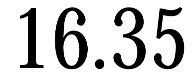


Aerospace Software Engineering

Software Architecture
The “4+1” view
Patterns

Prof. Kristina Lundqvist
Dept. of Aero/Astro, MIT

Why Care About Software Architecture?

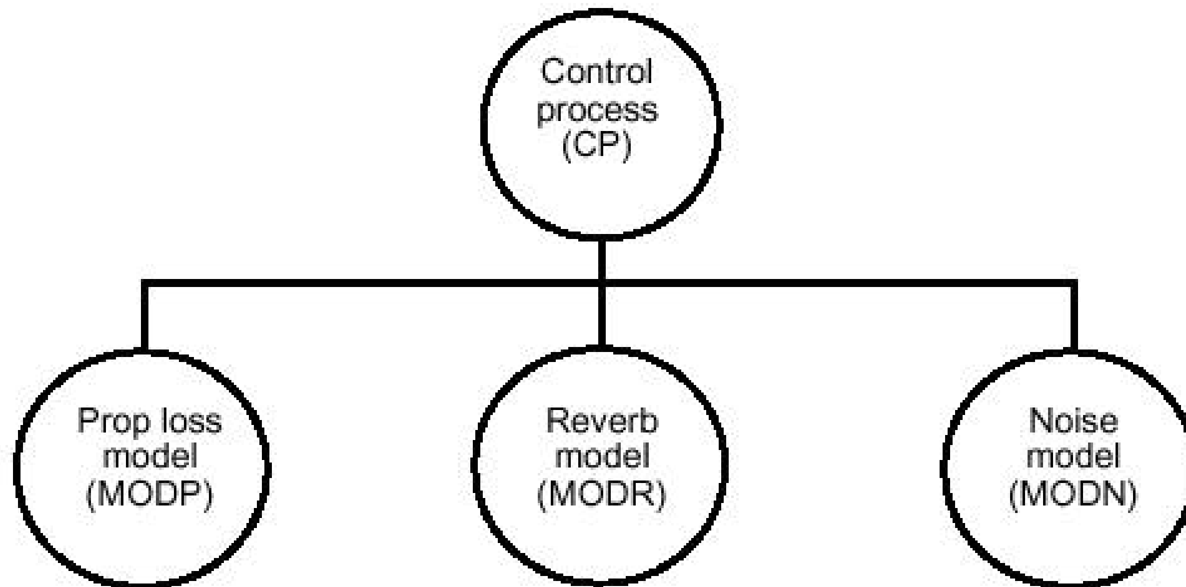
- An architecture provides a vehicle for *communication among stakeholders*
- It is the manifestation of the *earliest design decisions* about a system
- It is a *transferable, reusable abstraction* of a system

Every system has an architecture (which may or may not be known!) - But how we represent it is of crucial importance

What Does Software Architecture Do?

- An architecture defines constraints on its implementation
- Dictates organizational structures
- Inhibits or enables a system's quality attributes – which can be predicted
  - A good architecture is necessary, but not sufficient, to ensure quality
- Makes it easier to reason about and manage change

Is this a Software Architecture?



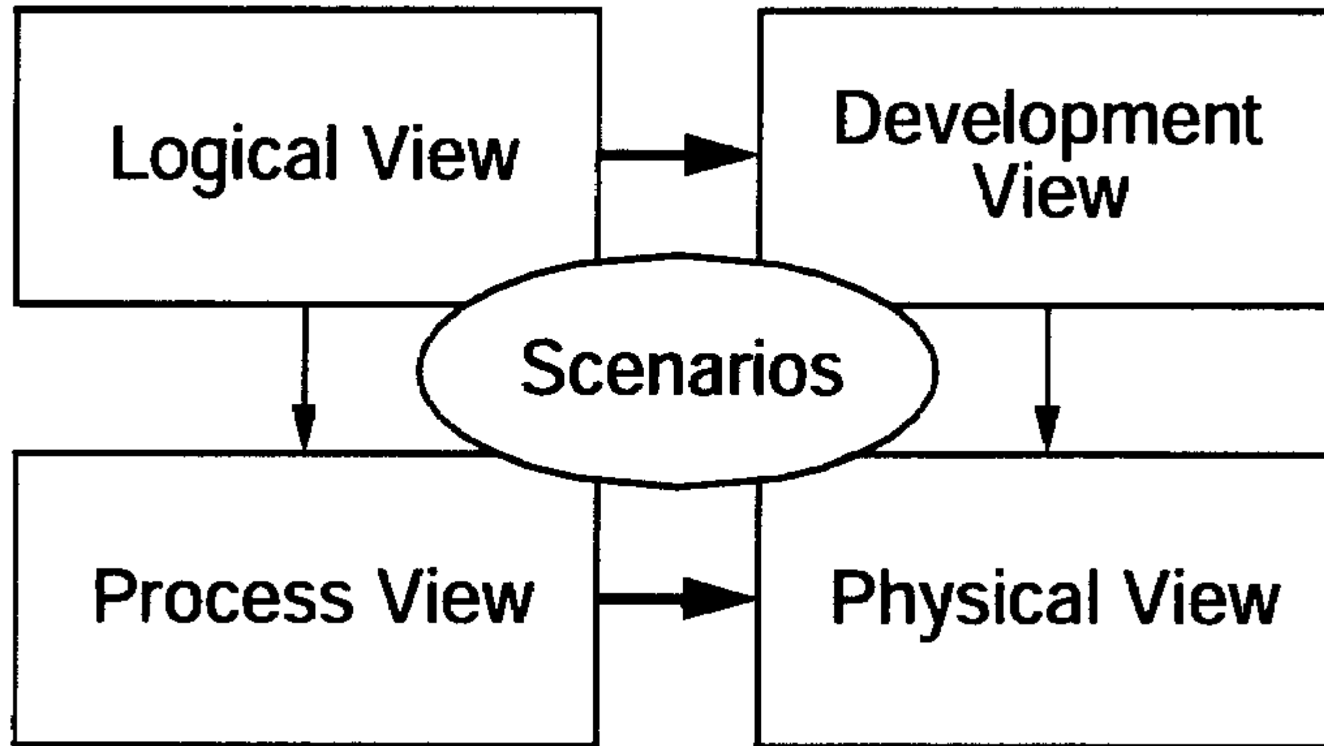
Architectural views

- Are used by different people
- Used to achieve different functional and non-functional qualities
- Used as a description and prescription
- Should be annotated to support analysis (scenarios aid in annotating views with design rationale)

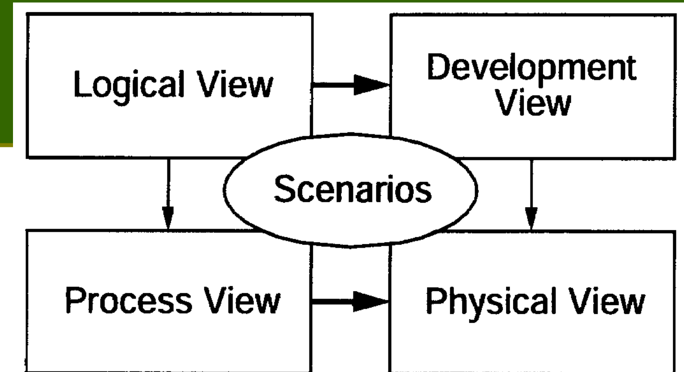
Views

- Software comprises many structures
  - Partial description of a system
- Philippe B. Kruchten: Four main views of software architecture that can be used to advantage in system-building + a distinguished fifth view that ties the other four together
  - The "four plus one" approach
    - logical view
    - process view
    - physical view
    - development view
    - + scenario view
- A view can be used to assess one or more quality attributes.

“4+1” View Architecture Model

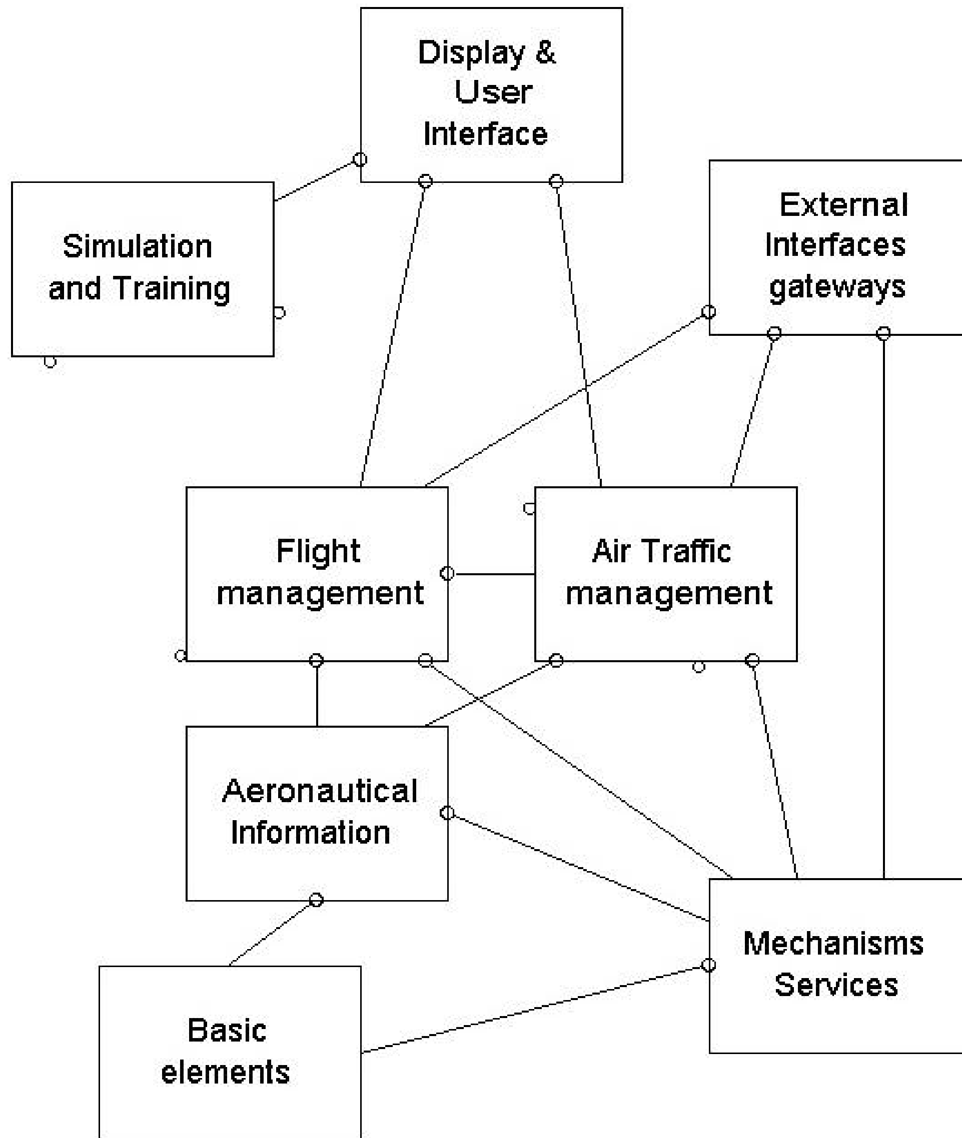


4 + 1: Logical View



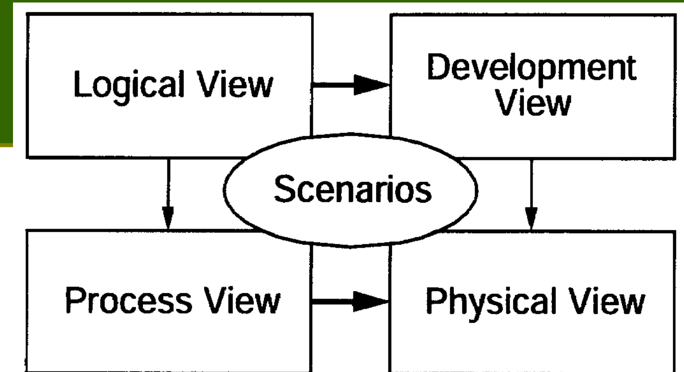
- The logical view supports the functional requirements, i.e., the services the system should provide to its end users.
- Typically, it shows the key abstractions (e.g., classes and interactions amongst them).

Logical View: Notation



Blueprint for an
Air Traffic Control System

4 + 1: Process View



- The process view gives the mapping of functions to runtime elements
- It takes into account some nonfunctional requirements, such as performance, system availability, concurrency and distribution, system integrity, and fault-tolerance.

Process View: Notation

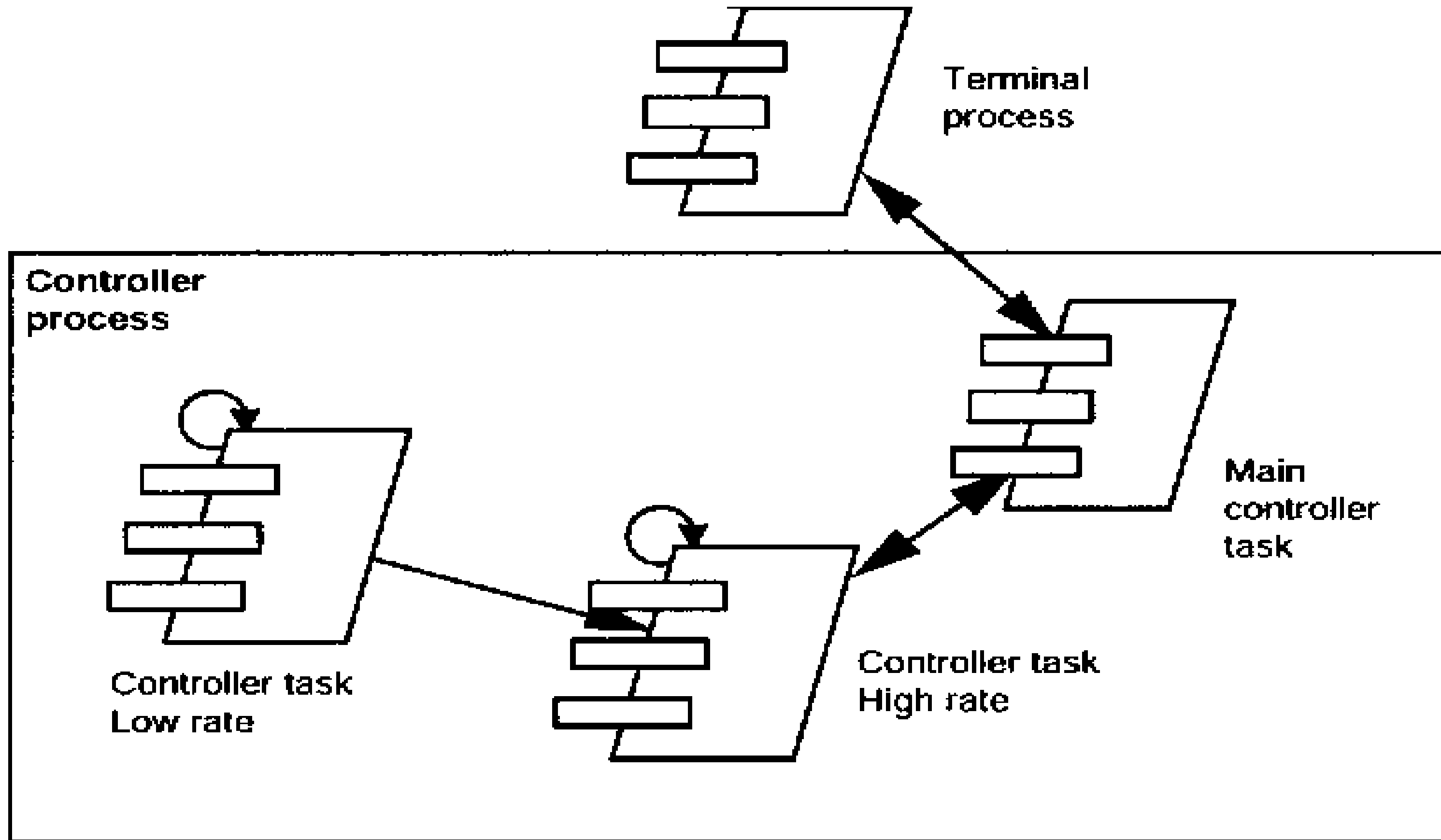
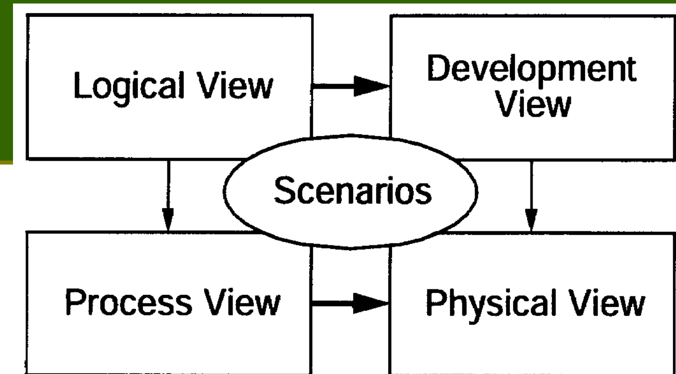


Figure 5 — Process blueprint for the Télic PABX (partial)

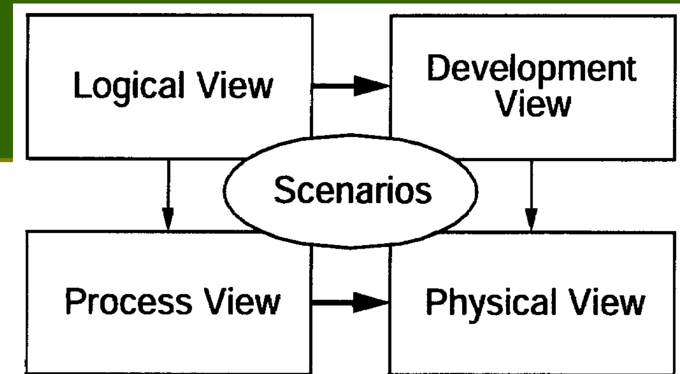
4 + 1: Physical View



- The physical view defines how the various elements identified in the logical, process, and development views-networks, processes, tasks, and objects-must be mapped onto the various nodes.

- It takes into account the system's nonfunctional requirements such as system availability, reliability (fault-tolerance), performance (throughput), and scalability.

4 + 1: Development View



- The development view focuses on the organization of the actual software modules in the software-development environment.

- The software is packaged in small chunks-program libraries or subsystems-that can be developed by one or more developers.

Development View: Notation

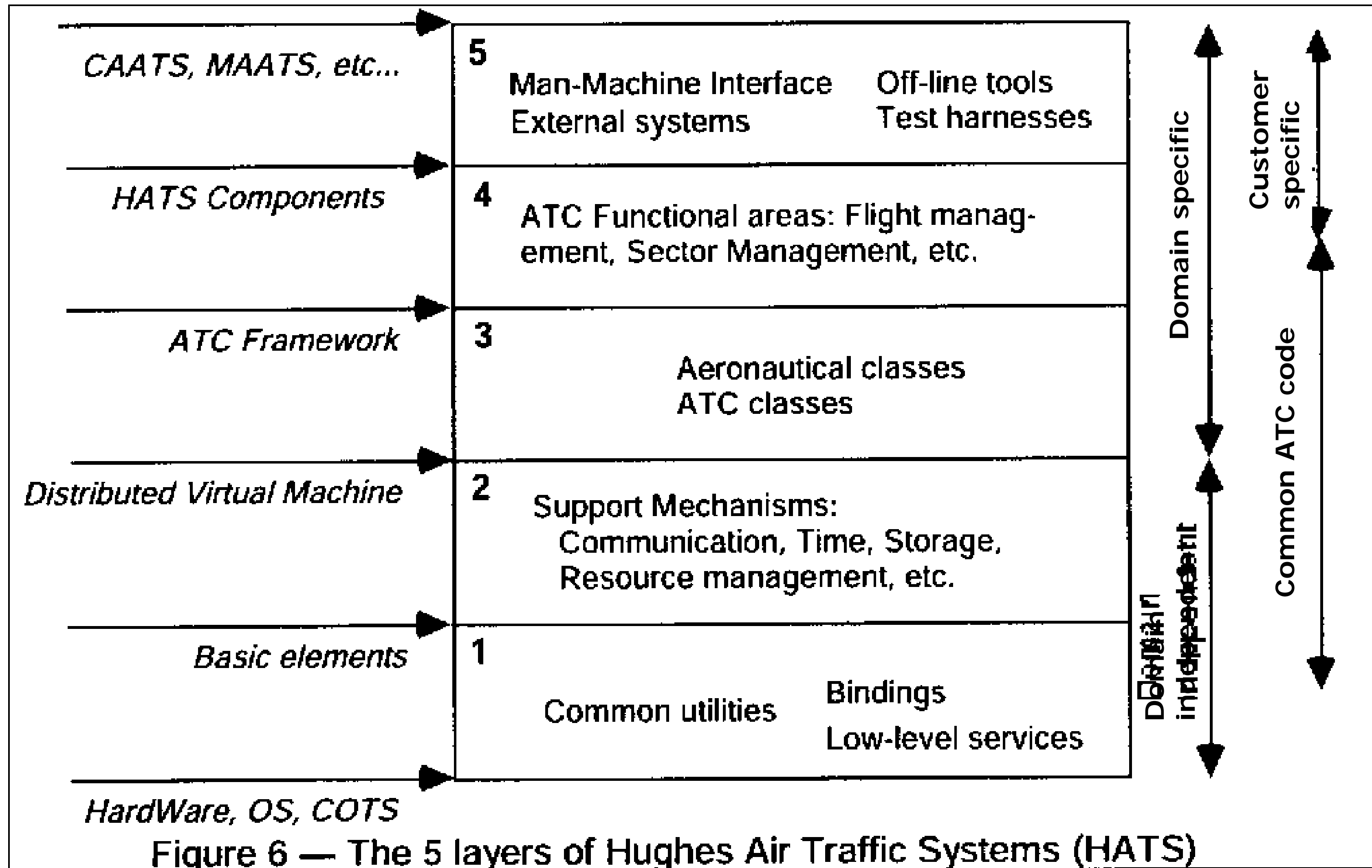
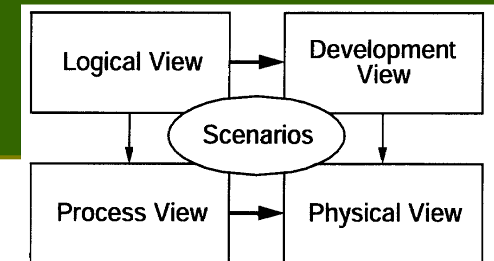


Figure 6 — The 5 layers of Hughes Air Traffic Systems (HATS)

4 + 1: Scenario View



- The scenario view consists of a small subset of important scenarios (e.g., use cases) to show that the elements of the four views work together seamlessly.

- This view is redundant with the other ones (hence the "+1"), but it plays two critical roles:
  - it acts as a driver to help designers discover architectural elements during the architecture design;
  - it validates and illustrates the architecture design, both on paper and as the starting point for the tests of an architectural prototype.

Scenario View

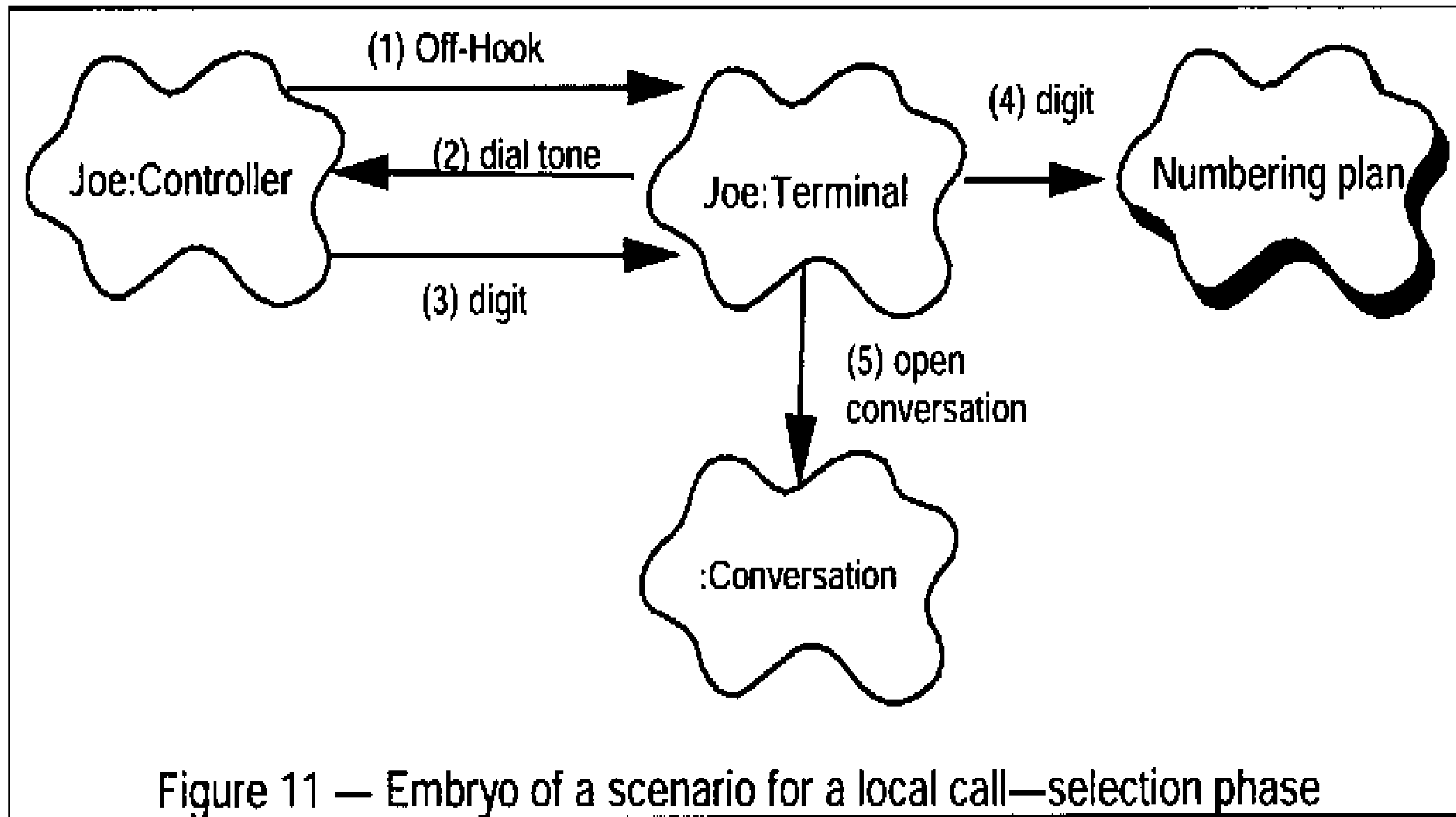


Figure 11 — Embryo of a scenario for a local call—selection phase

Relating Structures and Quality Attributes with Viewpoints

	Logical	Process	Physical	Development
Structure				
Module				✓
Conceptual (logical)	✓			
Process		✓		
Physical			✓	
Uses				✓
Calls	✓	✓		
Data flow		✓		
Control flow	✓	✓		
Class	✓			
Quality Attributes				
Performance		✓	✓	
Security			✓	
Availability			✓	
Functionality	✓			
Usability	✓	✓		
Modifiability	✓			✓
Portability	✓	✓	✓	✓
Reusability	✓	✓		✓
Integrability				✓
Testability	✓	✓		✓

Design patterns

- Vehicle for reasoning about design or architecture at a higher level of abstraction (design confidence)

- Patterns == Problem/Solution pairs
  in a given context

Note: the words style and pattern are sometimes used interchangeably…

Design Patterns

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
  - Christopher Alexander, A Pattern Language: Towns/Buildings/Construction, 1977
- An OO design pattern systematically names, explains and evaluates an important and recurring design in OO systems

A Good Pattern

- Why patterns?
- A pattern:
  - solves a problem
  - is a proven concept
  - solution isn’t obvious
  - has a significant human component
- Patterns aren’t written – they’re discovered!

A Good Pattern

- Essential components of a pattern format
  - Name
  - Problem, context
  - Solution, examples
  - Consequences, rationale, related patterns, known uses
- Properties of patterns:
  - Encapsulation and abstraction
  - Openness and variability

Classes of patterns

- Creational patterns:
  - Deal with initializing and configuring classes and objects
  - *Abstract factory* – factory for building related objects
- Structural patterns:
  - Deal with decoupling interface and implementation of classes and objects
  - *Adapter* – translator adapts a server interface for a client
- Behavioural patterns:
  - Deal with dynamic interactions between societies of classes and objects
  - *Iterator* – aggregated elements are accessed sequentially

Design Pattern Template

- **Intent**: short description of patten and its purpose
- **Also known as**: other names for pattern
- **Motivation**: motivation scenario showing pattern's use
- **Applicability**: circumstances in which pattern applies
- **Structure**: graphical representation of the pattern
- **Participants**: participating classes and/or objects and their responsibilities

Template cont.

- **Collaborations**: how participants co-operate to carry out responsibilities
- **Consequences**: the results of application, benefits and liabilities
- **Implementation**: pitfalls, hints or techniques, plus language dependency
- **Sample code**: example implementations in OO language
- **Know uses**: examples drawn from existing systems
- **Related patterns**: discussion of other patterns that relate to this one

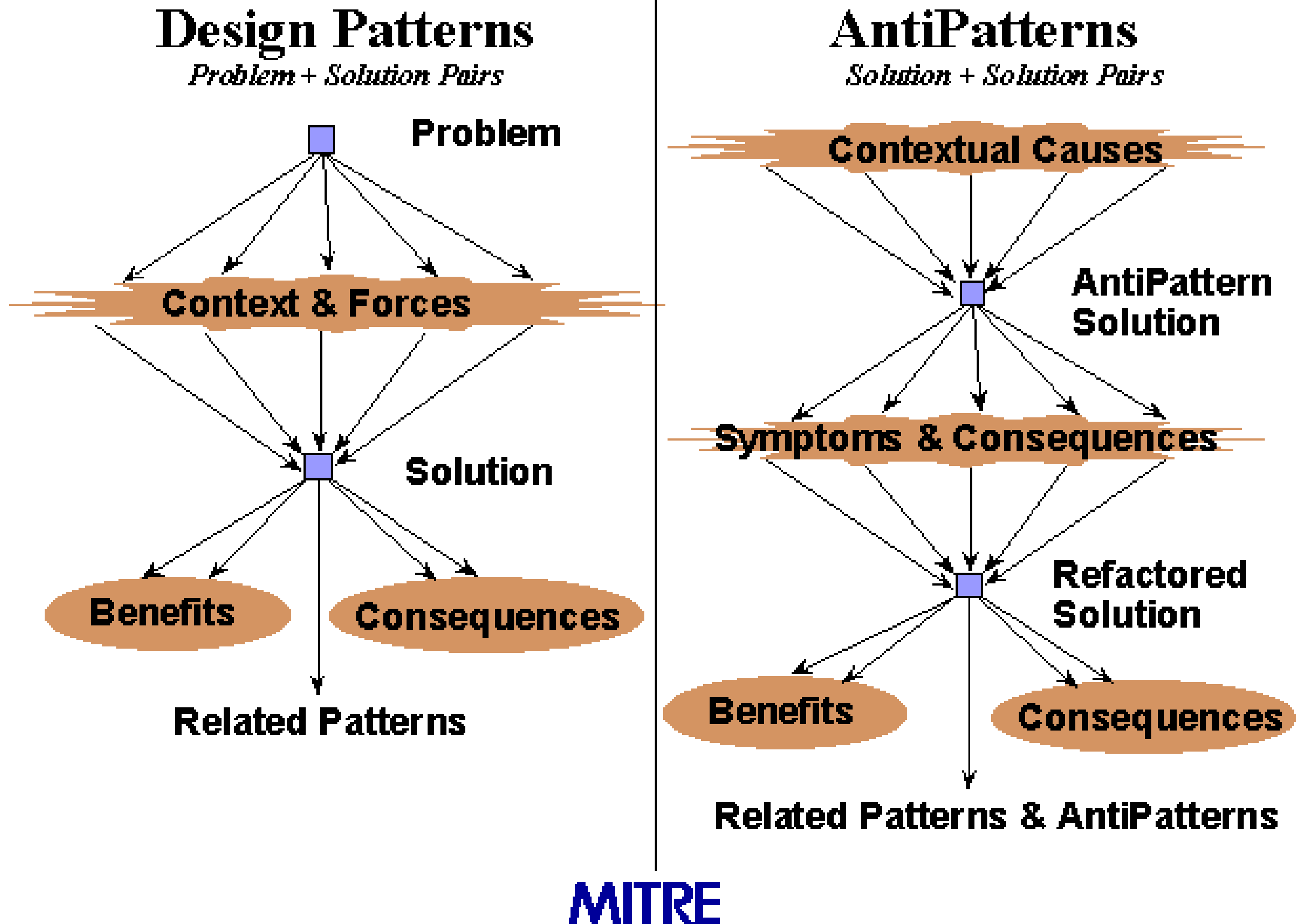
Observer Pattern

- **Intent**: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Key forces:
  - There may be many observers
  - Each may react differently to the same notification
  - The subject should be decoupled from the observers so that the observers can be changed independently of the subject

When Not to Use Patterns…

- When the solution is already obvious…
- When the use of the pattern might be overkill (although what be obvious to one person may not be to another…)
- If it is not detailed enough… but they can act as a bridge….

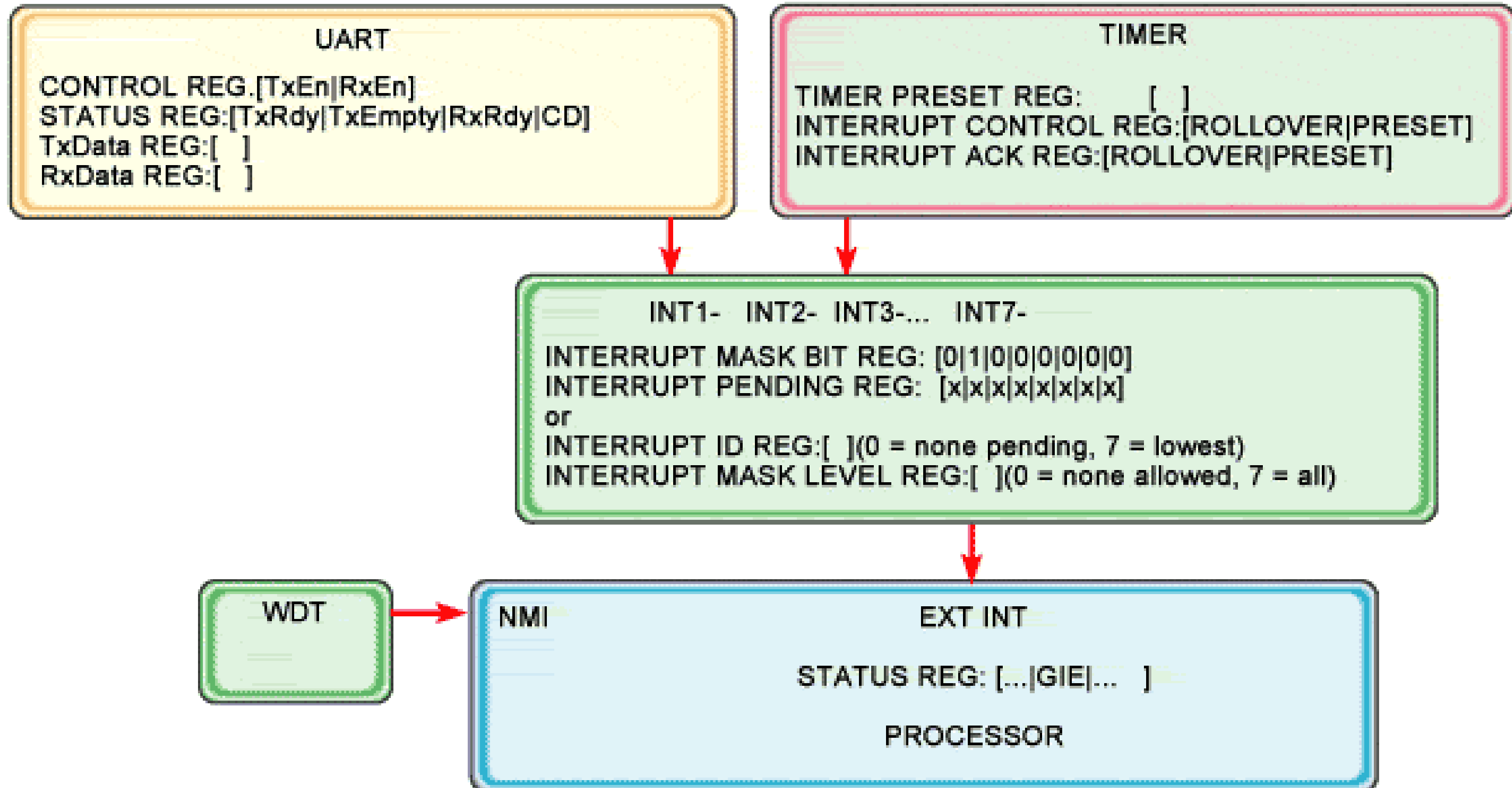
AntiPatterns



Introduction to Interrupts

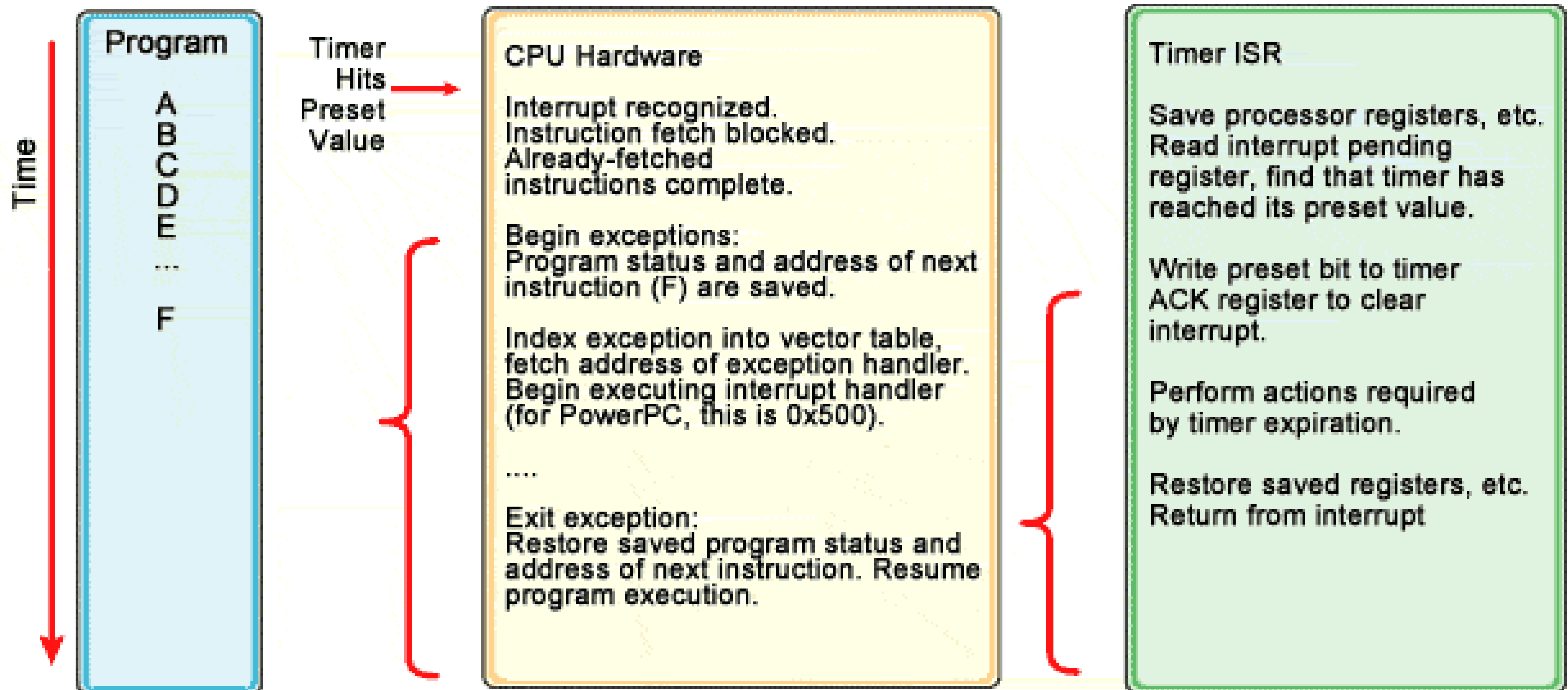
- Breaks in program flow
  - Exceptions and traps: predictable, synchronous breaks in program flow
  - Interrupts: asynchronous breaks in program flow that occurs as a result of events outside the running program
- An interrupt is a signal that causes the main program that operates the computer (the *operating system*) to stop and figure out what to do next.

Interrupt HW Model



http://www.embedded.com/story/OEG20010518S0075

Interrupt Processing



http://www.embedded.com/story/OEG20010518S0075

Domain-specific Architectures

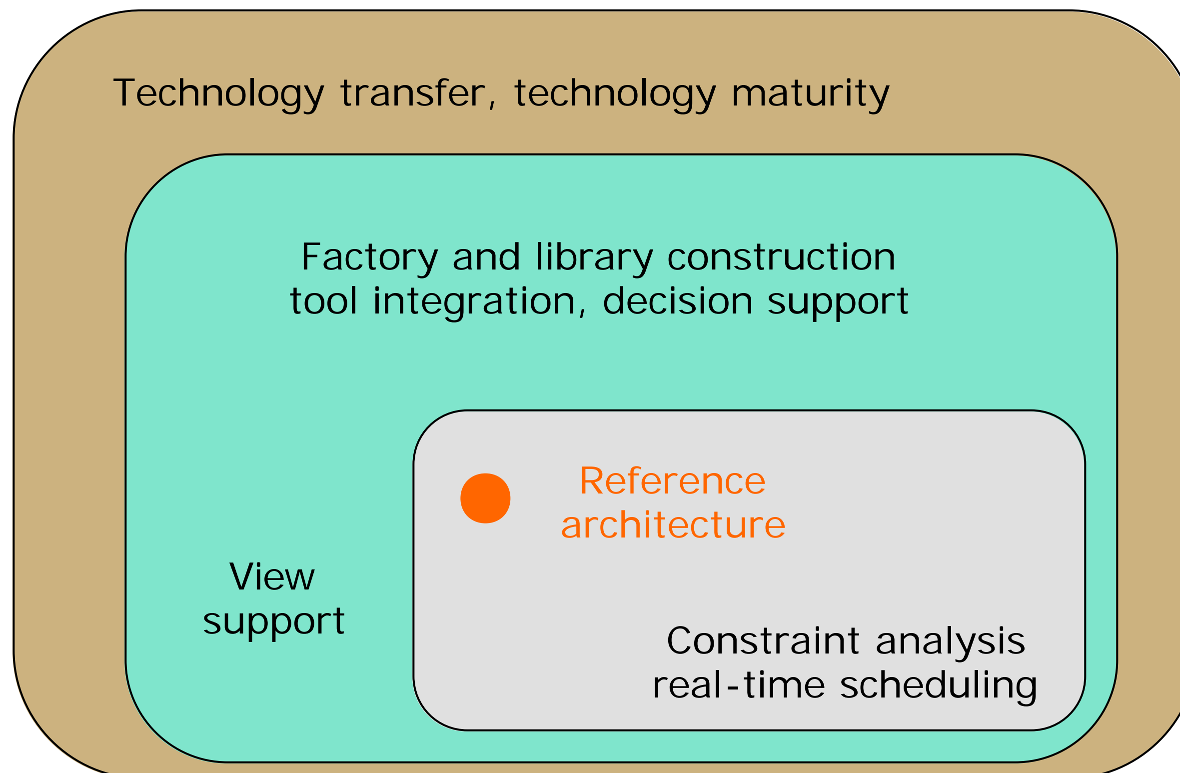
- Architectural models which are specific to some application domain
- Two types of domain-specific model
  - **Generic models** which are abstractions from a number of real systems and which encapsulate the principal characteristics of these systems
  - **Reference models** which are more abstract, idealised model. Provide a means of information about that class of system and of comparing different architectures
- Generic models are usually bottom-up models; Reference models are top-down models

Reference Architectures

- Reference models are derived from a study of the application domain rather than from existing systems
- May be used as a basis for system implementation or to compare different systems. It acts as a standard against which systems can be evaluated
- ADAGE: project to define and build a domain-specific SW architecture environment for assisting the development of avionics SW
  - Avionics Domain Application Generation Environment

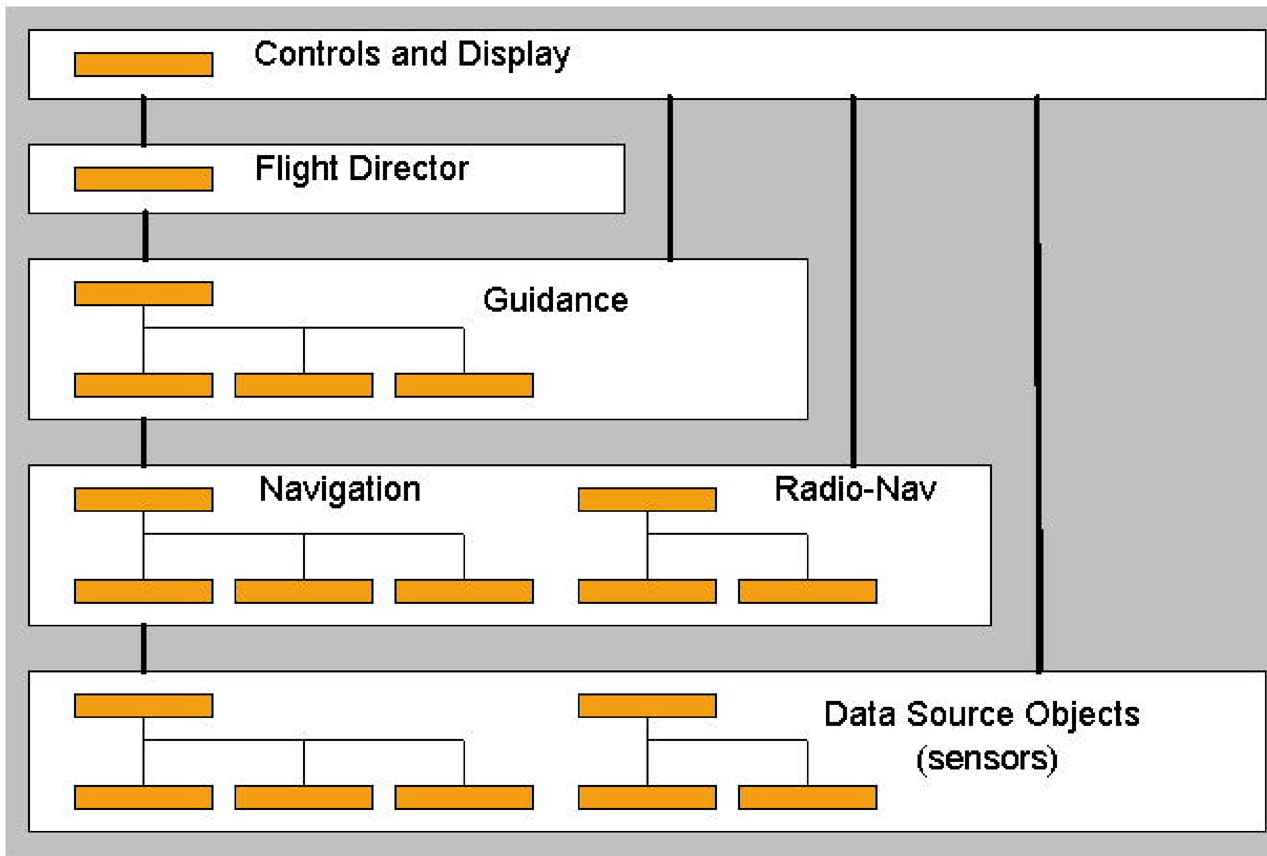
ADAGE

Integrated environment for exploring, evaluating, and synthesizing different avionics software architectures



Reference Architecture: An Example for Avionics

Reference architecture is defined by component realms and domain-specific composition constraints



Even simple avionics systems often require over 50 distinct components stacked 15 layers deep