ECE4574 – Large-Scale SW Development for Engineering Systems
Lecture 13 – Architectural Issues for Engineering Systems

Creed Jones, PhD

Course Updates

- Sprint 2 is underway
- Homework 3 (the final one) will be posted later this week
  - Due November 8
- Quiz 5 is TODAY!!!
  - covers lectures 12-14
  - open 7 PM to 1 AM

Today’s Objectives

Some example architectural patterns

- N-tier Client Server
- Messaging
- Publish-Subscribe
- Broker
- Process Coordinator

Embedded Systems Software Architectures

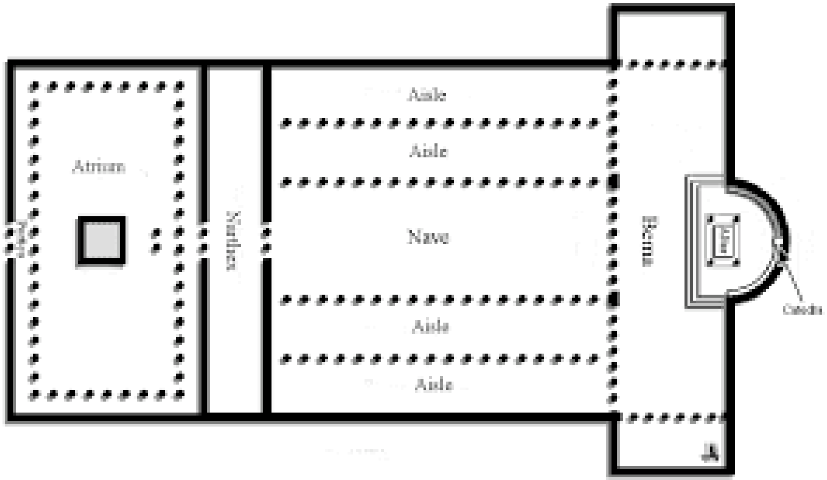
- Software Architecture for Embedded Systems
- Automotive Software Engineering

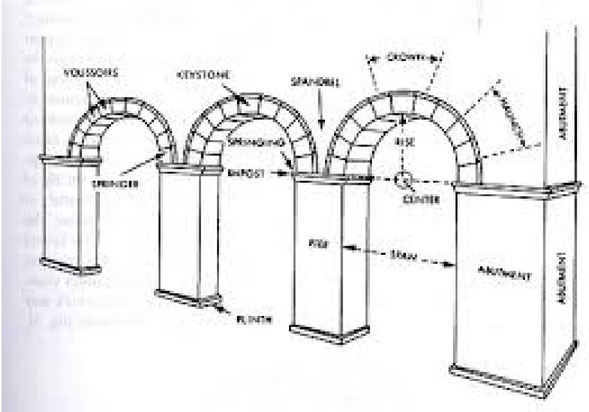
ARCHITECTURAL PATTERNS

There is a distinction between an *Architectural Pattern* and a *Design Pattern*

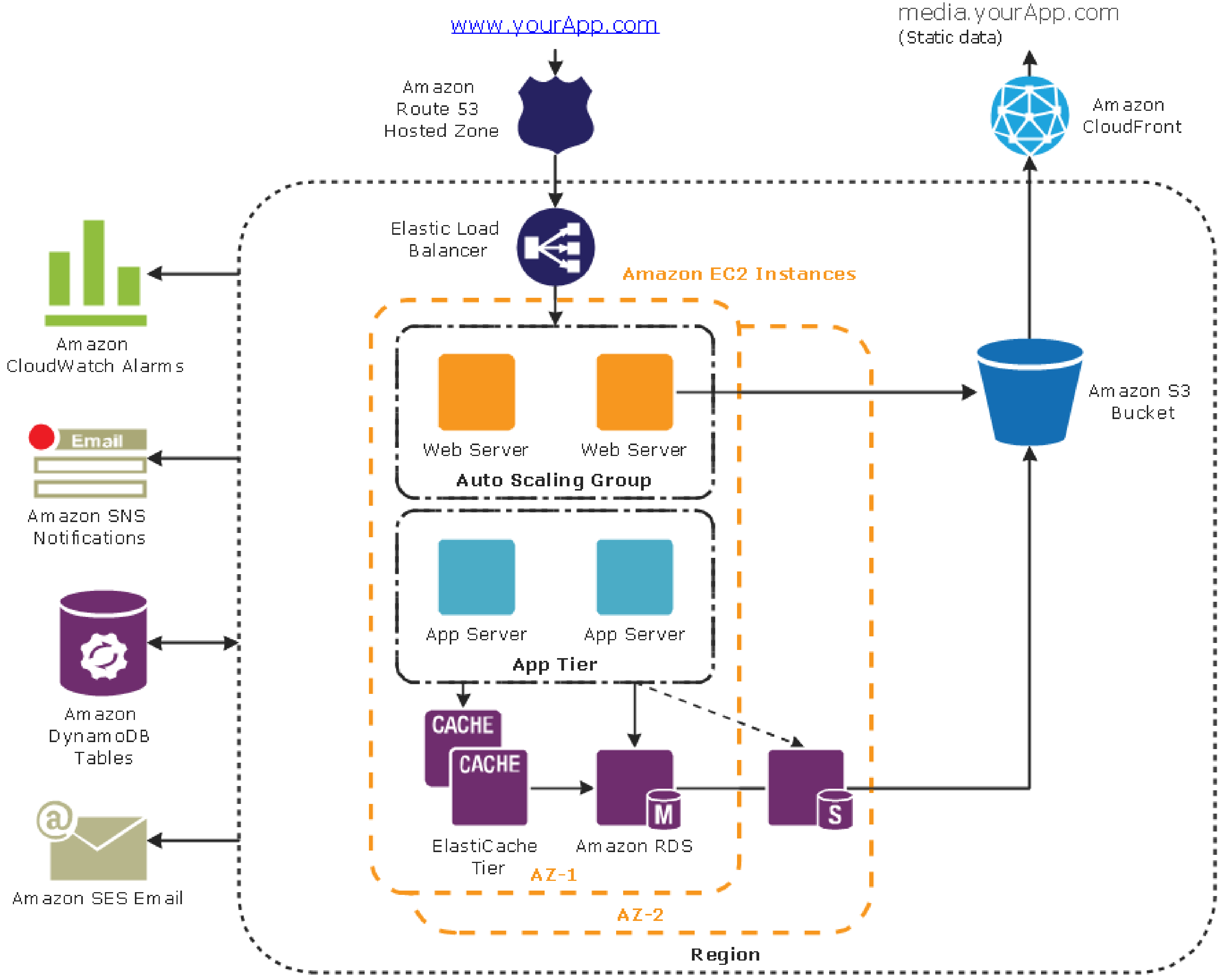
- The architectural pattern is a time-tested arrangement of overall components that defines the structure of the entire project
- Design patterns are used to specify the shape and working of specific pieces

- In classical church architecture, the basilica is an Architectural Pattern
- A vaulted arch is a design pattern

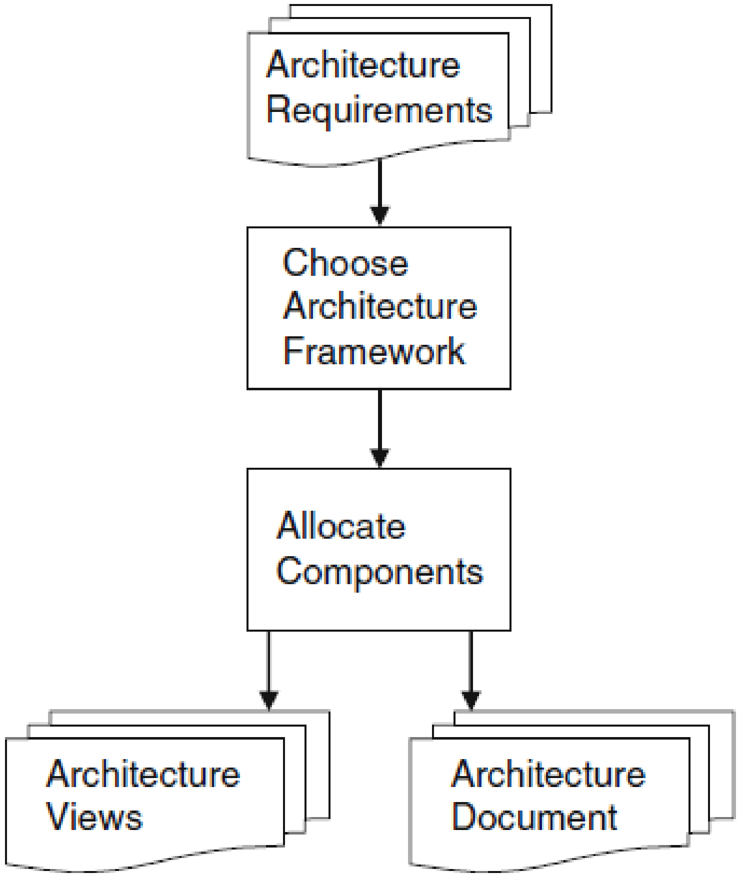




3-Tier Auto-scalable Web Application Architecture



"The design stage itself has two steps…choosing an overall strategy for the architecture, based around proven architecture patterns…[and] specifying the individual components that make up the application"

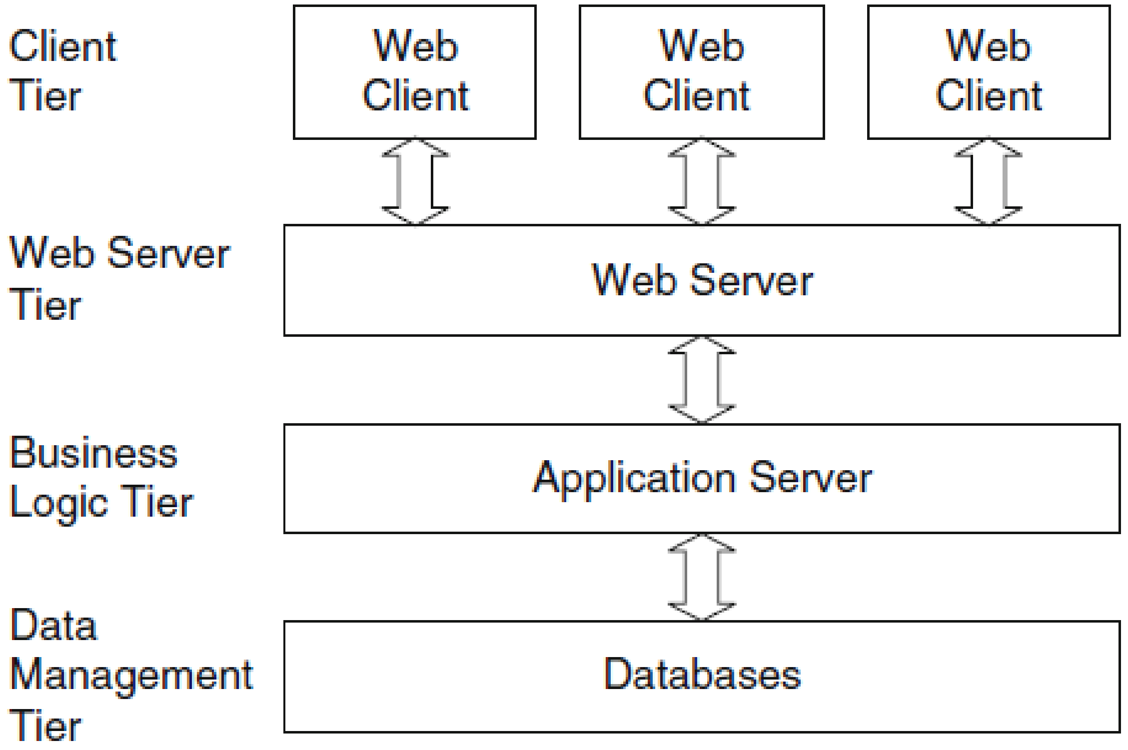


- Note that this process does <u>not</u> iterate
  - hopefully
- We choose a framework from past experience and familiarity, aligned with the requirements
- That leads to defining components

ARCHITECTURAL PATTERNS – SOME EXAMPLES

The N-tier client server framework supports web applications

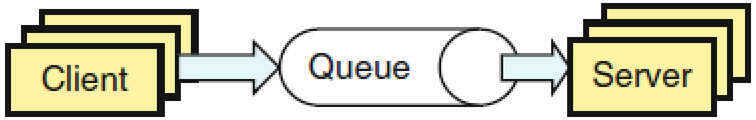
1. Separation of Concerns
   - Presentation, business and data are handled in different tiers
2. Synchronous Communications
   - Tiers talk to each other via "request-reply"
   - Tiers wait for a response before proceeding
3. Flexible Deployment
   - One or multiple machines



How does the N-Tier Client Server meet general quality requirements?

Quality attribute	Issues
Availability	Servers in each tier can be replicated for reliability (at degraded performance)
Failure handling	If a server fails while a client is communicating, we want the request routed to a live server for completion (most app servers do this)
Modifiability	Most changes will be limited to one tier of the design
Performance	High performance, if the servers support many concurrent threads and the connection between tiers is lightweight
Scalability	Servers can be replicated up to a point; the data management tier will often become the bottleneck

The Messaging pattern is a simple concept in which all components communicate by FIFO queues with non-trivial size



1. Asynchronous Communication

   - Clients generally do not block on responses; messages are handled by the queue until a server accepts it

2. Configurable Quality of Service (QoS)

   - May be unreliable or reliable, slow or fast

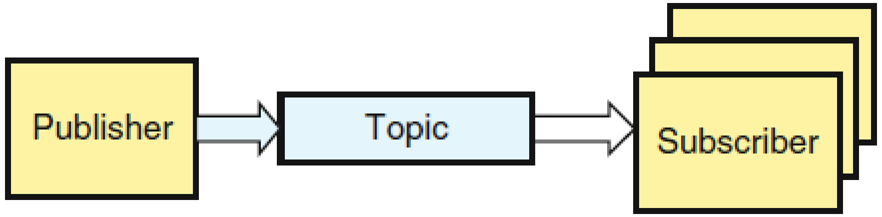
3. Loose Coupling

   - Servers process all messages; clients don't send requests to specific servers (sources and destinations are not known)

The Messaging architecture pattern has specific quality characteristics

Quality attribute	Issues
Availability	Duplicate queues with the same name on different machines provide redundancy
Failure handling	If a queue fails while a client is communicating, it can find a replica queue and repost
Modifiability	If messages are self-describing (XML), changes should be localized
Performance	Messaging can have very high throughput, depending on the reliability requirement
Scalability	Queues can reside on the endpoints or dedicated messaging servers; this is a highly scalable pattern

The Publish-Subscribe architectural pattern allows for one message to be received by many destination processes



1. Many-to-many messaging

   - Messages are received by any subscriber who "registers" for it, by *topic*

2. Configurable QoS

   - Reliable or unreliable, but also point-to-point or multicast

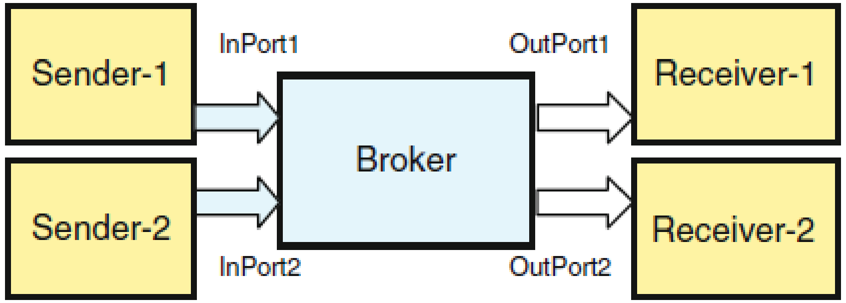
3. Loose Coupling

   - Servers process all messages; clients don't send requests to specific servers (sources and destinations are not known)

Publish-subscribe has good quality in many areas of concern

Quality attribute	Issues
Availability	Duplicate queues for the same topic on different machines provide redundancy
Failure handling	If a topic server fails while a client is communicating, it can find a replica server and repost
Modifiability	Loose coupling allows modification
Performance	Publish-subscribe can have very high throughput, depending on the reliability requirement (especially for multicast publish)
Scalability	Topics can be hosted on independent servers or clusters; this is a highly scalable pattern

The Broker pattern allows for exchange of messages that demand translation or other processing



1. Hub-and-spoke architecture

   - the broker is a central messaging hub; ports are associated with a specific message format

2. Performs message routing

   - Logic to determine message paths is in the router

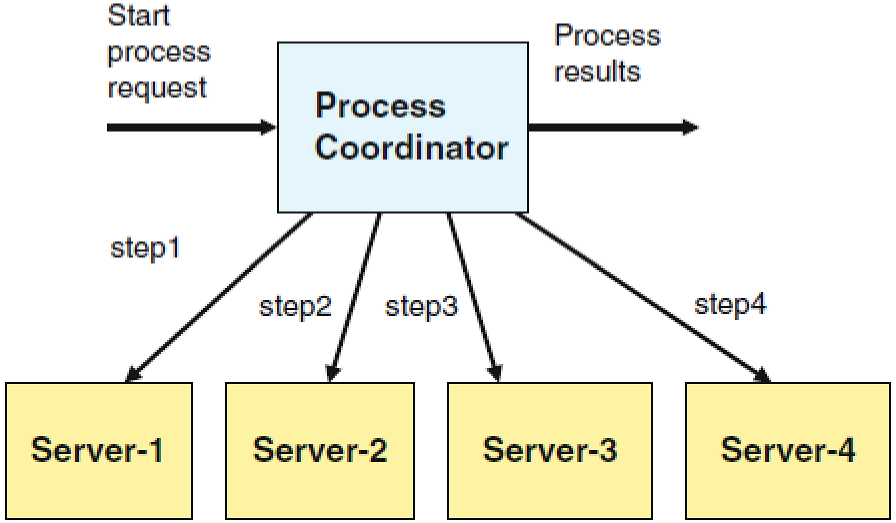
3. Performs message transformation

   - Message type to message type, as determined by the source and destination port

The Broker pattern and its quality characteristics

Quality attribute	Issues
Availability	High availability will require that the broker be replicated
Failure handling	Input ports expect particular message types; incompatible messages are discarded; senders can find a live broker in the case one fails
Modifiability	All changes in format or sequence are handled in the broker
Performance	The broker component can become a bottleneck
Scalability	Implementing the broker on a cluster of machines allows the Broker pattern to scale

The Process Coordinator can implement complex business rules in a system with multiple servers



1. Process encapsulation

   - Sequence of steps needed to accomplish the business process; receives a request and responds by initiating a list of steps

2. Loose coupling

   - Servers are not aware of the larger business context, but merely respond to requests

3. Flexible communications

   - Each path can be synchronous or asynchronous (queued); may use a callback function

A Process Coordinator pattern has some good quality attributes and some that are so-so

Quality attribute	Issues
Availability	We must replicate the coordinator to avoid a single point of failure
Failure handling	Lengthy business sequences dictate difficult and complex failure recovery mechanisms
Modifiability	The process coordinator holds the definition of the business process in one place
Performance	The coordinator must be able to handle multiple requests, perhaps at different points in their process
Scalability	We can replicate the coordinator to load-balance and allow scaling

Once an architectural framework is chosen, we allocate components (what are the large pieces of the project)

- Identifying the major application components, and how they plug into the framework.
- Identifying the interface or services that each component supports.
- Identifying the responsibilities of the component, stating what it can be relied upon to do when it receives a request.
- Identifying dependencies between components.
- Identifying partitions in the architecture that are candidates for distribution over servers in a network.

When identifying components, we remember our usual good design principles

- Minimize dependencies between components (loose coupling)
- Highly cohesive components
- Keep dependencies on COTS components (like middleware, DB, etc) localized
- Structure components hierarchically
- Minimize calls between components
- Look for places to apply design patterns

EMBEDDED SYSTEMS SOFTWARE ARCHITECTURES

Software Engineering for Embedded Systems

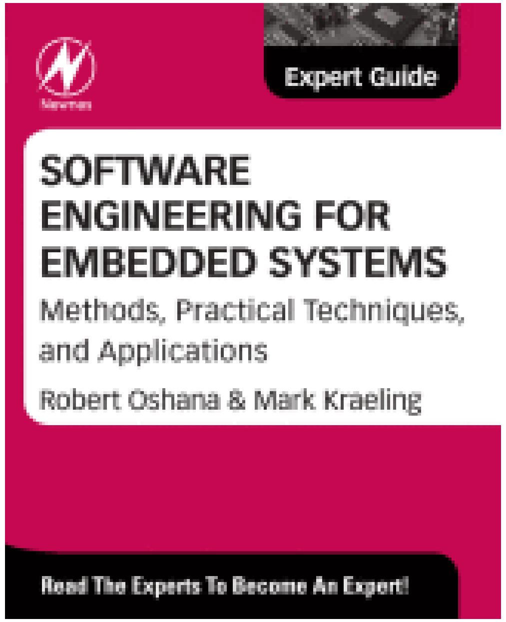
Software Engineering for Embedded Systems; Methods, Practical Techniques, and Applications, Newnes, 2013

ISBN 978-0-12-415917-4

Available through the VT library

This book highlights three important points for embedded systems architecture:

1. Constraints are paramount
2. Some new design patterns are useful
3. Deployment must be considered



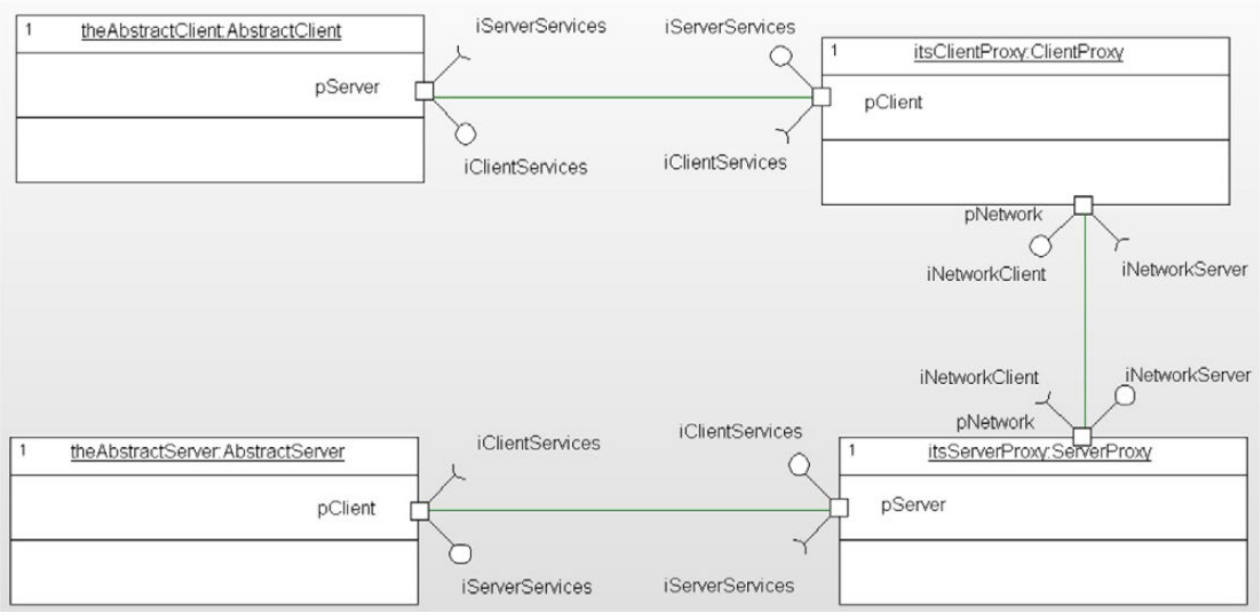
“Architecture is about system-wide optimization”

- Embedded systems usually have strict performance constraints
  - Throughput, reaction time, accuracy, resources, uptime…
- Hard vs. Soft real-time systems
  - Soft: faster response is preferred, and occasional reaction slightly over the “deadline” can be acceptable
  - Hard: deadlines are absolute
- Failure recovery is crucial
  - There’s no one there to reboot

Some new or enhanced design patterns are useful in embedded systems

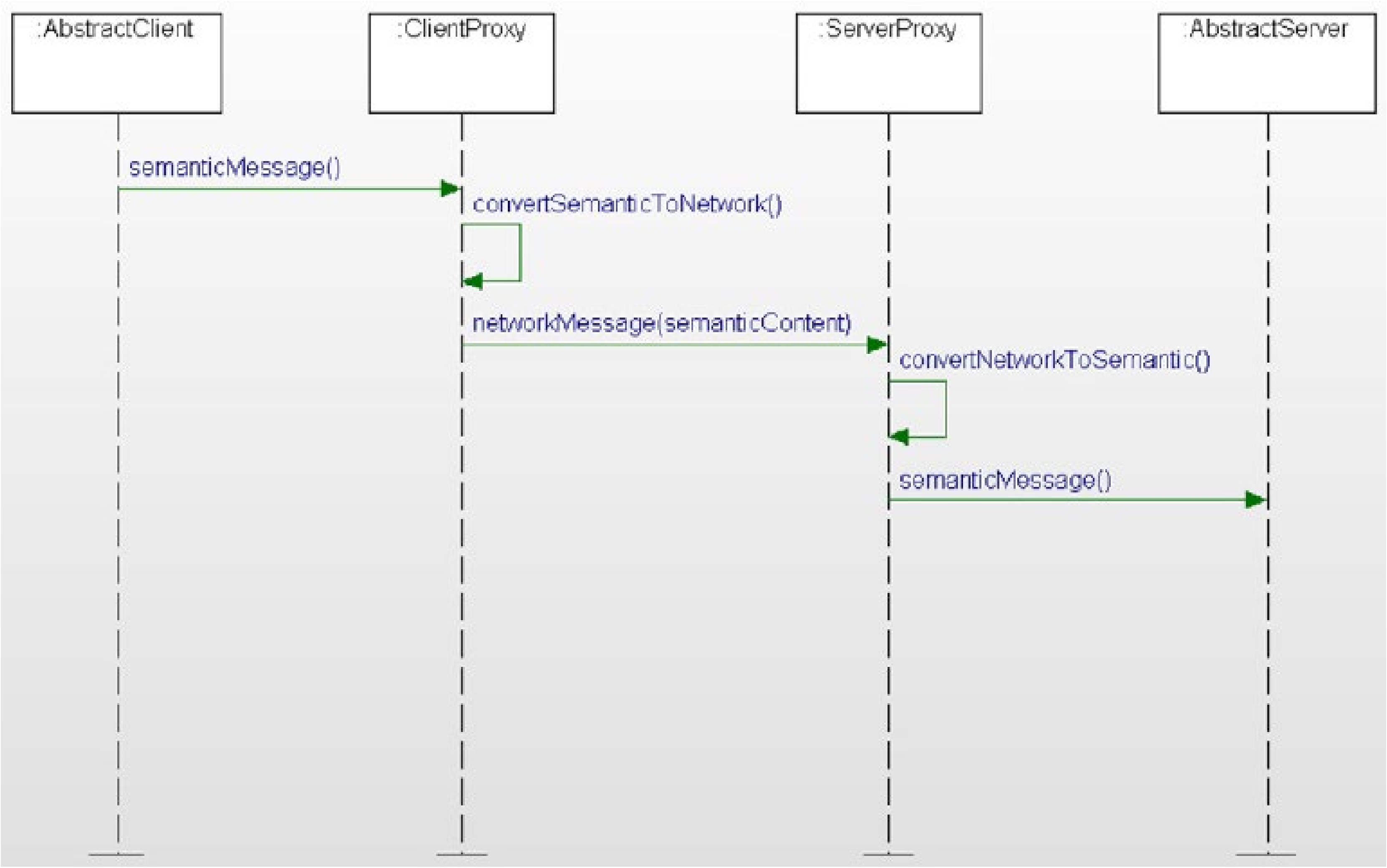
- Port proxy pattern
- Scheduling patterns:
  - Cyclic executive
  - Time-triggered cyclic executive
  - Rate monotonic scheduling (RMS)
  - Earliest deadline first (EDF)

In the *port proxy* pattern, a number of units are connected via discrete connections called ports



- The client and the server are both coded without knowledge of the messaging channel details
- Each has a proxy that handles conversion to/from the protocol of the channel
- Changing to a new network or middleware would only require updates to the proxies

The *semantic message* (the essence of the message) is converted to and from the network format



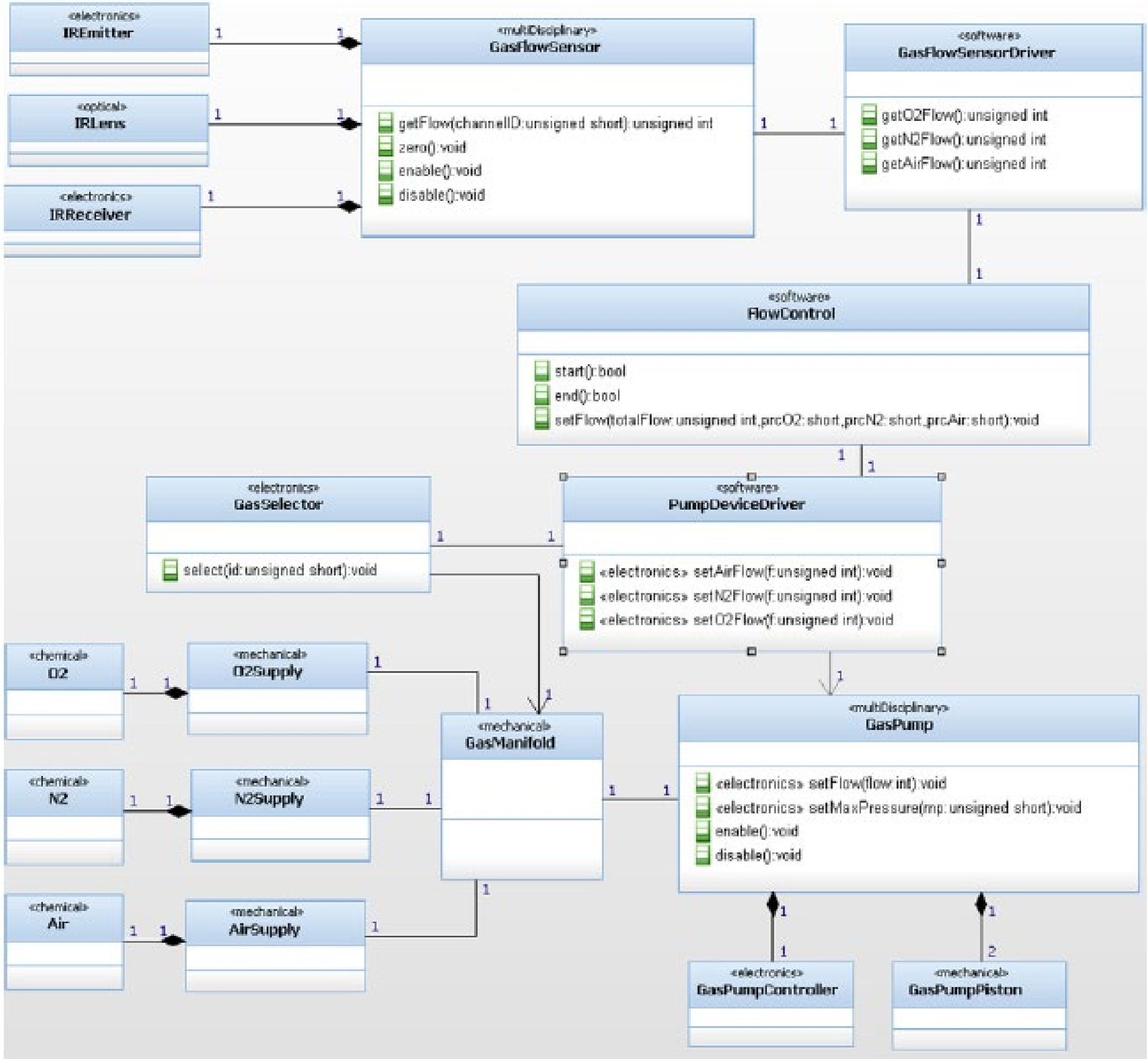
The various *scheduling* patterns enable implementations of different ways of sequencing and assigning tasks

Pattern	Description	Benefits	Drawbacks
Cyclic executive	The scheduler runs a list of tasks (each to completion) in the same order in a repetitive cycle.	Simple Fair Highly predictable	Low responsiveness Unstable Suboptimal performance Requires tuning
Time-triggered cyclic executive	Same as cyclic executive except that each cycle begins on a time-based epoch.	Simple Fair Highly predictable Synchronizes with reference clock	Low responsiveness Unstable Suboptimal performance Requires tuning
Rate monotonic scheduling (RMS)	All tasks are assumed to be periodic with the deadline at the end of the period. Priorities are assigned at design time on the basis of period - shorter periods have higher priority. Highest priority task always runs.	Stable Optimal Robust	Unfair May not scale to very complex systems More complex Less predictable
Earliest deadline first (EDF)	Priorities are assigned at run-time based on the nearness of the deadline (i.e., its urgency). Highest priority waiting task always runs.	Optimal Robust	Unfair Naïve implementation Leads to thrashing Unstable More complex Less predictable

The deployment of software is a different and important concern for embedded systems design

- What code will run where?
- What “OS” facilities and platforms will be available?
- Sometimes, the connection between system elements is:
  - Sporadic: think of cell connection in outlying areas
  - Occasional: scheduled connection to hosts (overnight?)

- Often, code deploys other code
- Bootloading and startup must be supported by system architecture



Note that the deployment diagram shows sensors and mechanical elements (!)

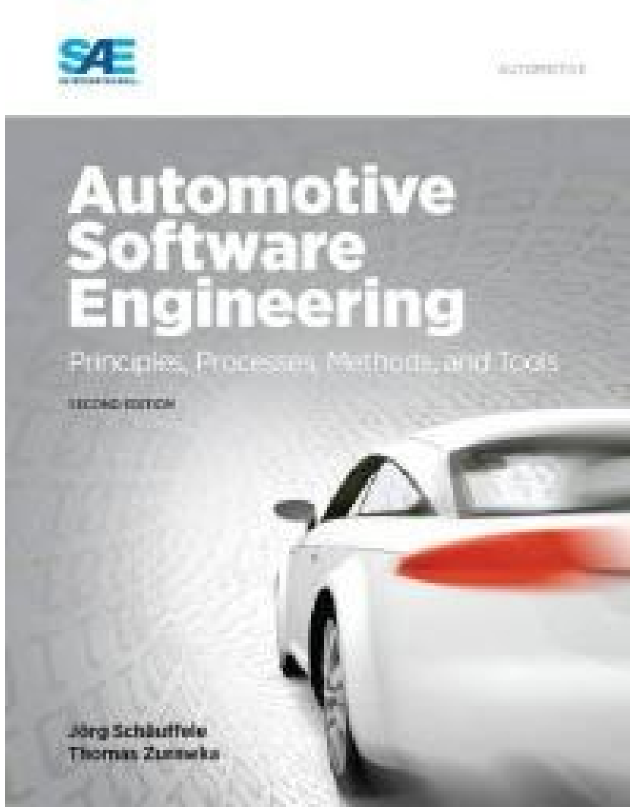
Automotive Software Engineering

Schaeuffele, J., & Zurawka, T. (2016). *Automotive software engineering, second edition*. ProQuest Ebook Central https://ebookcentral.proquest.com

Available through the VT library

Applies software engineering principles to automotive systems, which are:

- real-time
- modular
- networked
- cost-sensitive



CAN is a multi-master serial messaging bus; FlexRay is faster and more reliable but more expensive

On occasion, Ethernet is also used for automotive systems

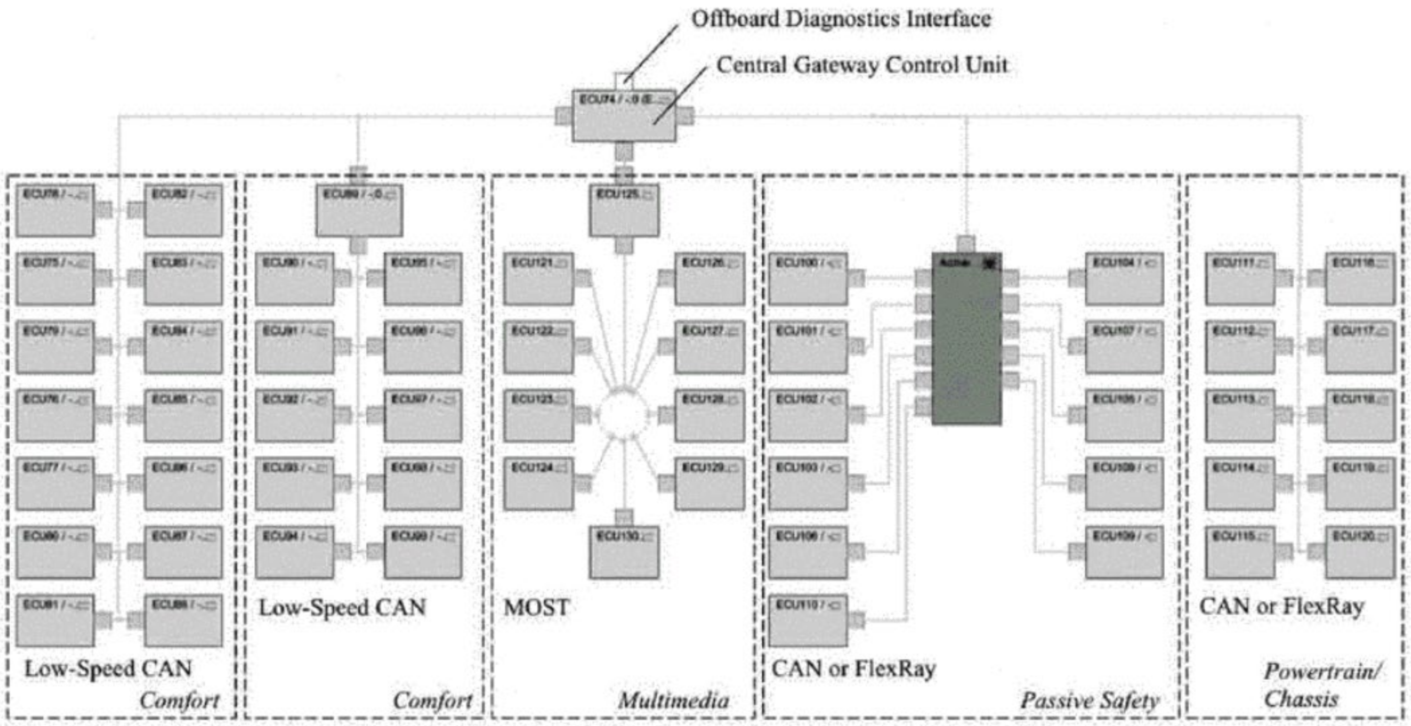


Figure 2.66 ECU Network of a premium class vehicle.

Automotive systems software is highly modular and component-based

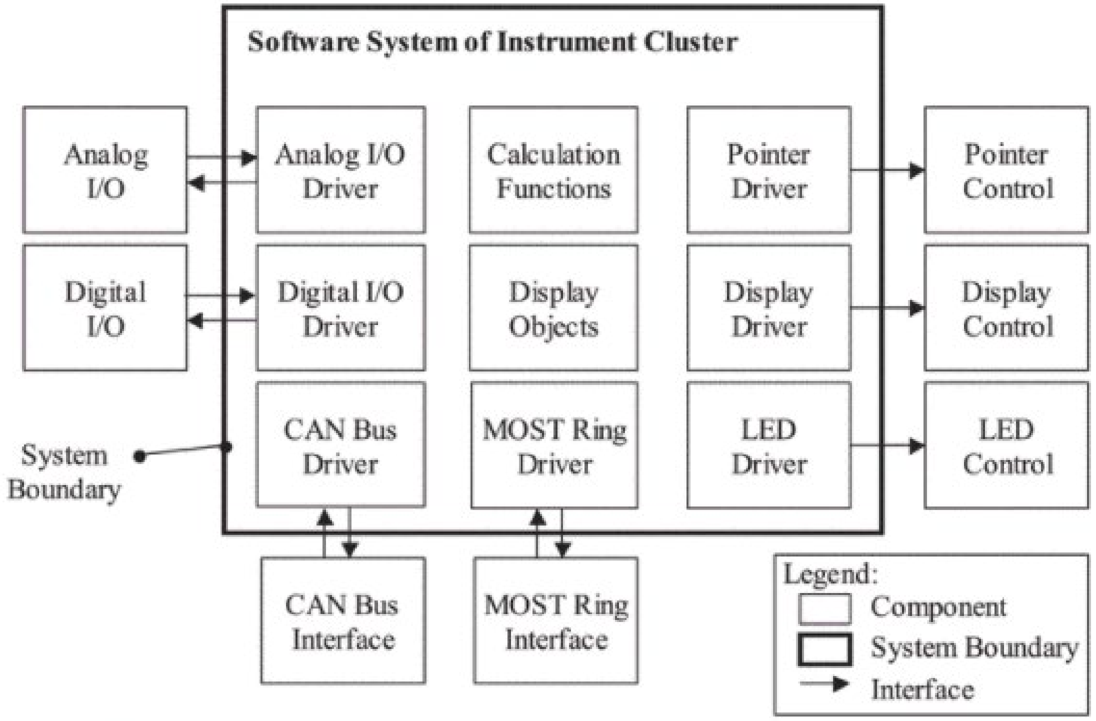


Figure 4.20 Context and interface model of the instrument cluster software.

- Software components mirror hardware components
- Some real-time events are tolerant of response lag
  - comfort systems
- Others are not
  - engine control and safety systems

AUTOSAR is a consortium-based standard for open architecture for automotive systems

VIRGINIA TECH

ece BRADLEY DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

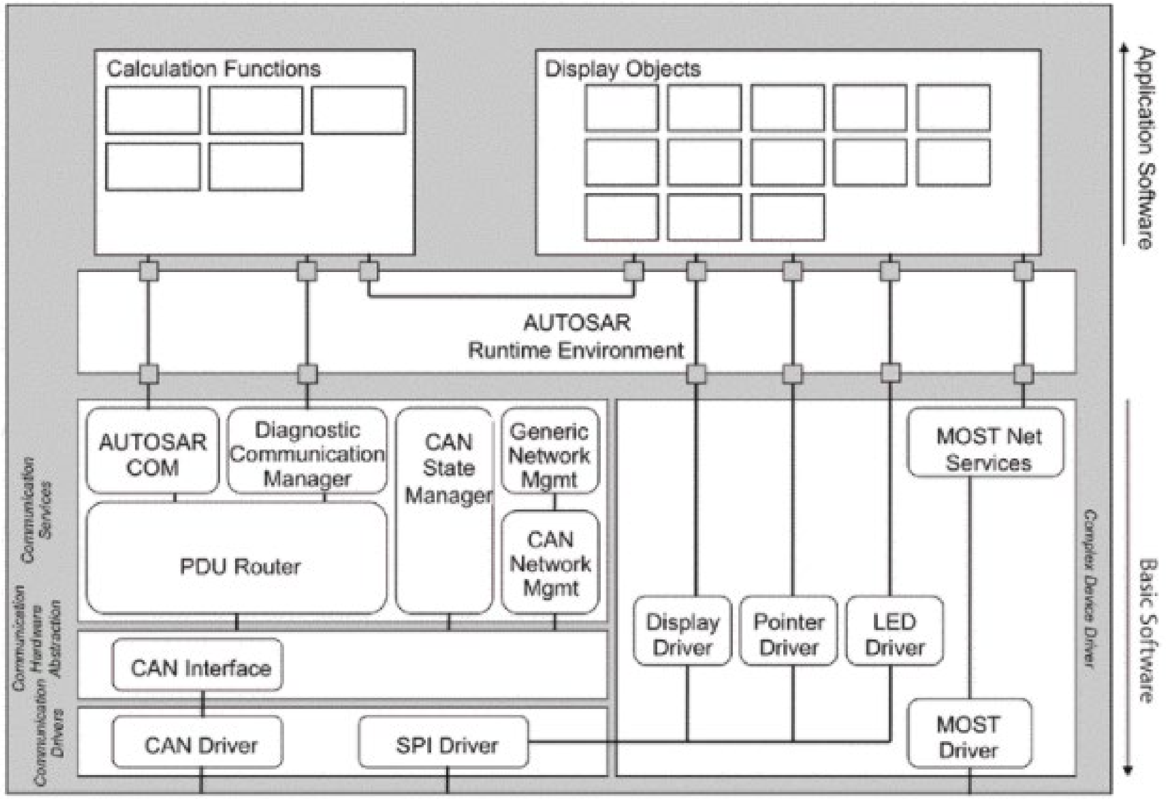


Figure 4.22 Software architecture of the instrument cluster.

- https://www.autosar.org/fileadmin/ABOUT/AUTOSAR_EXP_Introduction102020.pdf
- It defines three layers of software functions:
  - basic software: hardware support and general functions
  - runtime environment: messaging
  - application software: specific to the automotive task
- The AUTOSAR runtime serves as component middleware

Specification of automotive SW components reflects the needs of the application

- The data model for information to be processed:
  - scalar, vector, matrix
- The behavioral model (control flow)
- The real-time model (latency and throughput guarantees)

- Note the importance of testing

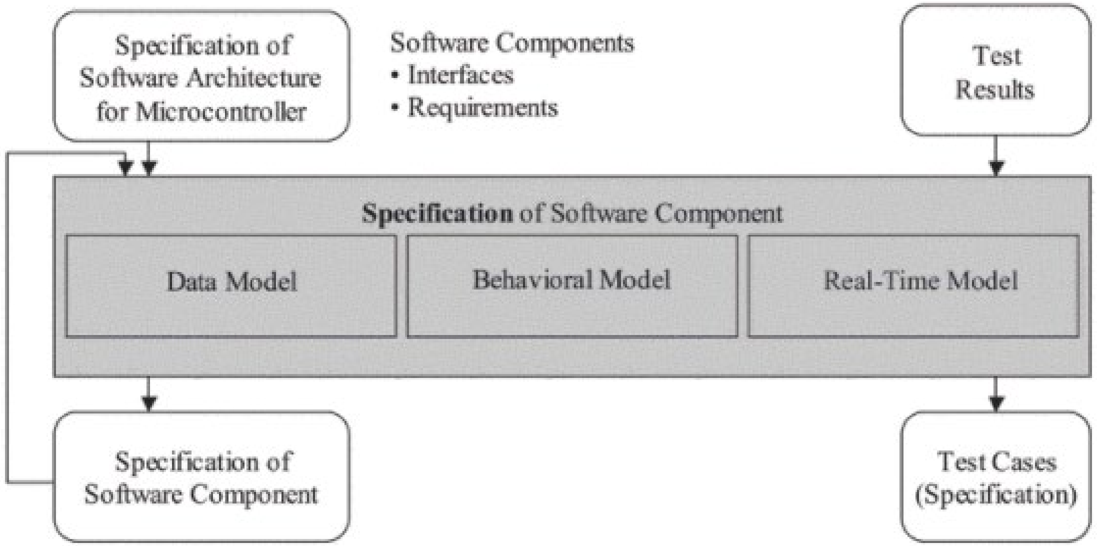


Figure 4.24 Specification of a software component.

Today’s Objectives

Some example architectural patterns

- N-tier Client Server
- Messaging
- Publish-Subscribe
- Broker
- Process Coordinator

Embedded Systems Software Architectures

- Software Architecture for Embedded Systems
- Automotive Software Engineering