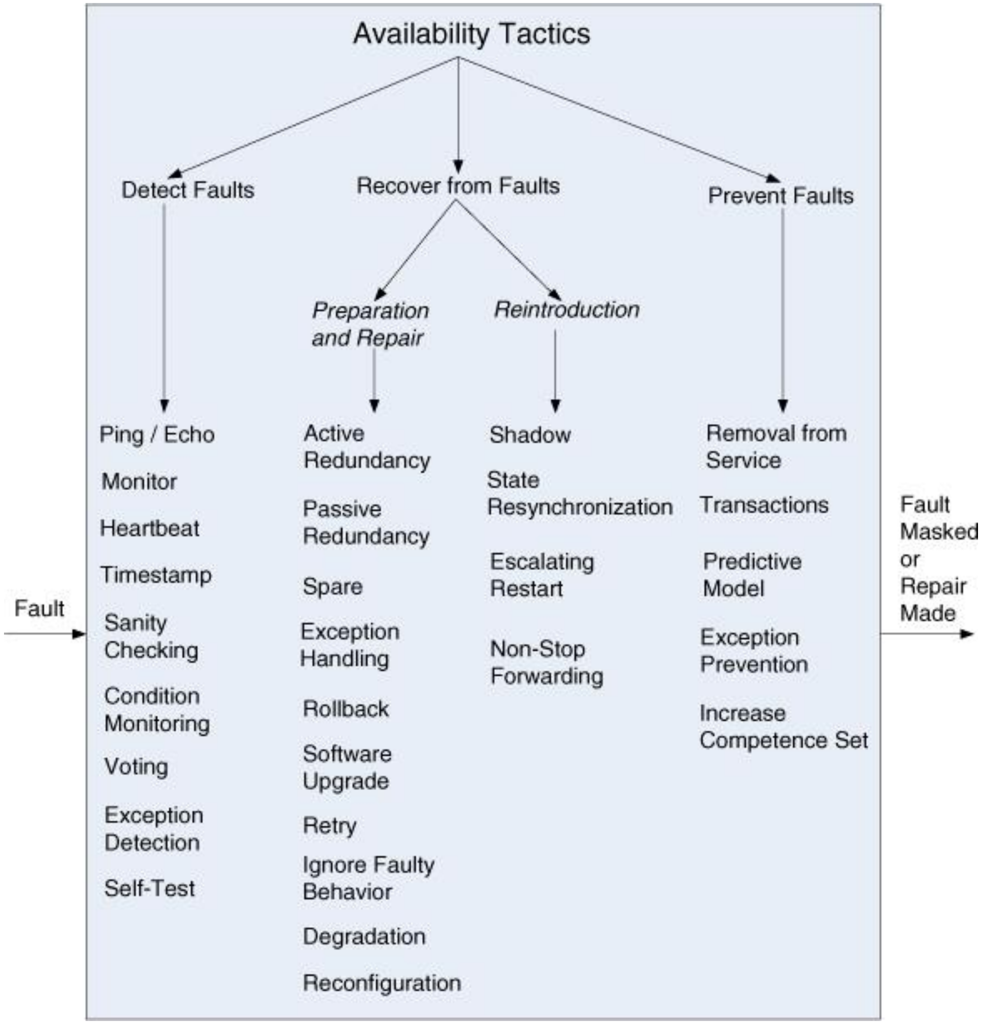
ConcreteObserver

-observerState

+Update()

Observer

Example Architectural Tactics



Availability Tactics

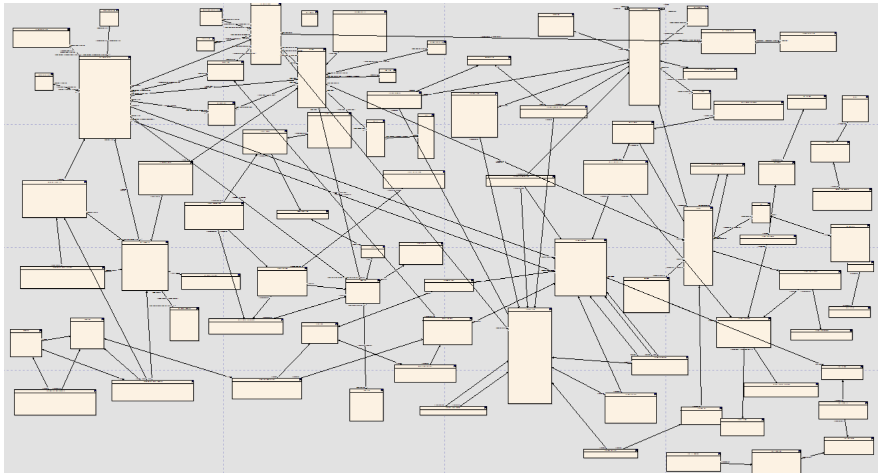
- Detect Faults
  - Ping / Echo
  - Monitor
  - Heartbeat
  - Timestamp
  - Sanity Checking
  - Condition Monitoring
  - Voting
  - Exception Detection
  - Self-Test
- Recover from Faults
  - *Preparation and Repair*
    - Active Redundancy
    - Passive Redundancy
    - Spare
    - Exception Handling
    - Rollback
    - Software Upgrade
    - Retry
    - Ignore Faulty Behavior
    - Degradation
    - Reconfiguration
  - *Reintroduction*
    - Shadow
    - State Resynchronization
    - Escalating Restart
    - Non-Stop Forwarding
- Prevent Faults
  - Removal from Service
  - Transactions
  - Predictive Model
  - Exception Prevention
  - Increase Competence Set

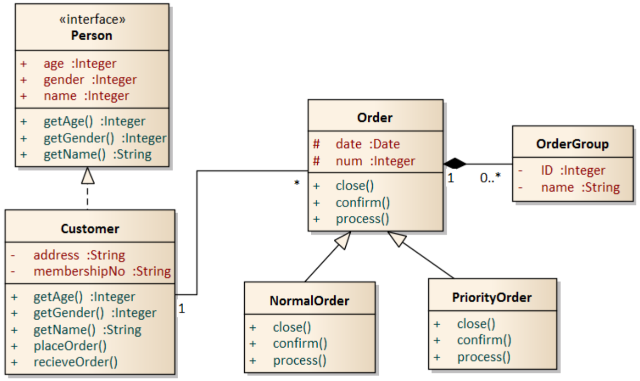
Fault

Fault Masked or Repair Made

Design Principles: Keep it Simple (KIS)

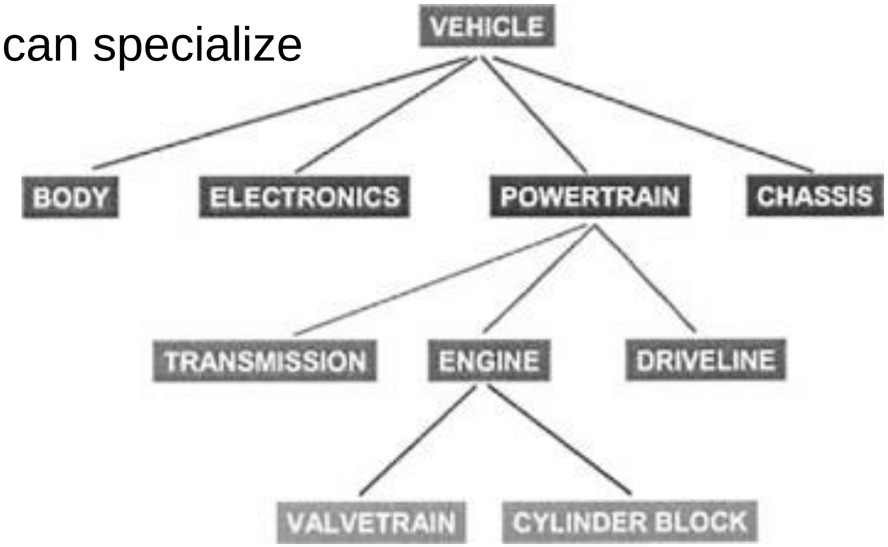
- Simplicity is a great virtue but it requires hard work to achieve it
  and education to appreciate it.
- And to make matters worse: complexity sells better.





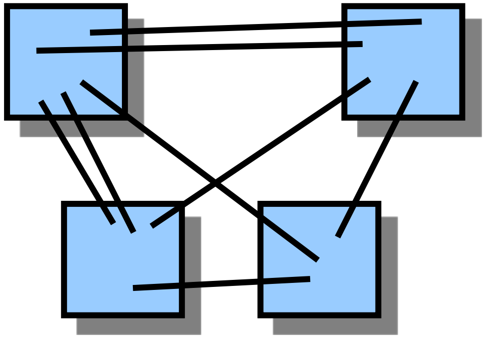
Design Principles: Decomposition

- Breaking problem into independent smaller parts
  - Each individual component is smaller, and therefore easier to understand
  - Parts can be replaced or changed without having to replace or extensively change other parts.
  - Separate people can work on separate parts
  - An individual software engineer can specialize

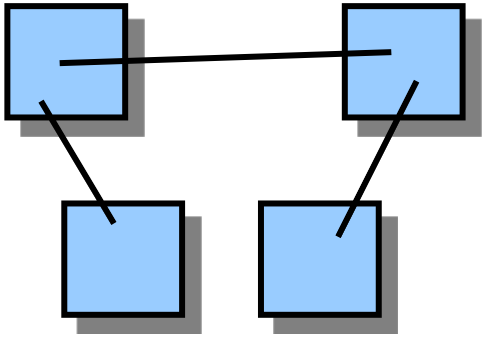


Design Principles: Coupling

- Coupling is a measure of interdependency between modules.



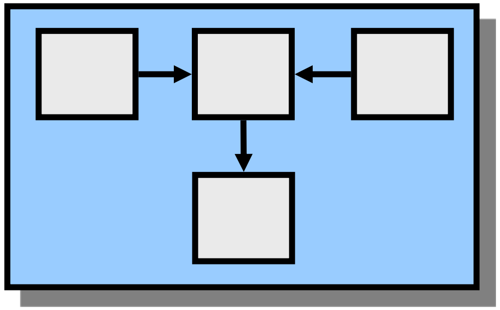
high coupling



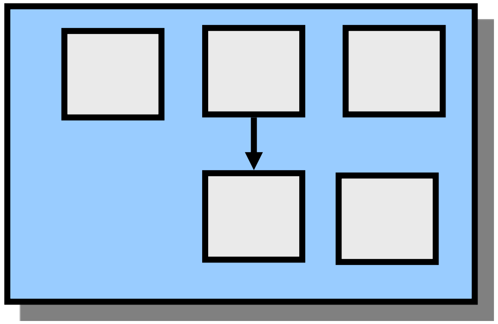
low coupling

Design Principles: Cohesion

- Cohesion is concerned with the relatedness within a module.



high cohesion

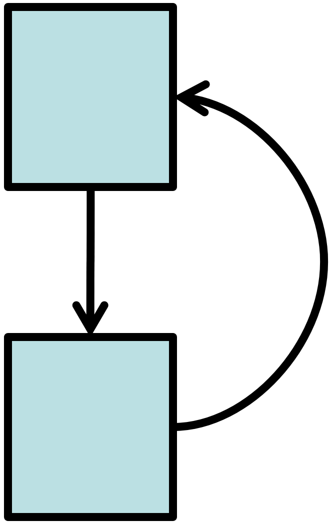


low cohesion

Design Principles: Information Hiding

- What is inside, must stay inside.

Design Principles: No Circular Dependencies



Callers must depend on callee, not vice versa

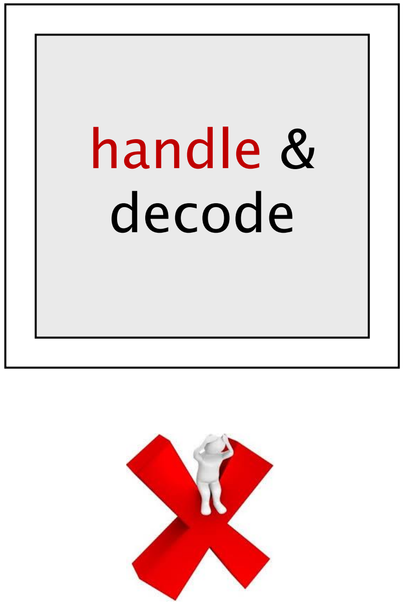
This violates an earlier design advice: Decomposition

Design Principles: Separation of Concerns

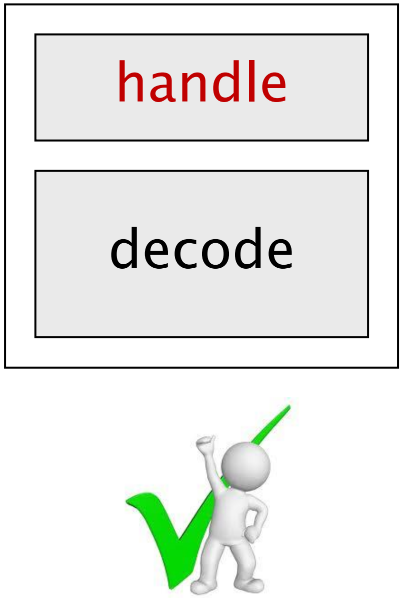
- Issues that are not related should be handled in different components

Telecom protocol:

decode1 ; handle1 ; decode2 ; handle2 ; decode3



handle & decode



handle

decode

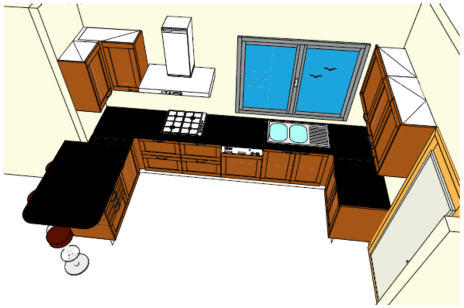
Design Principles: Open/Closed

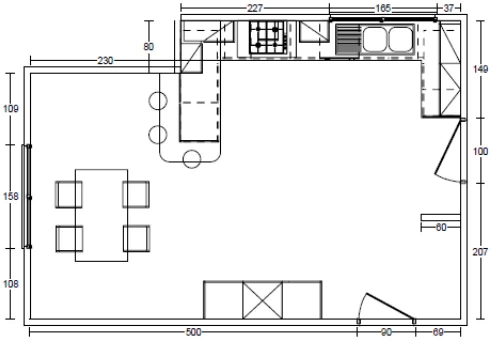
- Entities should be open for extension, but closed for modification.

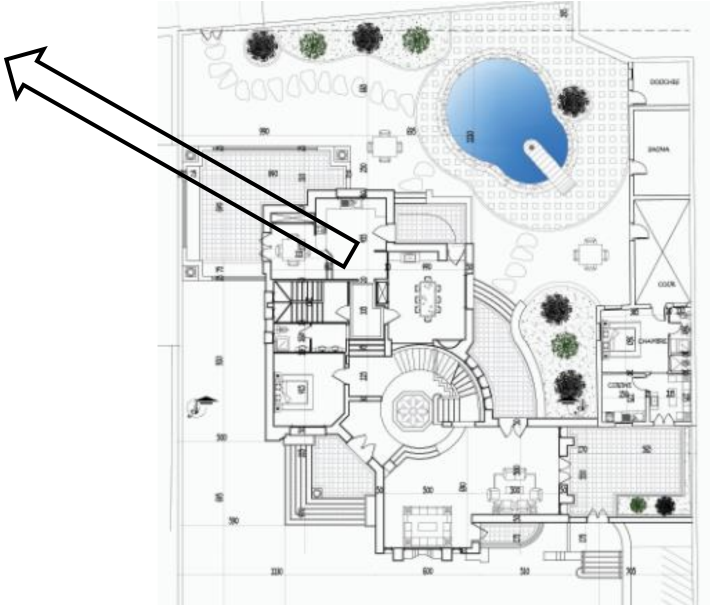
Evaluating Quality Attributes

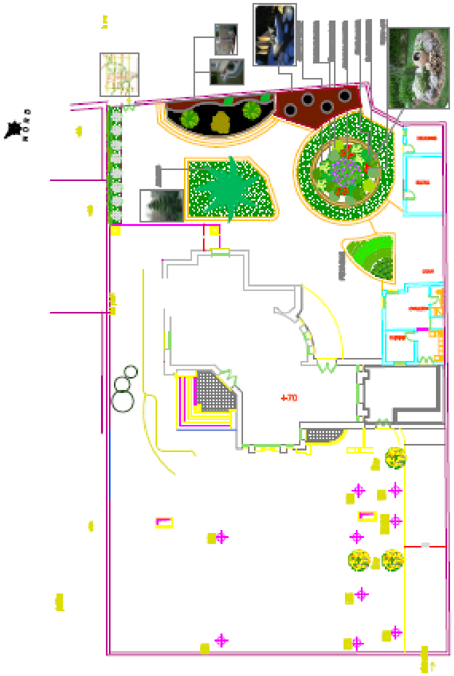
- Quality attributes can be evaluated through:
  - **Scenario-based evaluation**: for example *change scenarios* for assessing maintainability
  - **Simulation**: for example *Prototyping* is a form of simulation where a part of the architecture is implemented and executed in the actual system context.
  - **Mathematical modeling**: for example, checking for *potential deadlocks*.
  - **Experience-based assessment**: this is based on *subjective factors* like intuition and expertise of software engineers.

Views

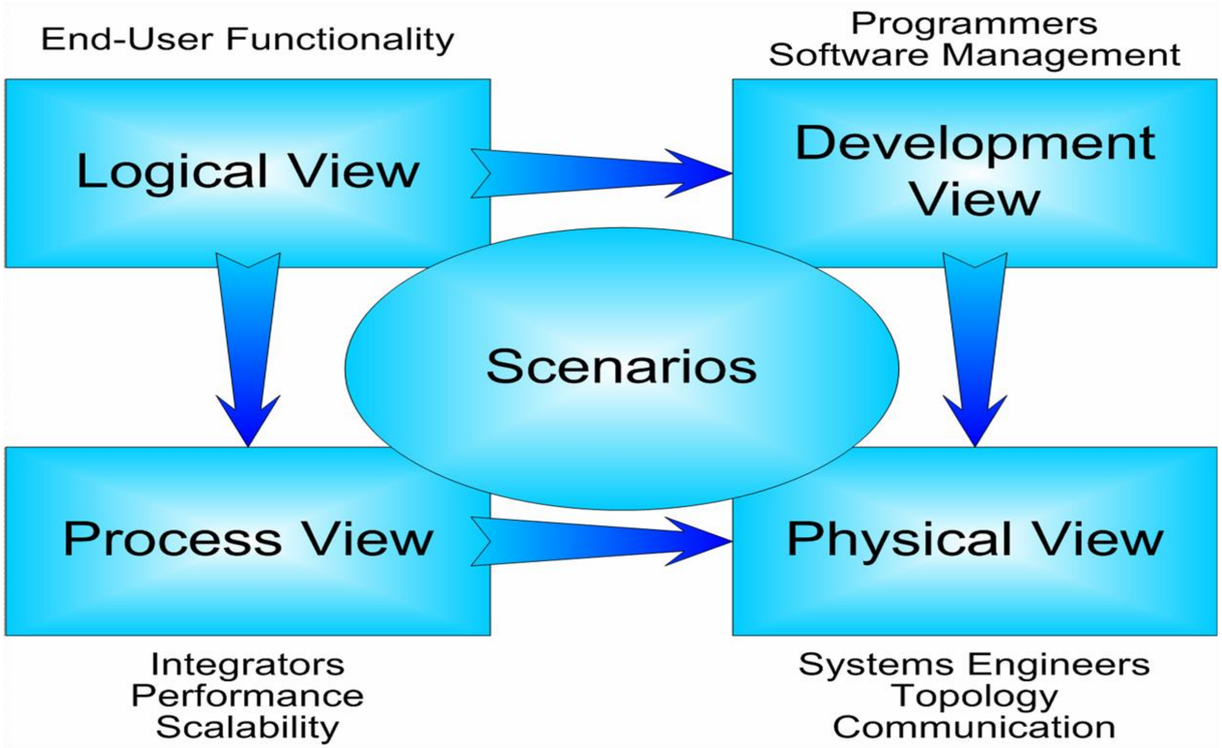








'4 + 1' View Model*



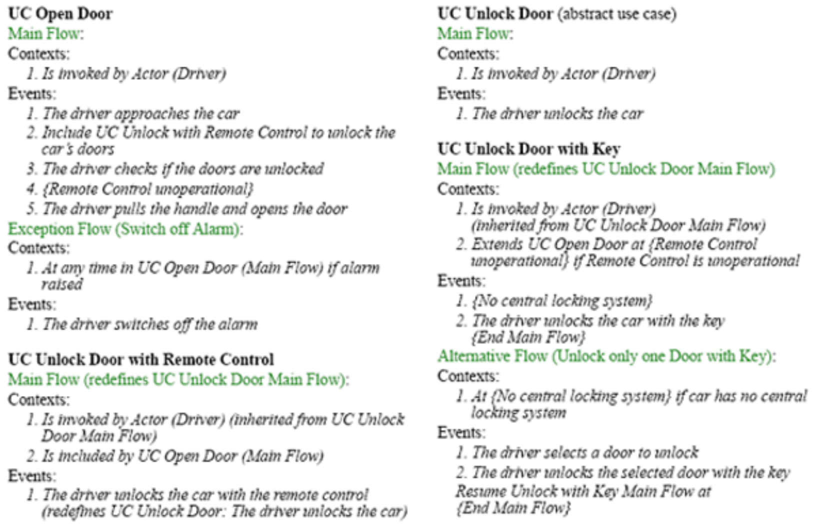
* Philippe Kruchten

’4 + 1’ View Model*

- **Logical**
  - Focus: Functional requirements of the system.
  - Contents: Class diagrams, Sequence diagrams, Layer diagrams.
- **Development** (implementation)
  - Focus: Static organization of the software in its development environment
  - Contents: Component diagram, Package diagrams.
- **Process**
  - Focus: Runtime behavior of the system, such as the system processes and communication, concurrency, performance and scalability.
  - Contents: Activity diagrams.
- **Physical** (Deployment)
  - Focus: System Engineer’s perspective, looking at the system topology, deployment and communication.
  - Contents: Deployment diagrams.
- **Scenarios**
  - Focus: Use cases for illustrating and validating the architecture.
  - Contents: Use case diagrams.

Notations for Arch. Documentation

- **Informal** notations.
  - Views are depicted (often graphically) using general-purpose diagramming and editing tools and visual conventions chosen for the system at hand.
  - The semantics of the description are characterized in **natural language** and **cannot be formally analyzed**.



UC Open Door
Main Flow:
Contexts:
1. Is invoked by Actor (Driver)
Events:
1. The driver approaches the car
2. Include UC Unlock with Remote Control to unlock the car's doors
3. The driver checks if the doors are unlocked
4. {Remote Control unoperational}
5. The driver pulls the handle and opens the door
Exception Flow (Switch off Alarm):
Contexts:
1. At any time in UC Open Door (Main Flow) if alarm raised
Events:
1. The driver switches off the alarm

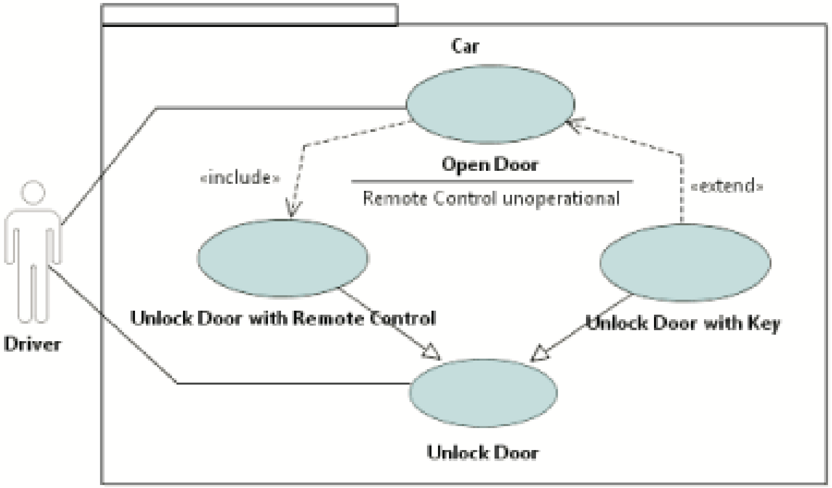
UC Unlock Door with Remote Control
Main Flow (redefines UC Unlock Door Main Flow):
Contexts:
1. Is invoked by Actor (Driver) (inherited from UC Unlock Door Main Flow)
2. Is included by UC Open Door (Main Flow)
Events:
1. The driver unlocks the car with the remote control (redefines UC Unlock Door: The driver unlocks the car)

UC Unlock Door (abstract use case)
Main Flow:
Contexts:
1. Is invoked by Actor (Driver)
Events:
1. The driver unlocks the car

UC Unlock Door with Key
Main Flow (redefines UC Unlock Door Main Flow)
Contexts:
1. Is invoked by Actor (Driver) (inherited from UC Unlock Door Main Flow)
2. Extends UC Open Door at {Remote Control unoperational} if Remote Control is unoperational
Events:
1. {No central locking system}
2. The driver unlocks the car with the key
{End Main Flow}
Alternative Flow (Unlock only one Door with Key):
Contexts:
1. At {No central locking system} if car has no central locking system
Events:
1. The driver selects a door to unlock
2. The driver unlocks the selected door with the key
Resume Unlock with Key Main Flow at
{End Main Flow}

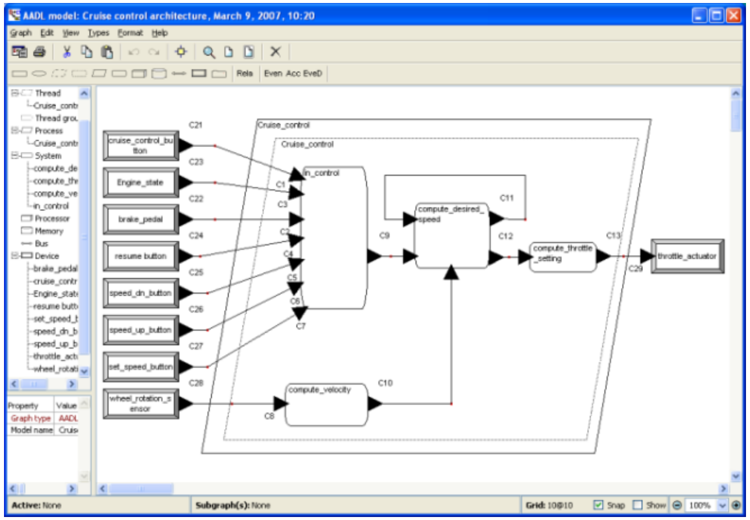
Notations for Arch. Documentation

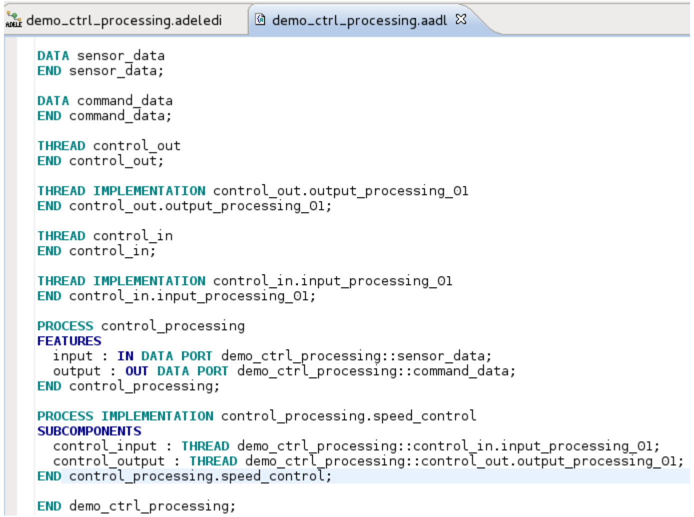
- **Semiformal** notations.
  - Views are expressed in a **standardized notation** that prescribes graphical elements and rules of construction, but does not provide a complete semantic treatment of the meaning of those elements.
  - **Rudimentary analysis** can be applied to determine if a description satisfies syntactic properties. Unified Modeling Language (UML) is a semiformal notation in this sense.



Notations for Arch. Documentation

- **Formal** notations.
  - Views are described in a notation that has a **precise** (usually mathematically based) **semantics**.
  - **Formal analysis** of both syntax and semantics is possible. There are a variety of formal notations for software architecture available, although none of them can be said to be in widespread use.
  - Architecture Description Languages (ADLs)





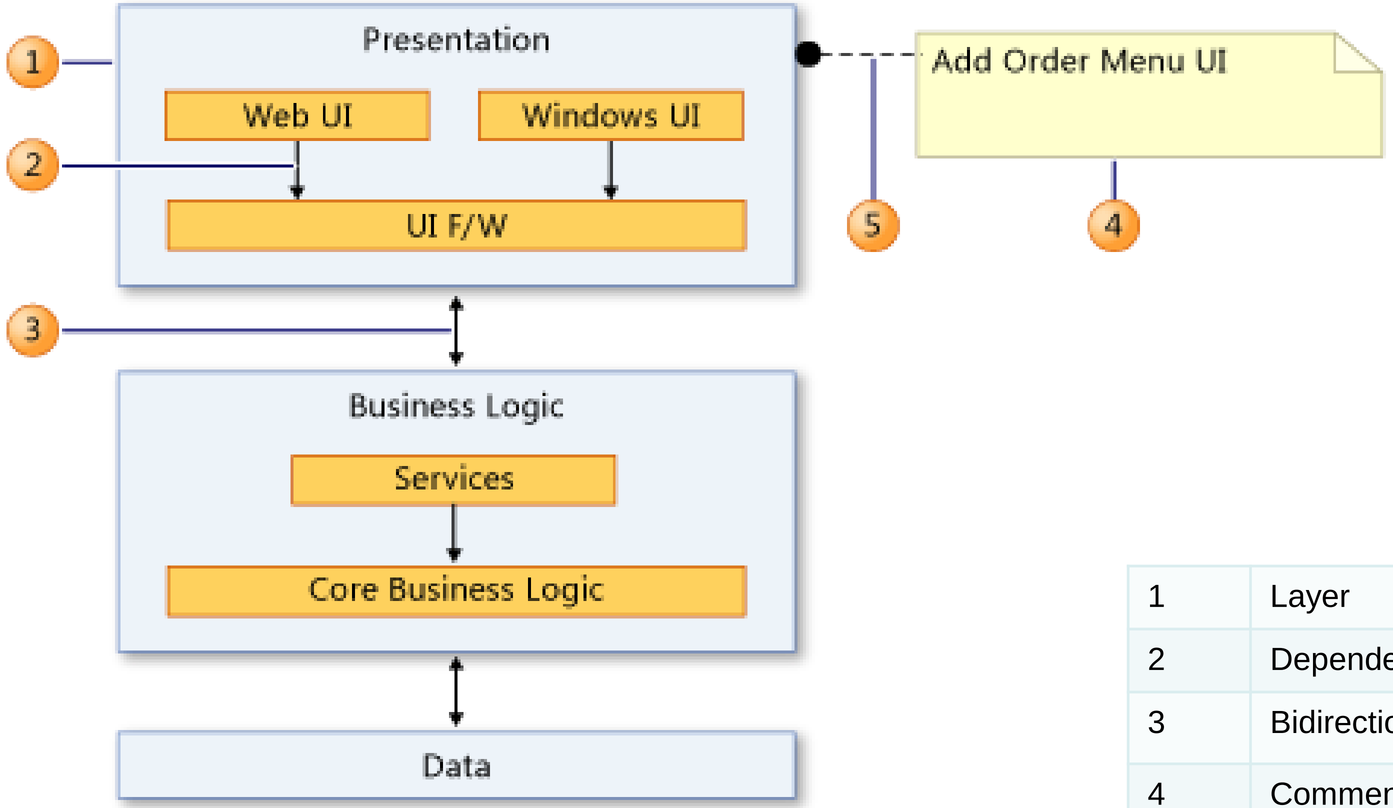
```
DATA sensor_data
END sensor_data;

DATA command_data
END command_data;

THREAD control_out
END control_out;

THREAD IMPLEMENTATION control_out.output_processing_01
END control_out.output_processing_01;

THREAD control_in
END control_in;

THREAD IMPLEMENTATION control_in.input_processing_01
END control_in.input_processing_01;

PROCESS control_processing
FEATURES
  input : IN DATA PORT demo_ctrl_processing::sensor_data;
  output : OUT DATA PORT demo_ctrl_processing::command_data;
END control_processing;

PROCESS IMPLEMENTATION control_processing.speed_control
SUBCOMPONENTS
  control_input : THREAD demo_ctrl_processing::control_in.input_processing_01;
  control_output : THREAD demo_ctrl_processing::control_out.output_processing_01;
END control_processing.speed_control;

END demo_ctrl_processing;
```

Online Catering Service

http://msdn.microsoft.com/en-us/library/dd409427.aspx

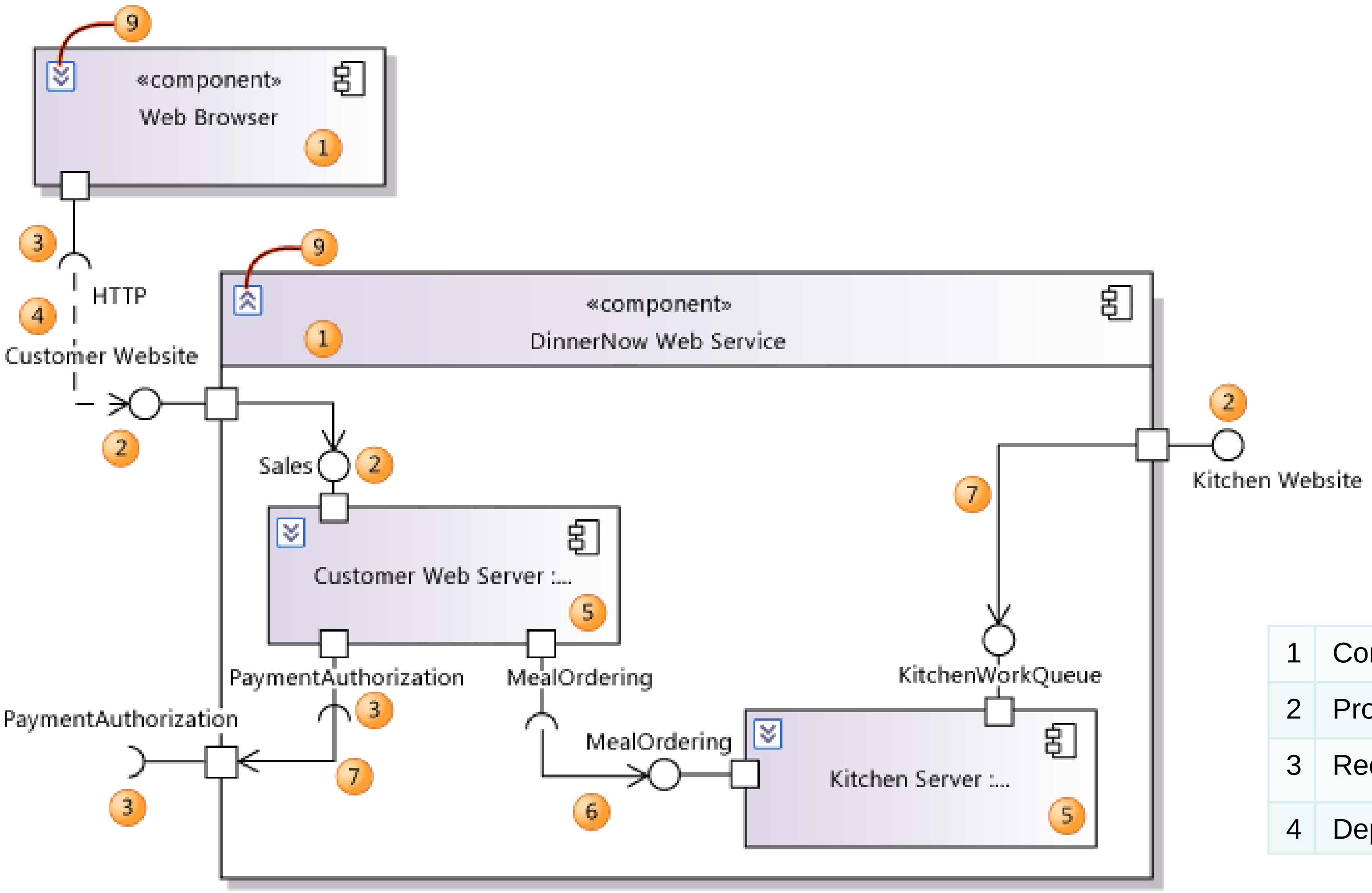
https://www.ibm.com/

Logical View: Layer Diagram



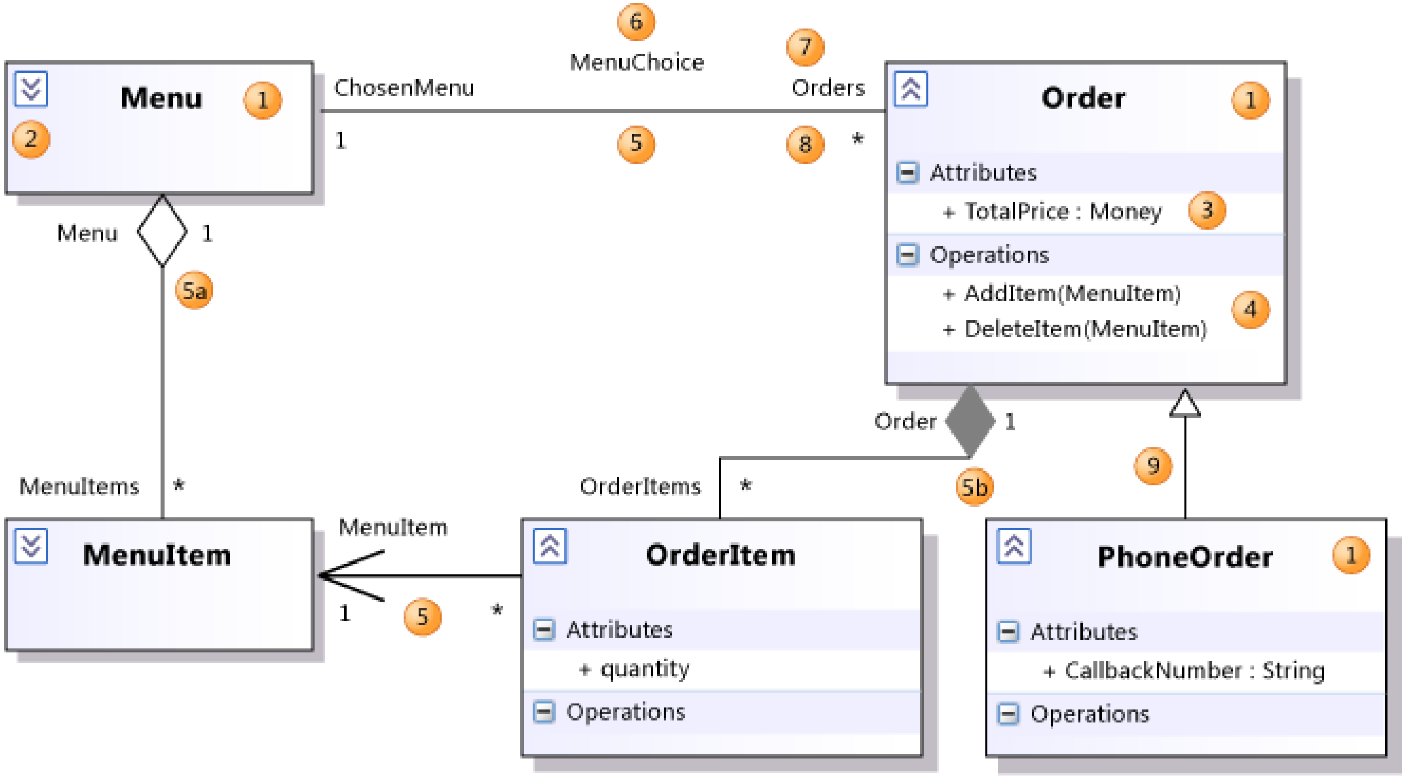
1	Layer
2	Dependency
3	Bidirectional dependency
4	Comment
5	Comment link

Logical View: Component Diagram



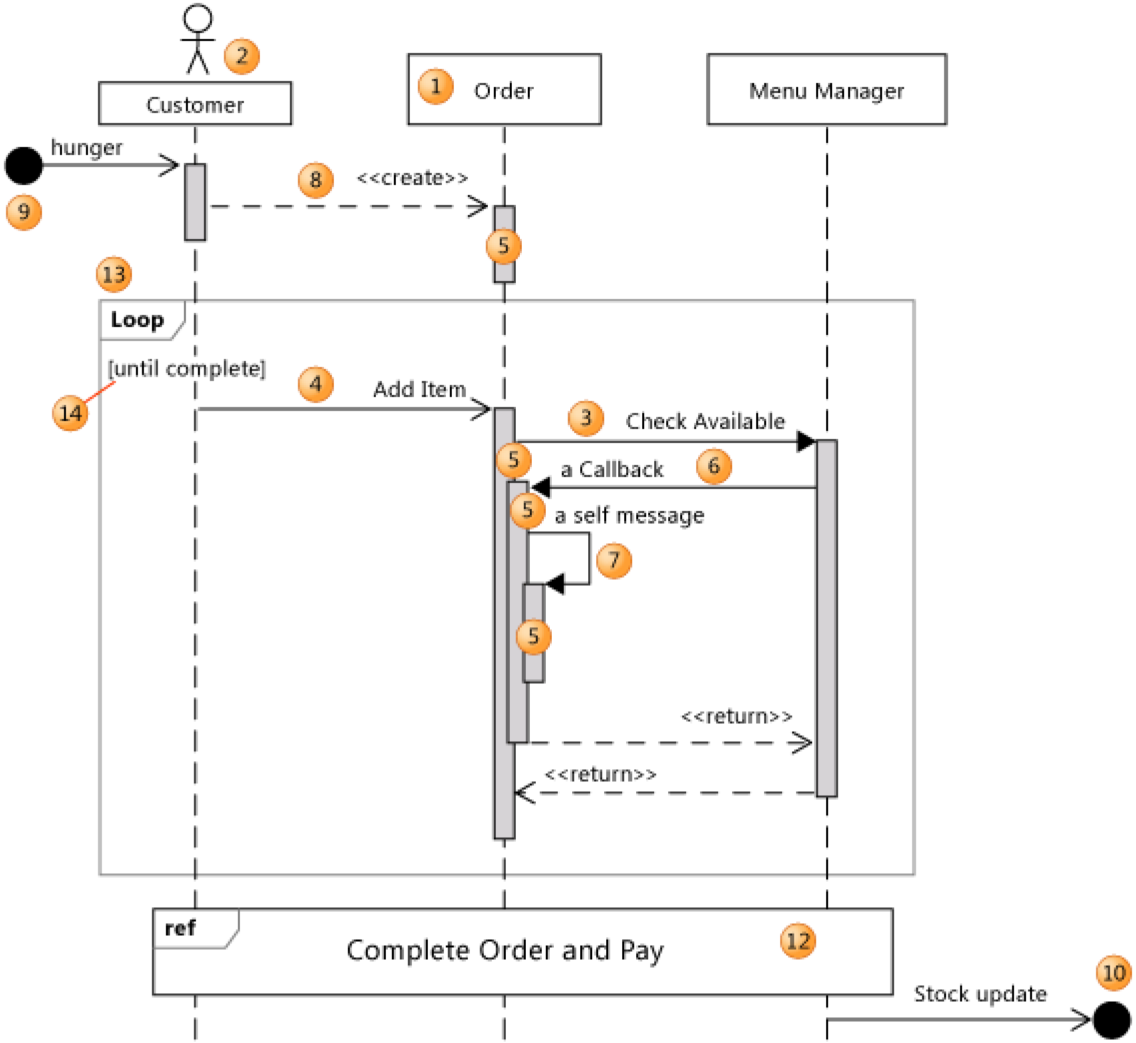
1	Component
2	Provided interface
3	Required interface
4	Dependency

Logical View: Class Diagram



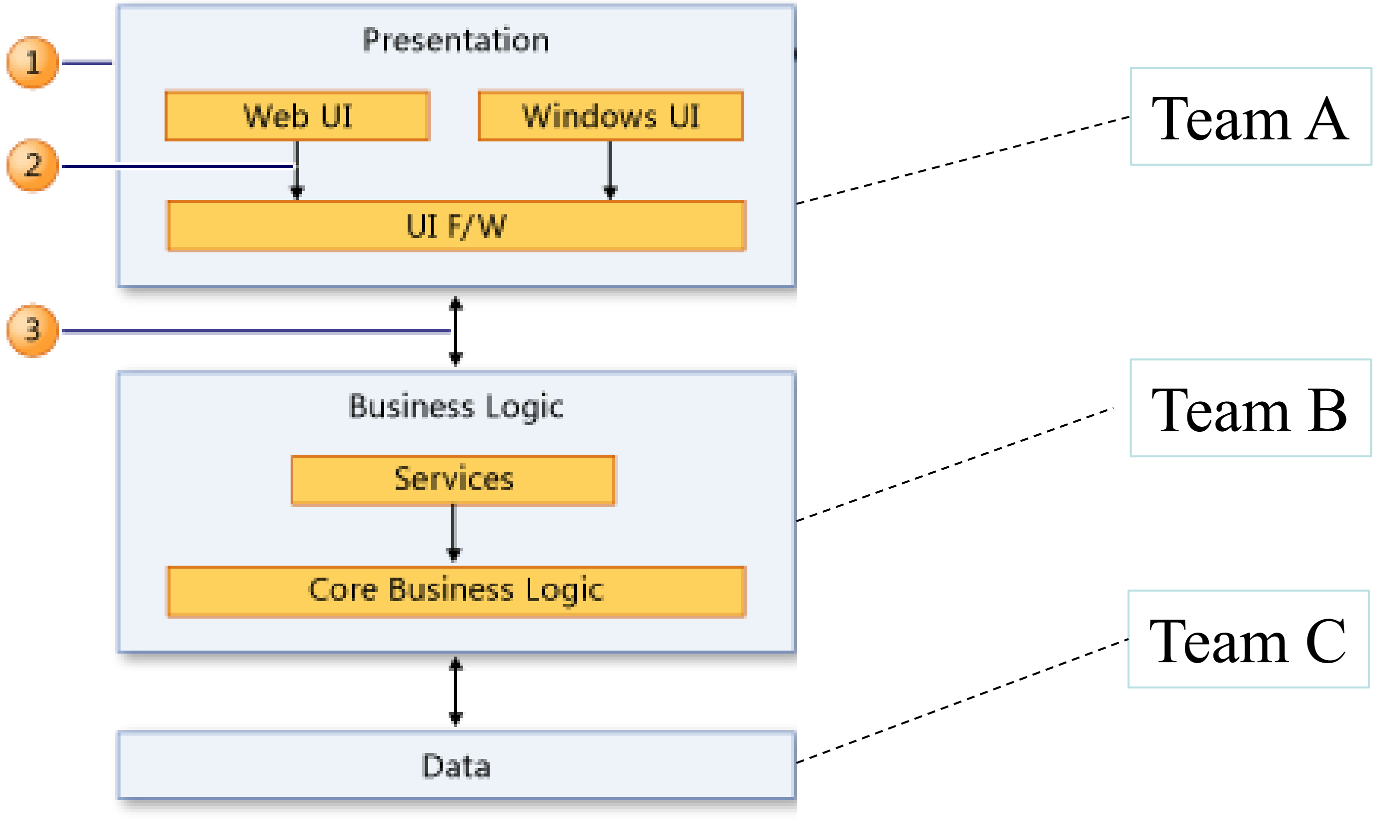
1	Class
3	Attribute
4	Operation
5	Association
5a	Aggregation
5b	Composition
9	Generalization

Logical View: Sequence Diagram

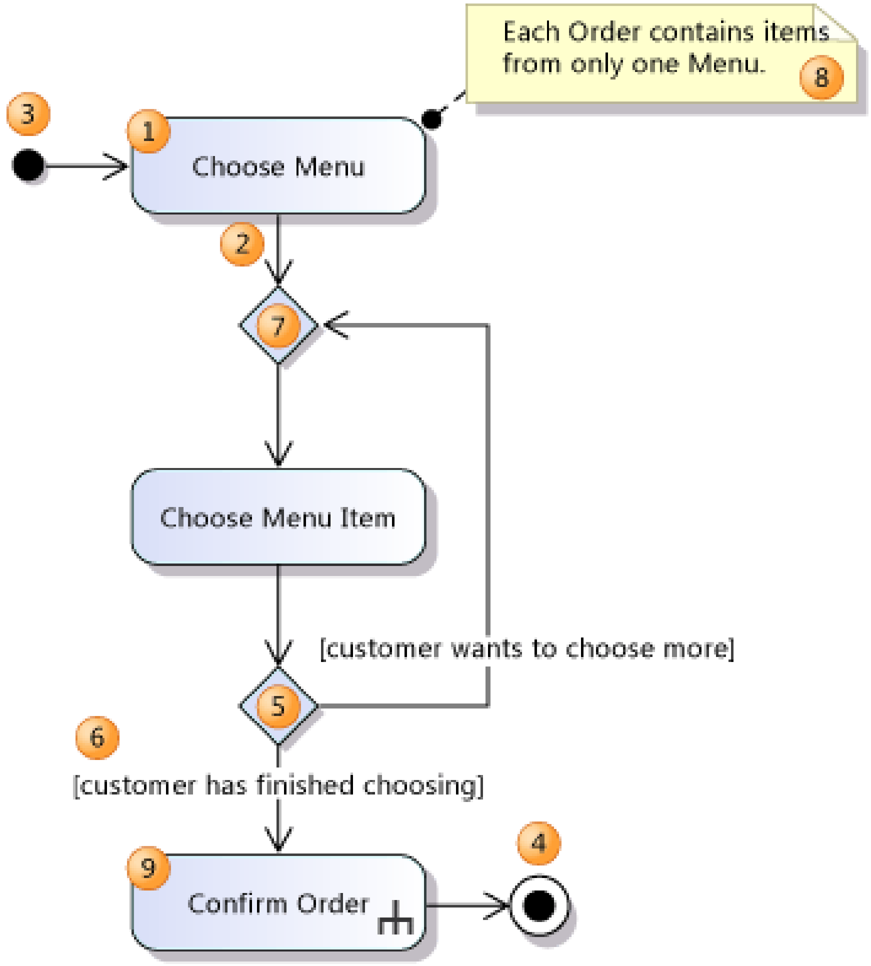


1	Lifeline
2	Actor
3	Sync. Message
4	Async. Message
5	Execution occurence

Development View: Layer Diagram

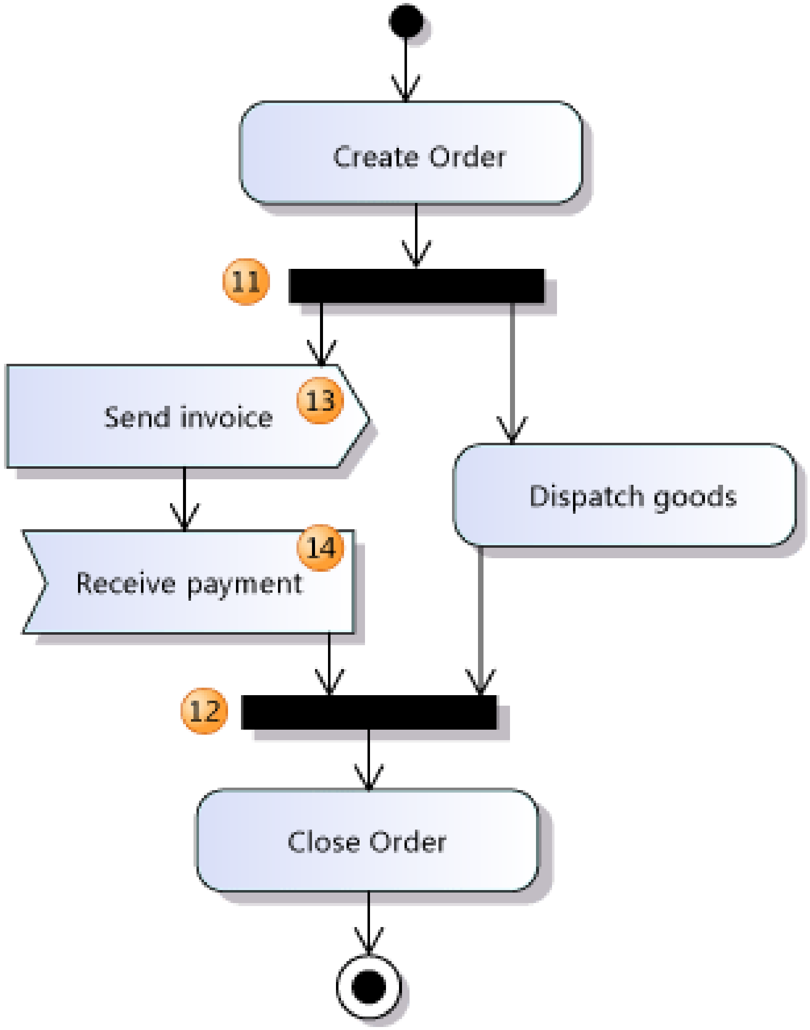


Process View: Activity Diagram



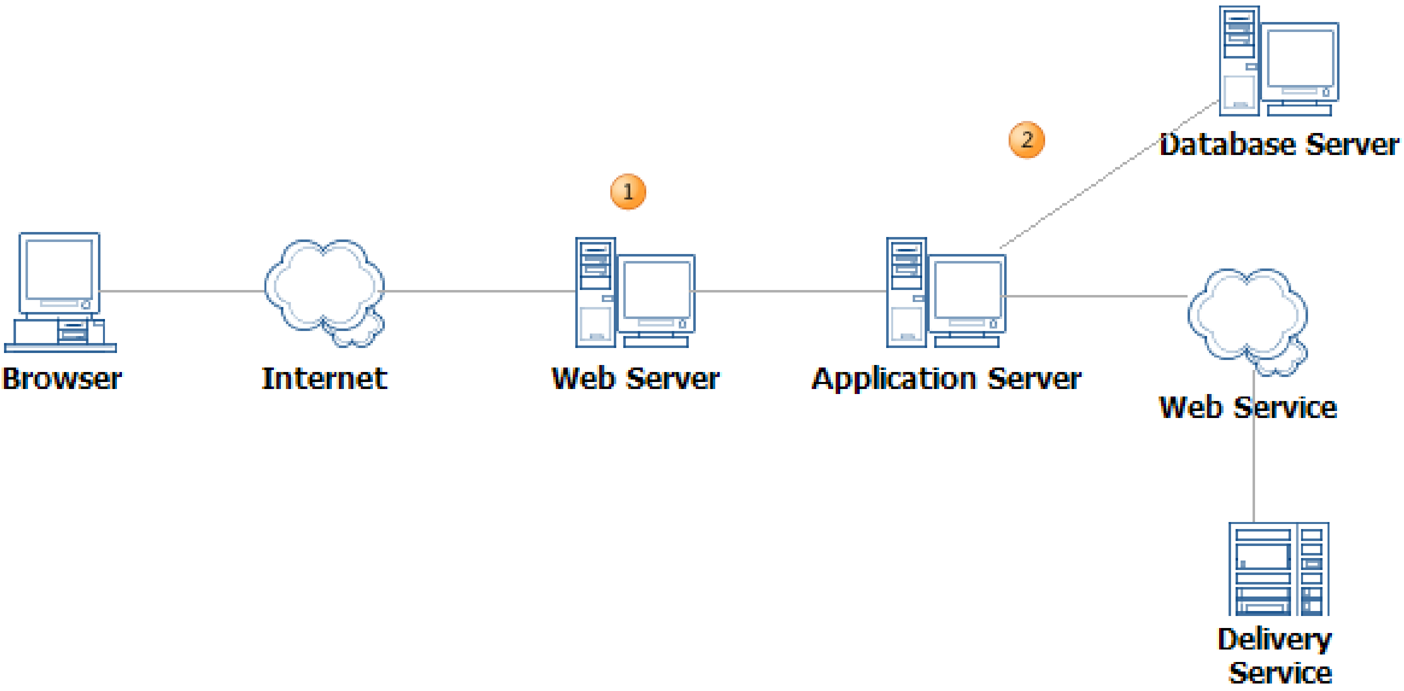
1	Action
2	Control flow
3	Initial node
4	Final node
5	Decision node
7	Merge node

Process View: Activity Diagram



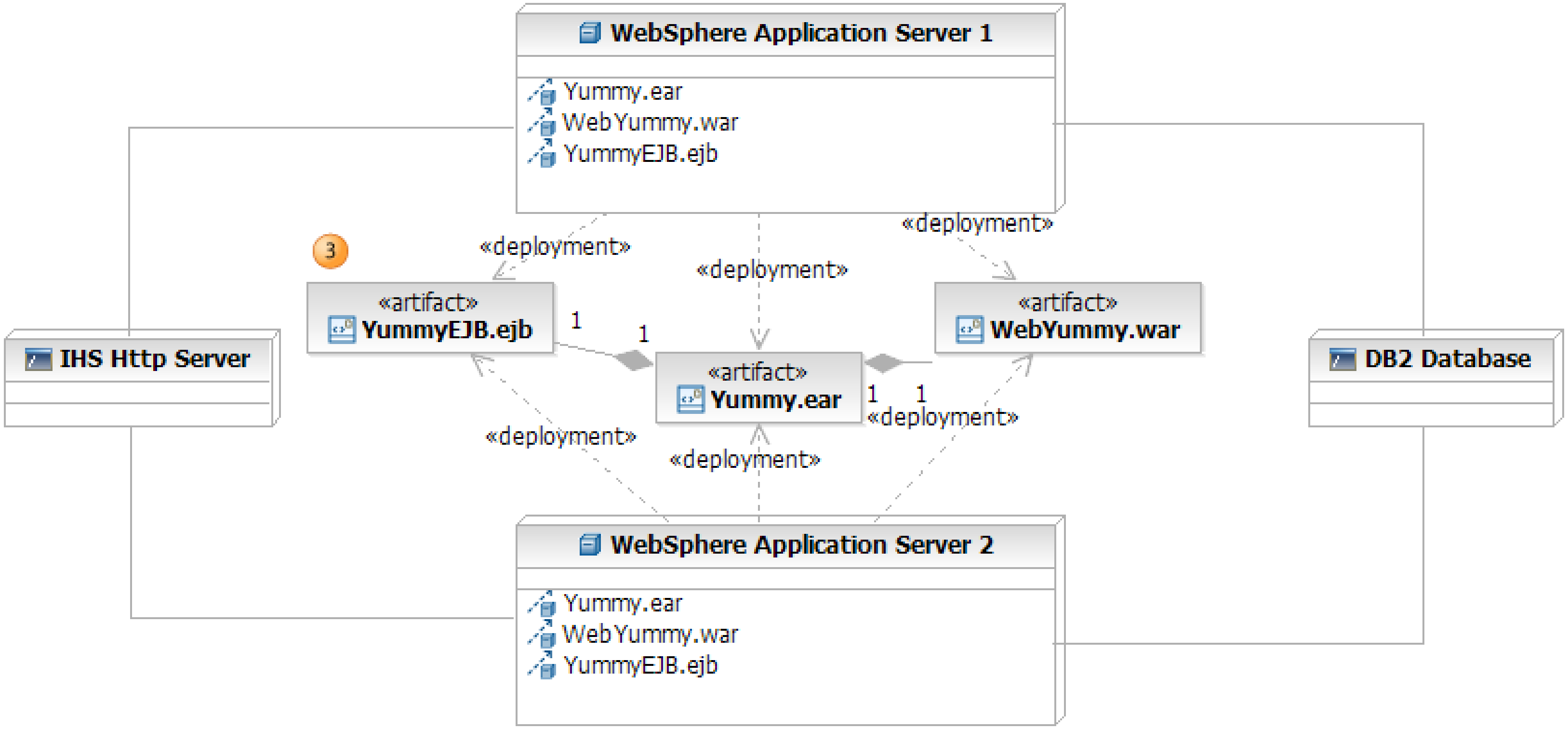
11	Fork node
12	Join node

Physical View: Deployment Diagram



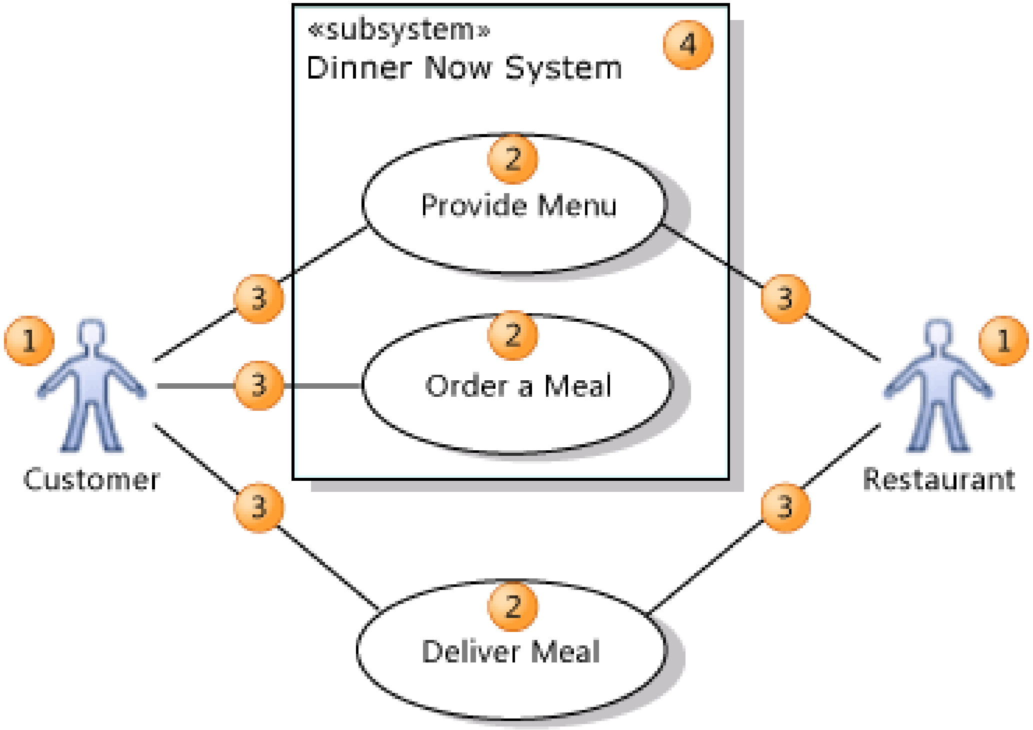
1	Node
2	Association

Physical View: Deployment Diagram



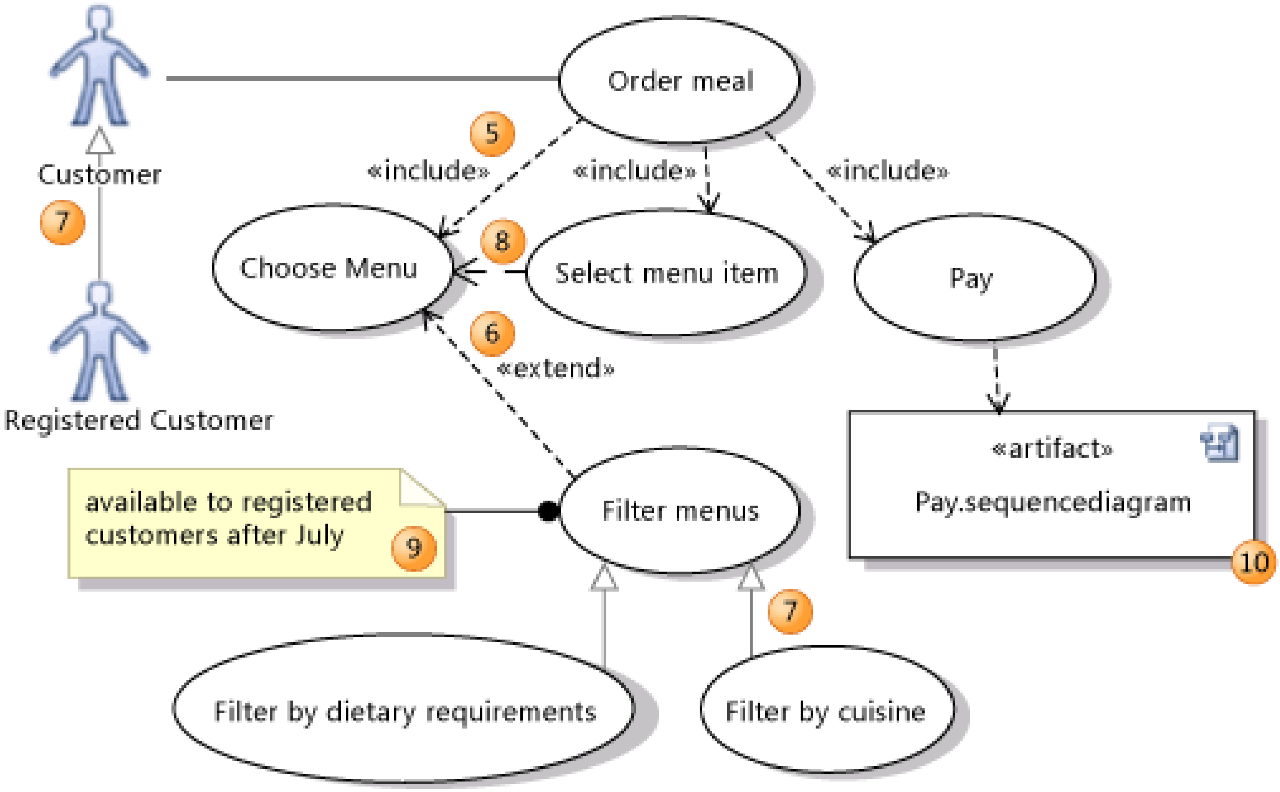
3	Artifact

Scenarios View: Use Case Diagram



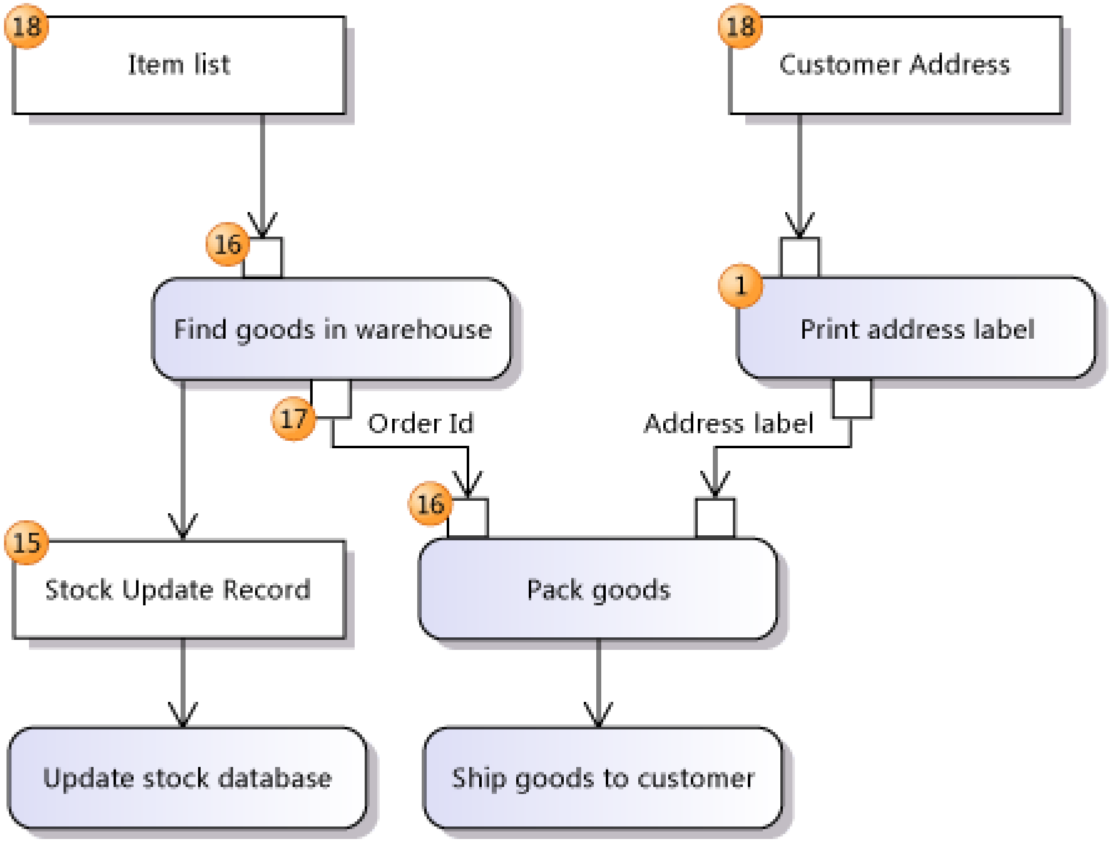
1	Actor
2	Use Case
3	Association
4	Subsystem

Scenarios View: Use Case Diagram



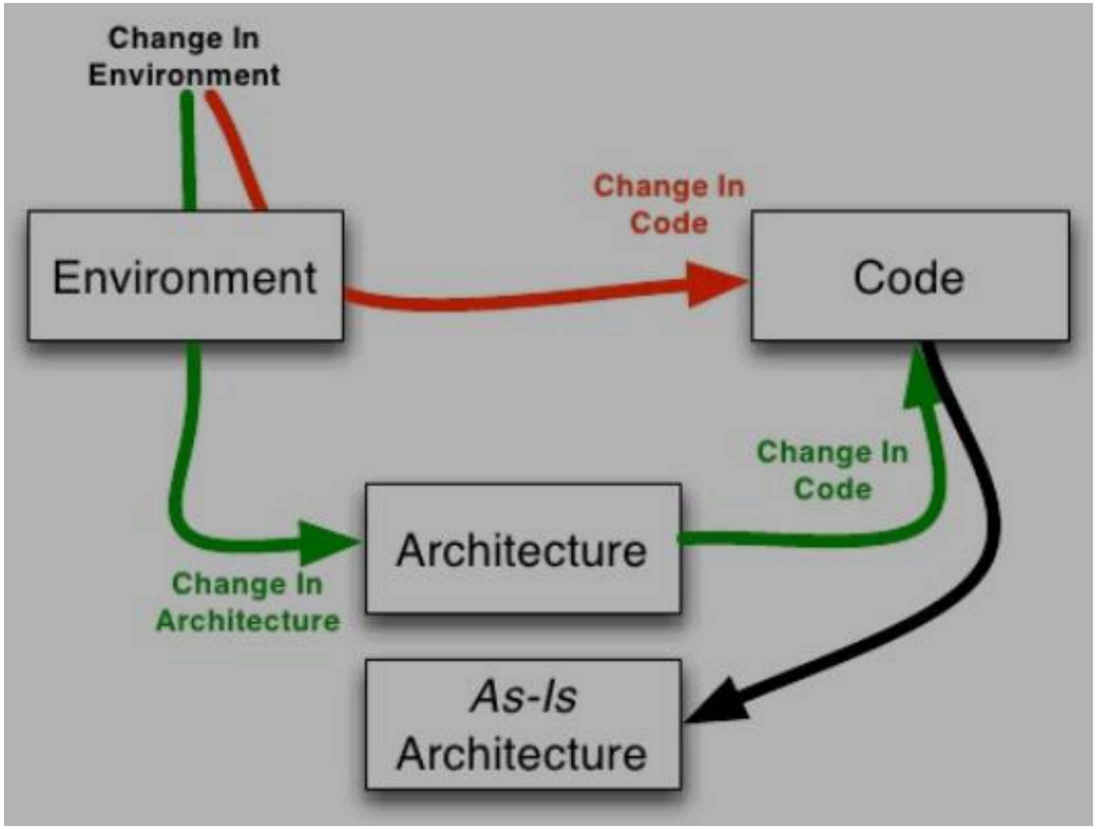
5	Include
6	Extend
7	Inheritance
8	Dependency

Data View: Activity Diagram

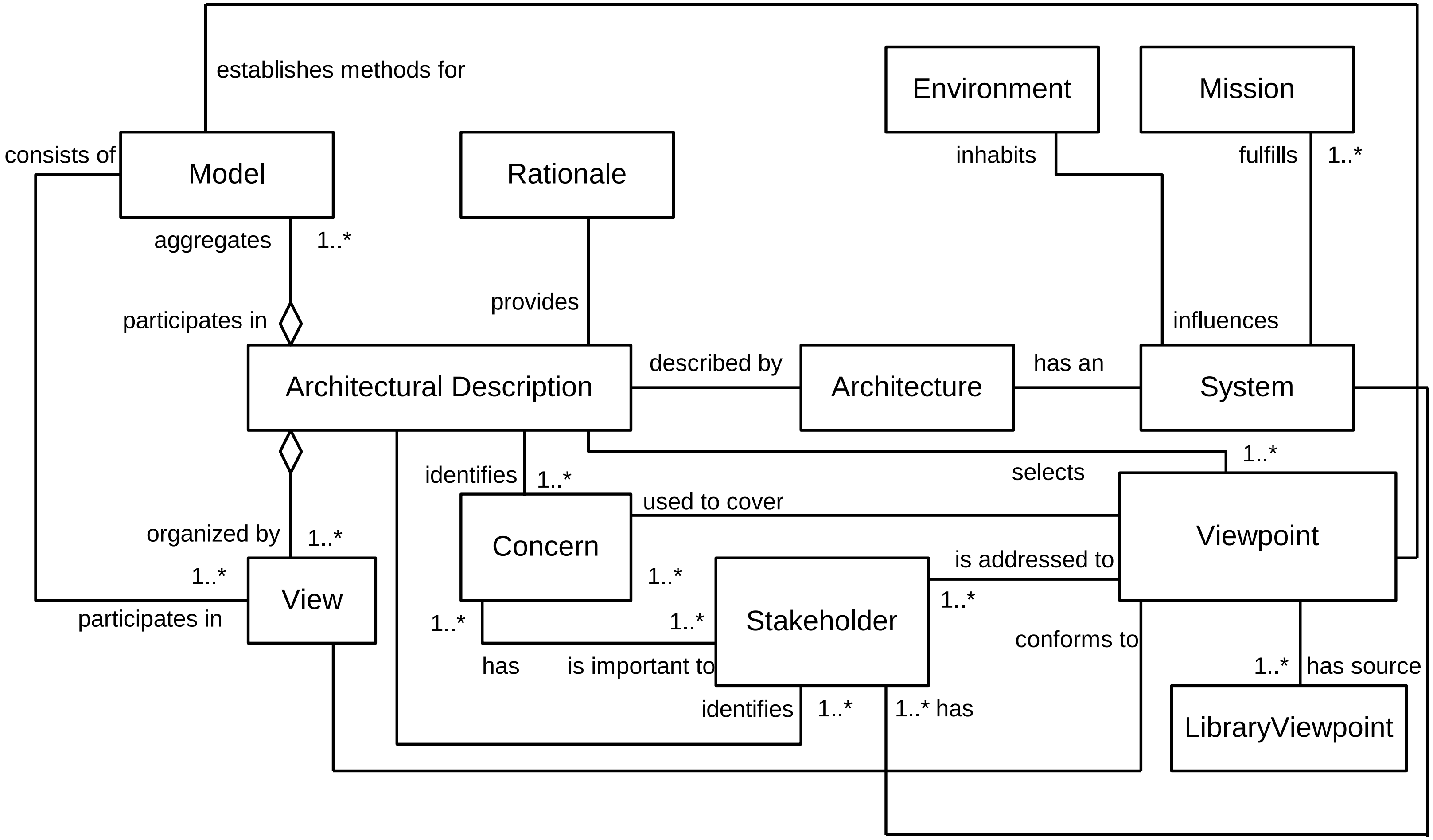


15, 18	Object node
16	Input pin
17	Output pin

The ”Sad” Reality



Wrap-up: IEEE-Std-1471 Conceptual Framework



establishes methods for
Environment
Mission
consists of
Model
Rationale
inhabits
fulfills
1..*
aggregates
1..*
provides
participates in
influences
Architectural Description
described by
Architecture
has an
System
1..*
identifies
1..*
selects
used to cover
organized by
1..*
Concern
Viewpoint
1..*
1..*
is addressed to
View
Stakeholder
participates in
1..*
1..*
1..*
conforms to
has
is important to
1..*
has source
identifies
1..*
1..* has
LibraryViewpoint

Wrap-up

- Architecture should be the product of a **single architect or a small group of architects** with an identified leader.
- Architect team should have **functional requirements** for the system and an articulated prioritized list of **quality attributes** that the architecture is expected to satisfy.
- Architecture should be well **documented**, and circulated and **reviewed** by system stakeholders.
- Architecture should be **analyzed** for applicable quantitative measures and formally evaluated for quality attributes before it is too late to make changes to it.
- Architecture should lend itself to **incremental refinement and implementation**.

Tack så mycket!

Frågor?