Designing Architectures

Software Architecture

How Do You Design?

Where do architectures come from?

Creativity

1) Fun!
2) Fraught with peril
3) May be unnecessary
4) May yield the best result

Method

1) Efficient in familiar terrain
2) Not always successful
3) Predictable outcome (+ & -)
4) Quality of methods varies

Objectives

- Creativity
  - Enhance your skill-set
  - Provide new tools
- Method
  - Focus on highly effective techniques
- Develop judgment: when to develop novel solutions, and when to follow established method

Engineering Design Process

- **Feasibility stage**: identifying a set of feasible concepts for the design as a whole.
- **Preliminary design stage**: selection and development of the best concept.
- **Detailed design stage**: development of engineering descriptions of the concept. (work in parallel)
- **Planning stage**: evaluating and altering the concept to suit the requirements of production, distribution, consumption and product retirement. (work in parallel)

Potential Problems

- If the designer is unable to produce a set of feasible concepts, progress stops.
- As problems and products increase in size and complexity, the probability that any one individual can successfully perform the first steps decreases.
- The standard approach does not directly address the situation where system design is at stake, i.e. when relationship between a set of products is at issue.
- As complexity increases or the experience of the designer is not sufficient, alternative approaches to the design process must be adopted.

Alternative Design Strategies

- Standard

  Linear model described above
- Cyclic

  Process can revert to an earlier stage
- Parallel

  Independent alternatives are explored in parallel
- Adaptive ("lay tracks as you go")

  The next design strategy of the design activity is decided at the end of a given stage
- Incremental

  Each stage of development is treated as a task of incrementally improving the existing design

Identifying a Viable Strategy

万事开头难: how to identify that set of viable arrangement

- Use fundamental design tools: abstraction and modularity.

  But how?

- Inspiration, where inspiration is needed. Predictable techniques elsewhere.

  But where is creativity required?

- Applying own experience or experience of others.

The Tools of "Software Engineering 101"

- Abstraction

  Abstraction(1): look at details, and abstract "up" to concepts

  Abstraction(2): choose concepts, then add detailed substructure, and move "down"

  - Example: design of a stack class

- Separation of concerns

A Few Definitions… from the *OED Online*

- Abstraction: "The act or process of separating in thought, of considering a thing independently of its associations; or a substance independently of its attributes; or an attribute or quality independently of the substance to which it belongs."
- Reification: "The mental conversion of … [an] abstract concept into a thing."
- Deduction: "The process of drawing a conclusion from a principle already known or assumed; spec. in Logic, inference by reasoning from generals to particulars; opposed to INDUCTION."
- Induction: "The process of inferring a general law or principle from the observation of particular instances (opposed to DEDUCTION, q.v.)."

Abstraction and the Simple Machines

- What concepts should be chosen at the outset of a design task?

  One technique: Search for a "simple machine" that serves as an abstraction of a potential system that will perform the required task

  For instance, what kind of simple machine makes a software system embedded in a fax machine?

  - At core, it is basically just a little state machine.
- Simple machines provide a plausible first conception of how an application might be built.
- Every application domain has its common simple machines. (experience)

Simple Machines

Domain	Simple Machines
Graphics	Pixel arrays Transformation matrices Widgets Abstract depiction graphs
Word processing	Structured documents Layouts
Process control	Finite state machines
Income Tax Software	Hypertext Spreadsheets Form templates
Web pages	Hypertext Composite documents
Scientific computing	Matrices Mathematical functions
Banking	Spreadsheets Databases Transactions

Choosing the Level and Terms of Discourse

- Any attempt to use abstraction as a tool must choose a level of discourse, and once that is chosen, must choose the terms of discourse.
- *Alternative 1*: initial level of discourse is one of the application as a whole (step-wise refinement).
- *Alternative 2:* work, initially, at a level lower than that of the whole application.

  Once several such **sub-problems** are solved they can be **composed** together to form an overall solution
- *Alternative 3*: work, initially, at a level above that of the desired application.

  E.g. handling simple application input with a **general** parser.

Separation of Concerns

- Separation of concerns is the subdivision of a problem into (hopefully) independent parts.
- The difficulties arise when the issues are either actually or apparently intertwined.
- Separations of concerns frequently involves many **tradeoffs**
- Total independence of concepts may not be possible.
- Key example from software architecture: separation of components (computation) from connectors (communication)

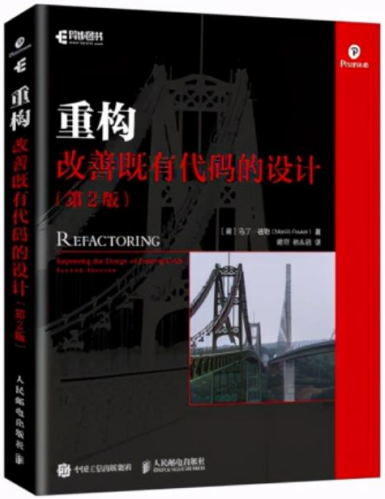
The Grand Tool: Refined Experience

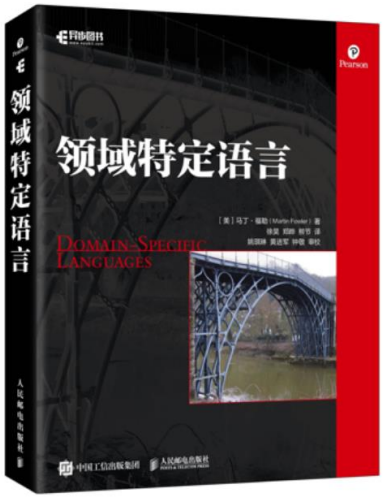
- **Experience** must be reflected upon and refined.
- The **lessons** from prior work include not only the lessons of successes, but also the lessons arising from failure.
- Learn from success and failure of other engineers
  - Literature
  - Conferences
- Experience can provide that initial feasible set of "alternative arrangements for the design as a whole".

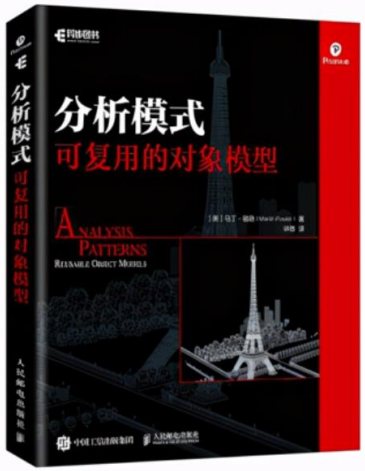
The Grand Tool: Refined Experience

马丁•福勒（Martin Fowler）

敏捷开发方法论
企业应用架构模式
微服务架构
重构





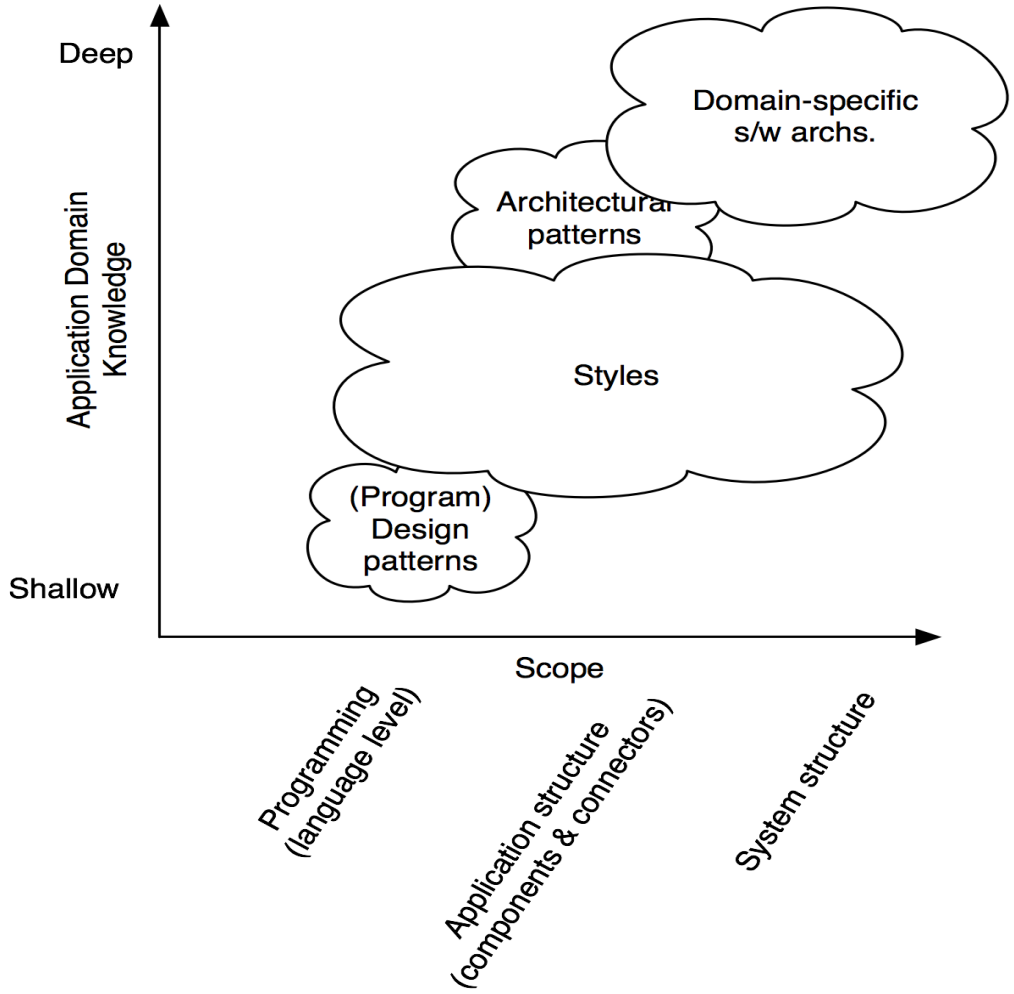


The Grand Tool: Refined Experience

极限编程
软件设计原则
Clean code(r)

Robert C. Martin
（罗伯特·C·马丁 ）

Patterns, Styles, and DSSAs



Domain-Specific Software Architectures

- DSSAs encode substantial knowledge, acquired through extensive experience, about how to structure complete applications within a particular domain.
- A useful operational definition of DSSAs is the combination of
  - (1) a reference architecture for an application
  - (2) a library of software components for that architecture containing reusable chunks of domain expertise,
  - (3) a method of choosing and configuring components to work within an instance of the reference architecture.
- a DSSA represents the most valuable type of experience useful in identifying a feasible set of "alternative arrangements for the design as a whole.

Domain-Specific Software Architectures

- A DSSA is an assemblage of software components
  specialized for a particular type of task (domain),
  generalized for effective use across that domain, and
  composed in a standardized structure (topology) effective for building successful applications.
- Since DSSAs are specialized for a particular domain they are only of value if one exists for the domain wherein the engineer is tasked with building a new application.
- DSSAs are the preeminent means for maximal **reuse** of knowledge and prior development and hence for developing a new architectural design.

Domain-Specific Software Architectures

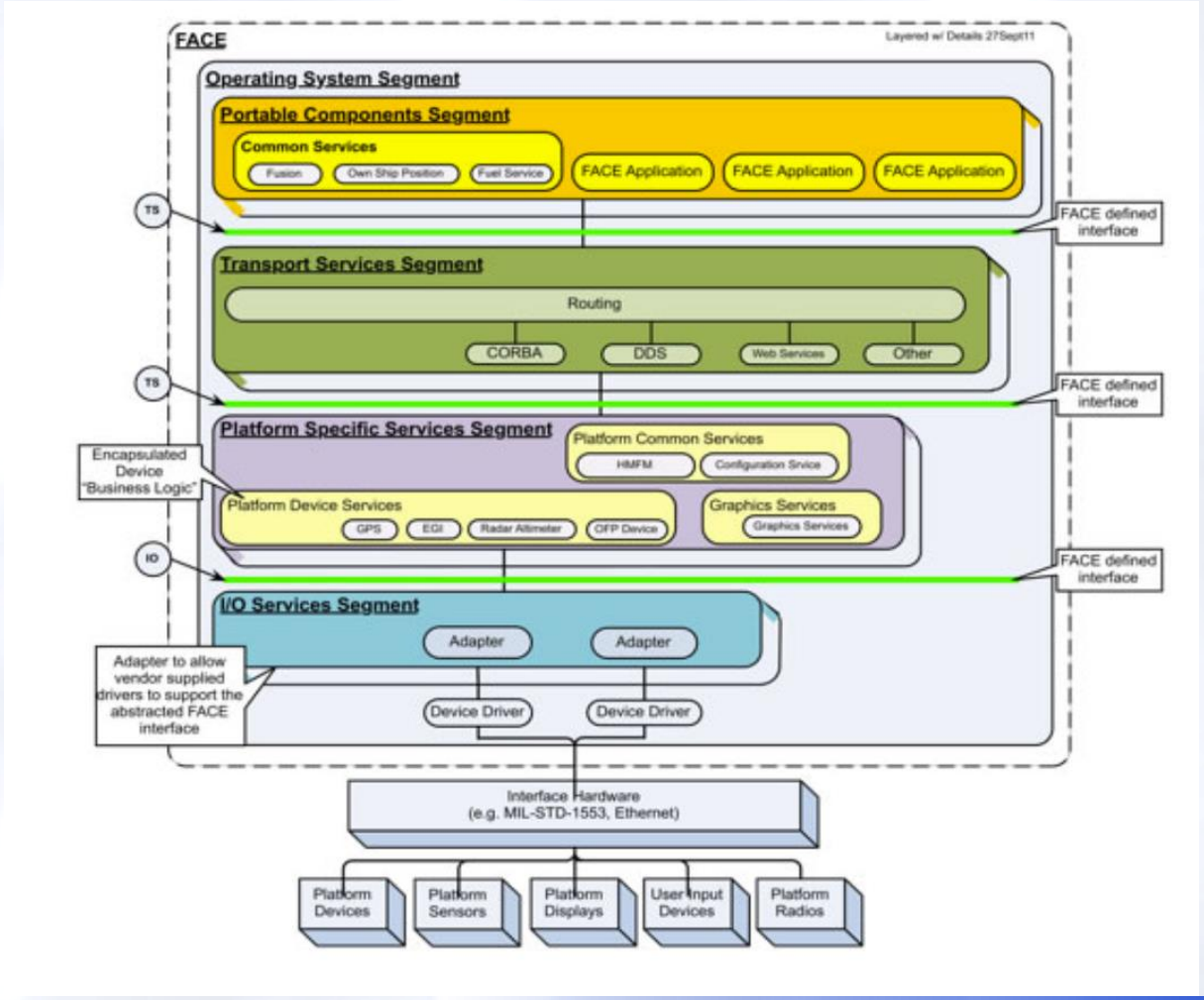
- Eg. FACE

美国国防部正在持续推动**开放式架构解决方案的应用**，以便让更好的航空电子硬件更加快速低成本地进入该领域。**未来机载能力环境（FACE）**联盟就是此类解决方案的提供者之一。该联盟成立于2010年，旨在开发一套开放架构的技术标准以及用于在系统中实施FACE标准的商业模型。该联盟由开放集团（Open Group）管理，成员来自政府、教育和工业领域。其会员单位必须在设立于美国，而个人会员则必须是美国公民。

FACE技术标准是开放式的航空电子标准，旨在使军事计算系统运作更加稳健、更具互操作性、更加便携以及更为安全。该标准使开发人员可以通过一个通用的操作环境来创建和部署门类广泛的应用系统，以便应用于整个军用航空系统。

FACE标准的概念旨在提供一种基于片段的参考架构，这些片段可以组合起来以满足最终系统的需求。各层内容的变化，包括应用代码的变化，使系统设计师可以灵活地设计和构建最终系统。FACE在这些层之间提供逻辑接口，以实现便携性和重用性。

Domain-Specific Software Architectures



Architectural Patterns

- An architectural pattern is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears.
- Architectural patterns are similar to DSSAs but applied "at a lower level" and within a much narrower scope.

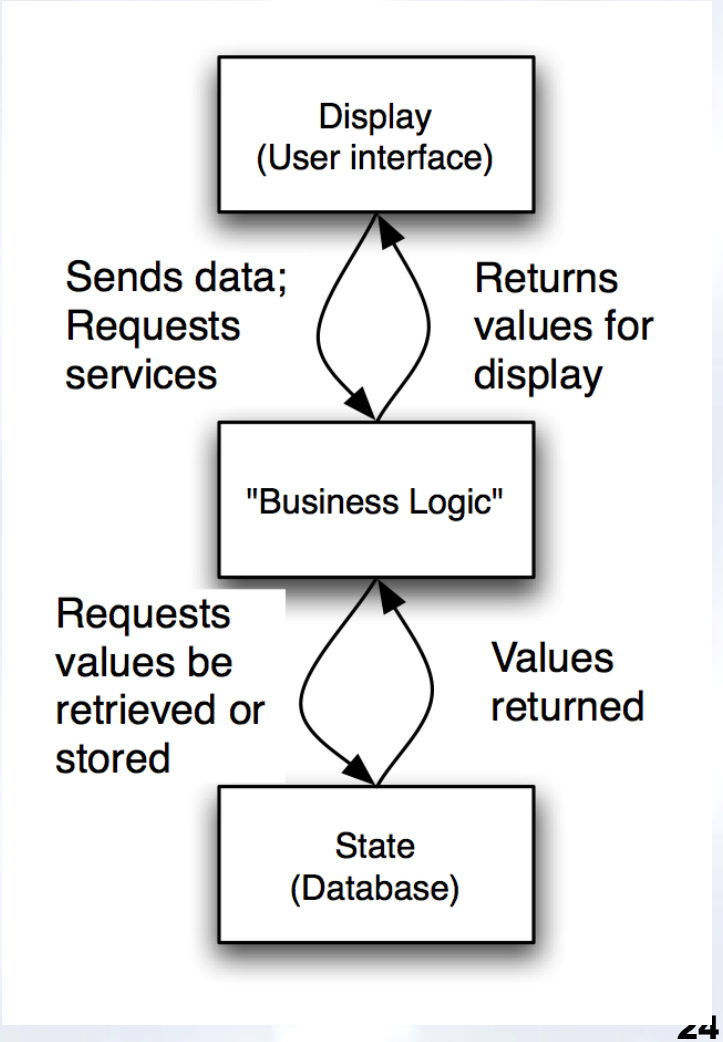
Architectural Patterns

- State-Logic-Display: three-tiered pattern
- Model-View-Controller (MVC)
- Sense-Compute-Control

State-Logic-Display: Three-Tiered Pattern

- Application Examples
  - Business applications
  - Multi-player games
  - Web-based applications

- *business apps: the data store is usually a large database server.*
- *Multi-player games: each player has his own display component.*
- *Web-based apps: the display component is the user's Web browser*



Model-View-Controller (MVC)

- Objective: Separation between information, presentation and user interaction.
- When a model object value changes, a notification is sent to the view and to the controller.

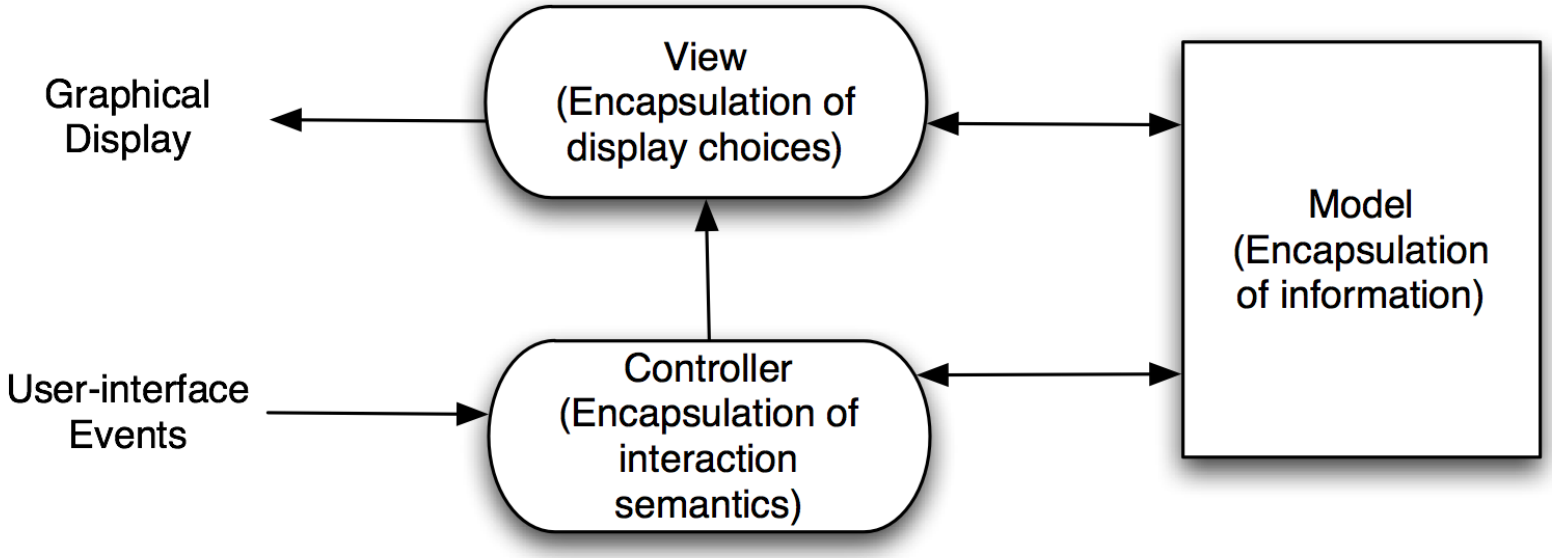
  Thus, the view can update itself and the controller can modify the view if its logic so requires.
- When handling input from the user the windowing system sends the user event to the controller.

  If a change is required, the controller updates the model object.

Model-View-Controller (MVC)

- Since its invention in the 1980s, the model-view-controller (MVC) pattern has been a dominant influence in the design of **graphical user interfaces**.
- Besides as a pattern, it can also be viewed as providing a structure for applications at a higher level.
- MVC promotes separation, and thus independent development paths, between information manipulated by a program and depictions of user interactions with that information.

Model-View-Controller



The model component encapsulates the information used by the application;

The view component encapsulates the information chosen and necessary for graphical depiction of that information;

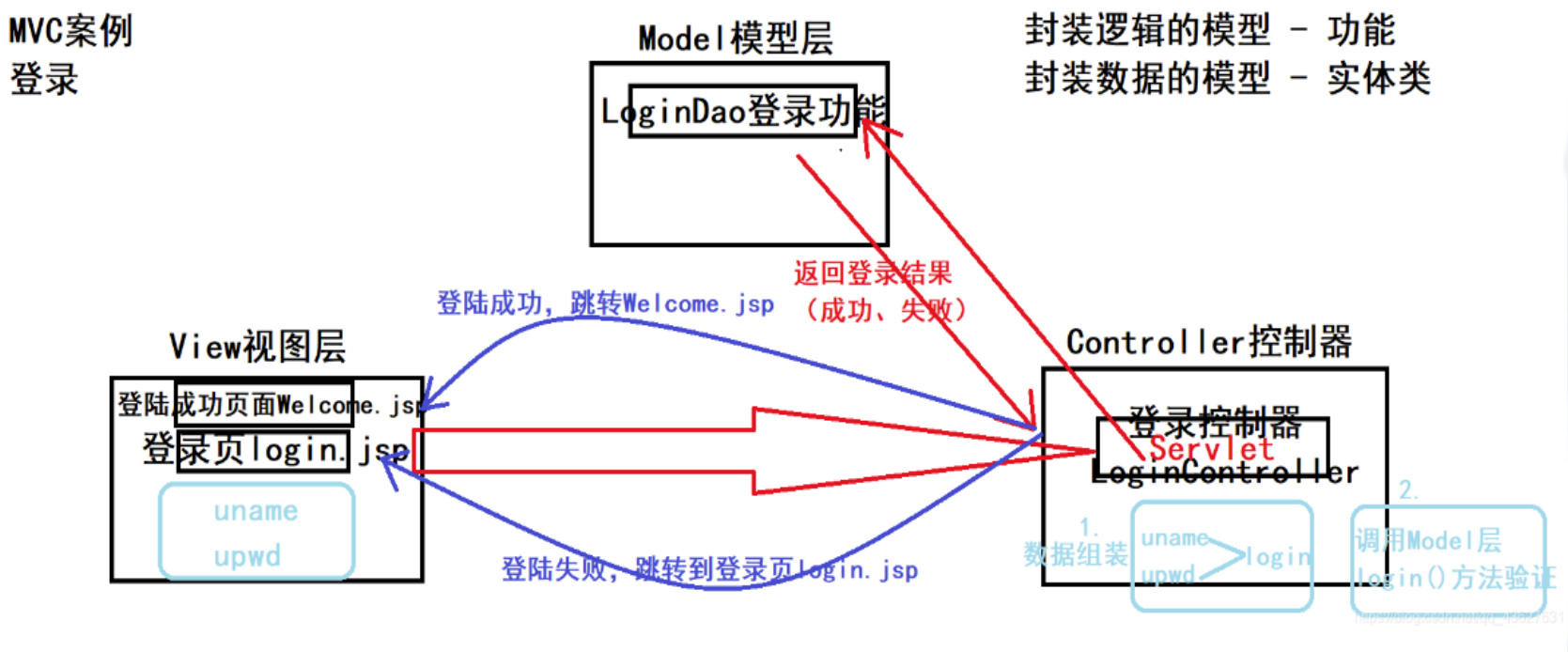
The controller component encapsulates the logic necessary to maintain consistency between the model and the view, and to handle inputs from the user as they relate to the depiction.

Model-View-Controller

In many cases, the view and controller components are merged; such is the case with the architecture of Java's Swing framework.

At a much higher level of abstraction than programing, the MVC paradigm can be seen at work in the World Wide Web.
Web resources correspond to the **model objects**;
the HTML rendering agent within a browser corresponds to **the viewer**;
the controller in the simplest case corresponds to the code that is part of the browser that responds to user input and which causes either interactions with a Web server or modifies the browser's display.

Model-View-Controller



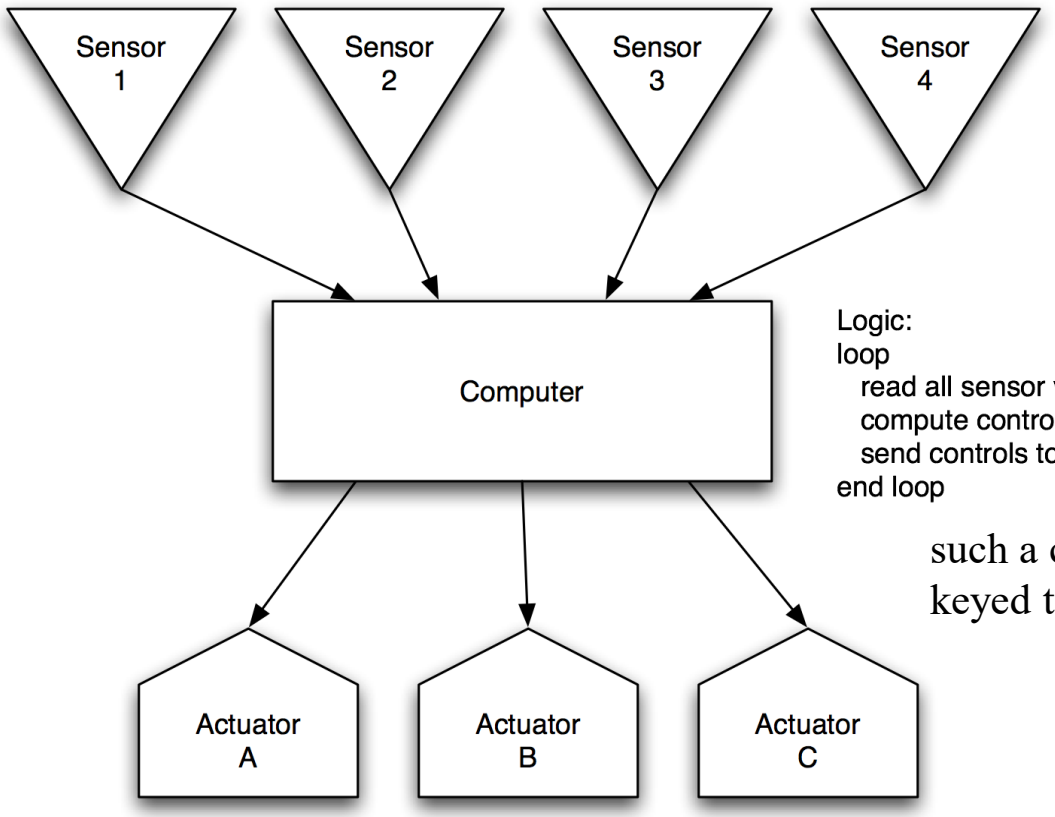
Model-View-Controller

Sense-Compute-Control

Kitchen appliance applications;

Automotive applications;

Robotic control.



Objective: Structuring embedded control applications

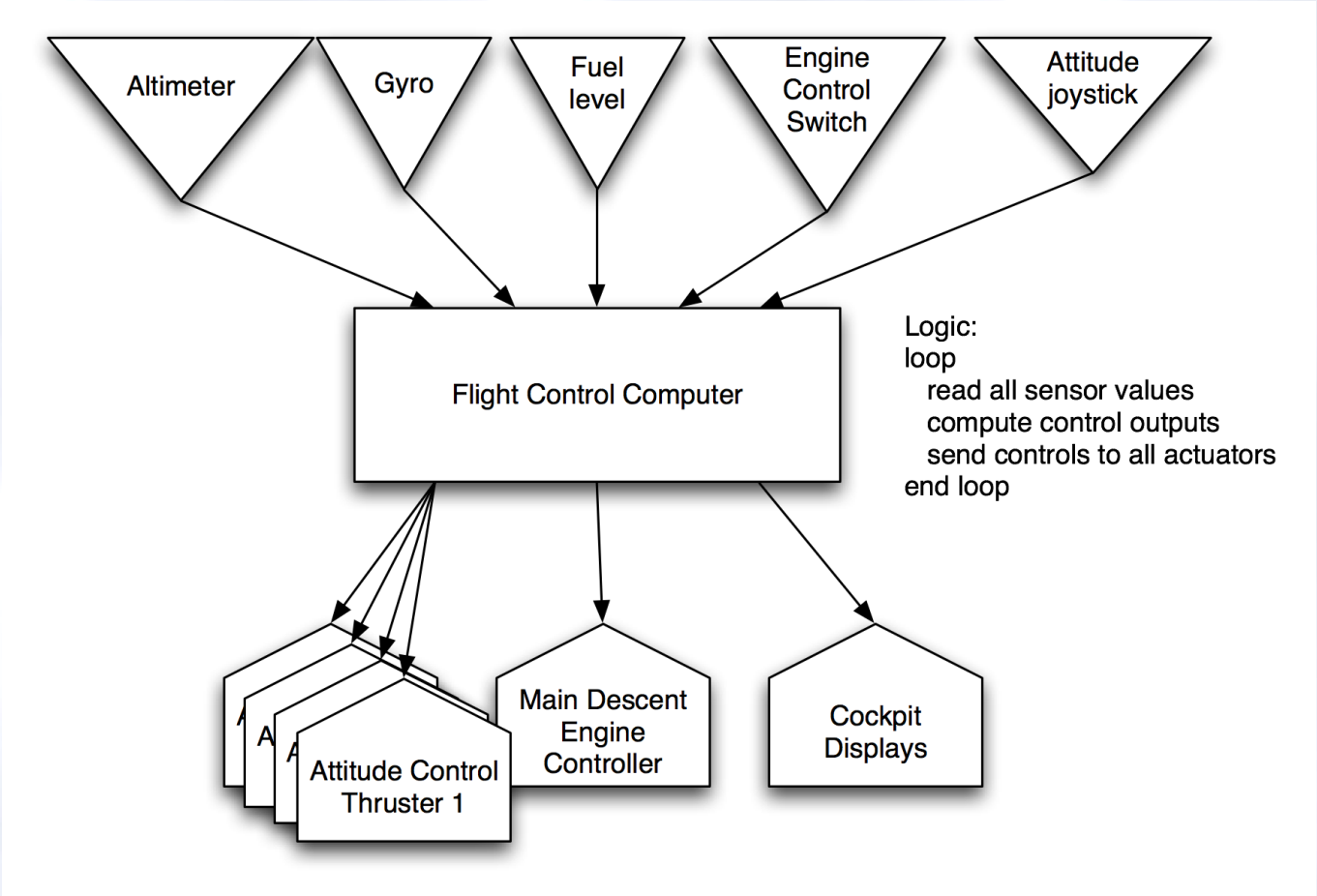
The Lunar Lander: A Long-Running Example

- A simple computer game that first appeared in the 1960's
- Simple concept:

  You (the pilot) control the descent rate of the Apollo-era Lunar Lander
  - Throttle setting controls descent engine
  - Limited fuel
  - Initial altitude and speed preset
  - If you land with a descent rate of < 5 fps: you win (whether there's fuel left or not)

  "Advanced" version: joystick controls attitude & horizontal motion

Sense-Compute-Control LL



Architectural Styles

- An architectural style is a named collection of architectural design decisions …
- A primary way of characterizing lessons from experience in software system design
- Reflect less domain specificity than architectural patterns
- Useful in determining everything from subroutine structure to top-level application structure
- Many styles exist and we have discussed them in detail in previous classes

Definitions of Architectural Style

- Definition. An architectural style is a named collection of architectural design decisions that
  - are applicable in a given development context
  - constrain architectural design decisions that are specific to a particular system within that context
  - elicit beneficial qualities in each resulting system.
- Recurring organizational patterns & idioms
  - Established, shared understanding of common design forms
  - Mark of mature engineering field.
    - **Shaw & Garlan**
- Abstraction of recurring composition & interaction characteristics in a set of architectures
    - **Medvidovic & Taylor**

Basic Properties of Styles

- A vocabulary of design elements
  - Component and connector types; data elements
  - e.g., pipes, filters, objects, servers
- A set of configuration rules
  - **Topological** constraints that determine allowed compositions of elements
  - e.g., a component may be connected to at most two other components
- **A semantic** interpretation
  - Compositions of design elements have well-defined meanings
- Possible analyses of systems built in a style

Benefits of Using Styles

- Design reuse
  - Well-understood solutions applied to new problems
- Code reuse
  - Shared implementations of invariant aspects of a style
- Understandability of system organization
  - A phrase such as "client-server" conveys a lot of information
- Interoperability
  - Supported by style standardization
- Style-specific analyses
  - Enabled by the **constrained design space**
- Visualizations
  - Style-specific depictions matching engineers' mental models

Style Analysis Dimensions

- What is the design vocabulary?
  Component and connector types
- What are the allowable structural patterns?
- What is the underlying computational model?
- What are the essential invariants of the style?
- What are common examples of its use?
- What are the (dis)advantages of using the style?
- What are the style's specializations?