COMP354

Software Engineering

Lecture 13

Software Architectures

Software Architecture

Basics — do not forget these!

encapsulation, information hiding

decomposition into subsystems, modules

layers

client-supplier relationship

clear clean interfaces

hierarchical uses relationship in module diagram

Architecture looks at **structural** issues

- organization into subsystems and modules
- assignment of functionality to components
- distribution of control
- protocols for communication, synchronization, and data access
- physical allocation of components to processors

Using Known Architectures

There are several well-known architectures

  eg, interactive interface, continuous transformation

One or more may match your system requirements.

  see **Characteristics**

Select an appropriate architecture

  eg, interactive interface

Focus the architecture to your context

  physical events $\rightarrow$ mouse clicks, keys

  physical events $\rightarrow$ draw, dialog box, file write

  user interface $\rightarrow$ spreadsheet UI

  rest of system $\rightarrow$ spreadsheet model subsystem

  functionality $\rightarrow$ update and re-calculation

Concentrate on the *issues* important to that architecture

  understanding user protocol

  event handler to isolate UI from spreadhseet model

The architecture will give general guidelines on how to *resolve* the issues

  see **Steps in Designing ...**

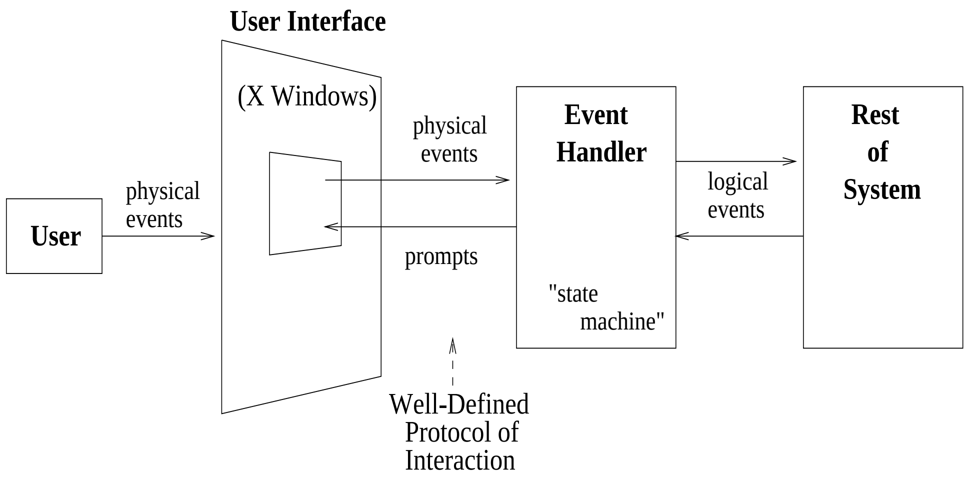
Interactive Interface Architecture

Characterisation

An interactive system is dominated by the interactions with external agents, such as users, sensors, and IO devices, over which it has no control.
Events and interactions occur independently of the system, (but they may be in response to system prompts).

Design Principles

Isolate the interactive interface from rest of the system.
Understand the protocol of the interaction.
The dynamic model of behaviour is very important.



Steps in Designing an Interactive Interface

- Isolate the objects which form the interface from the objects that define the application semantics.
- Use predefined objects to interact with external agents, if possible. eg, a toolbox like XR11.
- Use the dynamic model to structure the program. Use concurrent control (multi-tasking), or use event-driven control (interrupts or callbacks).
- Isolate physical events from logical events.
- Fully specify the application functions invoked by the interface. Make sure the information to implement them is present.

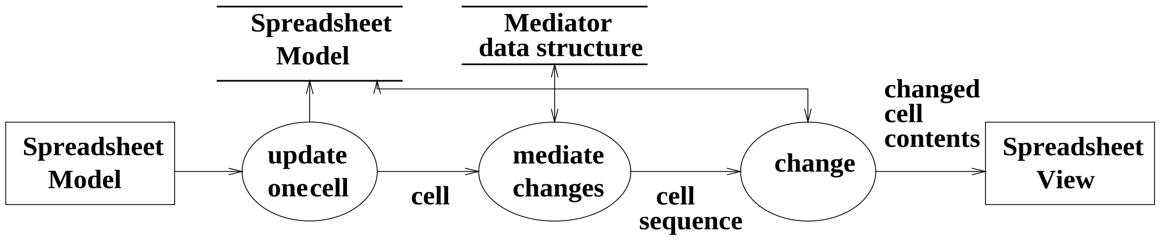
Continuous Transformation Architecture

Characterisation

A continuous transformation system has outputs that actively depend on changing inputs, and severe time constraints means that the complete output cannot be recomputed each time the input changes.
New output values must be computed incrementally.

Design Principles

Functional model, together with object model, defines the values being computed.
Transformation can be implemented as a pipeline.
Input increments propagate down pipeline.
Incremental change needs intermediate objects holding intermediate values.
Redundant values may be introduced for performance reasons.
Synchronization of pipeline may be important for high-performance systems.



Steps in Designing a Continuous Transformation

- Draw data flow diagram. Input and output actors correspond to data structures whose values change continuously. Data stores show parameters that affect input-to-output mapping.
- Define intermediate objects between pipeline stages.
- Define incremental version of each pipeline stage, to be able to propagate incremental changes
- Optimize with additional intermediate objects.

Dynamic Simulation Architecture

Characterisation

A dynamic simulation models or tracks objects in the real world.
Events and interactions may result from the behaviour of the objects in the simulation model, as well as there being user interactions.

Design Principles

Identify and understand the behaviour of the real-world object in the simulation model.
Control can be either by an explict external controller, or by message passing between the active objects in the simulation model.
The dynamic model of behaviour is very important.
There may be complex functional model too.

Steps in Designing an Dynamic Simulation

- Identify the active real-world objects in the simulation model. These have attributes that are periodically updated.
- Identify the discrete events, and the corresponding interactions. Discrete events can be implemented as operations on the objects.
- Identify continuous dependencies, and model these approximately as discrete incremental upadtes.
- Generally, there is a timing loop at a fine time scale, that manages events.

Architectural Styles

An *architectural style* describes a family of architectures.

An architectural style is described in terms of

- components
- connectors
- configurations
- ports
- roles

Example: Pipe-and-Filters
unix processes, like `grep` `sort` are components
unix pipes are connectors
may only allow linear configuration of pipeline
`stdin` and `stdout` are the ports
source, sink, filter are roles

A description of an architectural style provides

- a *vocabulary* of the basic design elements (components and connector types),
- a set of *configuration rules* which constrain how components and connectors may be configured,
- a *semantic interpretation* which defines when suitably configured designs have a well-defined meaning as an architecture, and
- *analyses* that may be performed on well-defined designs.

Examples of Architectural Styles

- *Batch Transformation* architectures transform the entire input data set once.
- *Continuous Transformation* architectures transform the input data continuously in response to incremental changes in the input.
- *Interactive Interface* architectures are dominated by external interactions.
- *Dynamic Simulation* architectures simulate evolving real-world objects.
- *Real-time System* architectures are dominated by strict timing constraints.
- *Transaction Manager* architectures process transactions on data stores that are being accessed in a concurrent and distributed fashion.
- Hybrid systems combine architectural styles.

Shaw and Garlan

- *Pipe-and-Filters* architectures, like that supported by the Unix shell, connect filters in a linear fashion. Each filter has one stream of inputs and one stream of outputs.
- *Data Abstraction* and *Object-Oriented Organization* architectures promote the decomposition of the system into entities (data type variables or objects) that encapsulate their implementation details and present an interface that completely describes their behaviour or functionality.
- *Event-based, Implicit Invocation* architectures are based on componen ts that register an interest in (a class of) events. The components are invoked in response to the occurrence of an event implicitly rather than being called directly by another component.
- *Layered System* architectures are organized as a hierarchy of layers,each layer providing services to the layers above it and each layer being a client for the services provided by layers below it.
- *Repository* architectures have distinct components for acentral data store (the repository) and those components which operate upon the repository.
- *Table Driven Interpreter* architectures implement a software virtual machine (the interpreter) by separating the interpretation engine from a table that describes the machine behaviour.
- Heterogeneous architectures combine architectural styles.

Buschmann et al

- Structural
  - Layers
  - Pipes-and-Filters
  - Blackboard
- Distributed Systems
  - Brokers
- Interactive Systems
  - Model-View-Controller
  - Presentation-Abstraction-Control
- Adaptable Systems
  - Microkernel
  - Reflection

Layered Architecture

The modules of the architecture are assigned to *layers* arranged from the *least* to the *most abstract*

— low layers close to machine-level abstractions,
high layers close to user-level abstractions —

A module may use only modules in its own layer and the layer immediately below.

Example: Client Server

Walkthrough for Object-Oriented Design Quality
Alistair Cockburn

1. Data Connectedness Test
Whether a (domain) model supports the use cases: can objects access the data they need.

2. Abstraction Test
Does the name of the object convey its abstraction?
Does the abstraction named have a natural meaning and use in the language of the experts?
Very many objects do not do well in this test.
It is highly subjective!

3. Responsibility Alignment Test
Do the name, main responsibility statement, and data and functions align?
During design evolution, usually the latter fattens up way past the point of what the name or primary responsibility call for.
That is often time to split the object, but sometimes it is time to rethink what abstraction you really have in front of you.

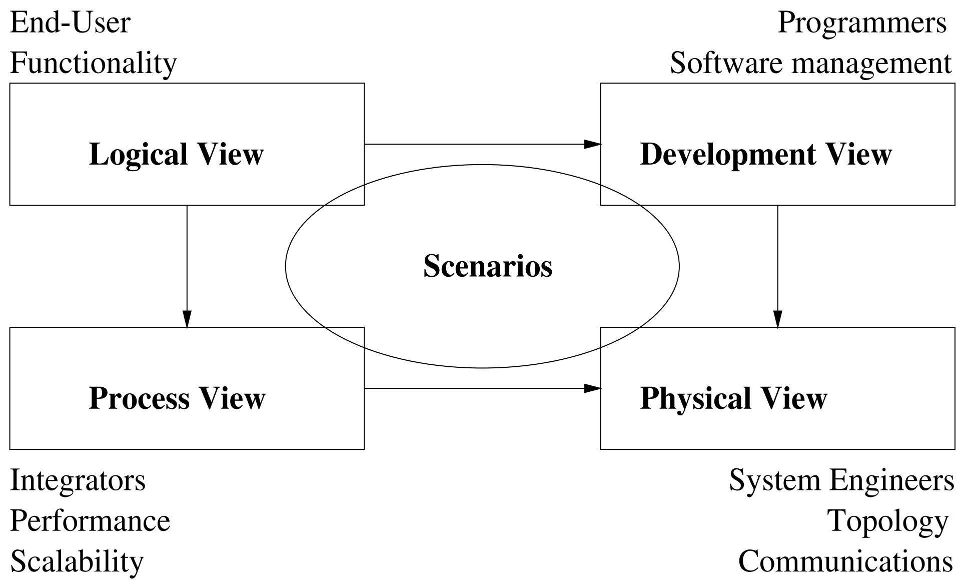
4. Variations Test Two parts:

Data Variations Test that the design naturally handles all the sorts and shapes of data the system will encounter.

Evolution Test What changes are likely in the business rules, assumptions, use cases, etc. How does the design handle these?

5. Communications Patterns Test
Run-time communications patterns. Particularly looks for cycles, but possibly other odd shapes.

Kruchten's 4+1 View Model



The *logical view* describes the design's object model.

The *process view* describes the design's concurrency and synchronization aspects.

The *physical view* describes the mapping of the software onto the hardware and reflects its distributed aspect.

The *development view* describes the software's static organization in its development environment.

The software designers illustrate their architectural decisions with a few selected *scenarios*.

The architecture is partially evolved from these scenarios.