**Sftwr Architecture Styles**

**What is Design?**

* Requirements specification was about WHAT the sys will do
* Design is about HOW the sys will perform funcs
  + Provides the overall decomposition of the sys
  + Allows to split the work among a team of developers
  + Lays down the groundwork for achieving nonfuncal requirements (performance, maintainability, reusability etc.)
  + Takes target technology into account (ex. kind of middleware, database design etc.)

**Levels of Design**

* Architectural design (high level design)
  + Architecture – the overall structure (main modules and the connections)
  + Design that covers the main use cases of the sys
  + Addresses the main nonfuncal requirements (ex. throughput, reliability)
  + Hard to change
* Detailed design (low level design)
  + Inner structure of main modules
  + May take the target programming language into account
  + Detailed enough to be implemented in the programming language

**Sftwr Architecture**

* Specification of the overall structure of the sys. This structure entails the syss components, basic characteristics of these components and the interaction b/w the components
* Provides a way to analyze and specify the sys at a higher level of abstraction
* Sftwr architecture specifications focus on the fundamental design decisions we take for a sys
* Abstractly, sftwr architecture involves the description of elements from which the syss are build, interactions among those elements, patterns that guide their composition and constrains on these patterns
  + In general, a particular sys is defined in terms of a collection of components and interactions among these components.
  + Such a sys may in turn be used as a (composite) element in a larger sys design
* Description of the subsyss and components of a sftwr sys and the relationships b/w them
  + Subsyss and components are typically specified in different views to show the relevant funcal and nonfuncal properties of a sftwr sys.
  + The sftwr architecture of a sys is an artifact. It is the result of the sftwr development activity

|  |  |
| --- | --- |
| **Issues addressed by an architectural design** | **Good properties of an architecture** |
| * Gross decomposition of a sys into interacting components   + Typically, hierarchical   + Using rich abstractions for “glue”   + Often using common design idioms/styles * Emergent sys properties   + Performance, throughput, latencies   + Reliability, security, fault tolerance, evolvability * Rationale   + Relates requirements and implementations * Envelop of allowed change   + “load bearing walls”   + Design idioms and styles | * Result of a consistent set of principles and techniques, applied consistently through all phases of a project * Resilient in the face of (inevitable) changes * Source of guidance throughout the produce lifetime * Reuse of established engineering knowledge |

**The need for multiple design views**

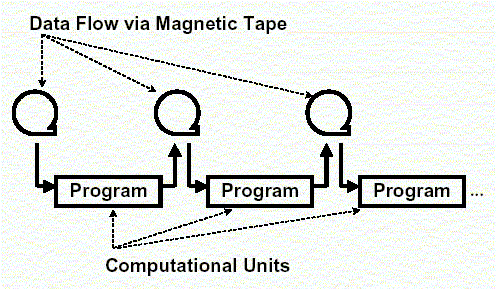
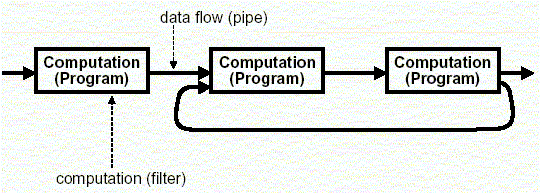
* Design is about sys structure
  + How the sys is decomposed into parts
  + Components and interactions w appropriate properties, enabling appropriate analyses
* But this begs the question: what kind of structure? – many possibilities
  + Code structure
  + Runtime structure
  + Process structure
  + Work breakdown structure
* Each of these can be the basis of a design view (or architecture view)

|  |  |
| --- | --- |
| **Kruchten’s “4+1 View Model” of the architecture** | **Architecture Style**   * A syss architecture style is composed of:   + **Components** Package and implement the syss funcs   + **Connectors** Implements the interconnections b/w the components * An architectural style defines a family of architectures which have   + Common topologies (how components are interconnected)   + Semantic constrains   + Common vocab for their components and their connectors |

**“Pure” Form of styles/ Architecture Styles and Patterns**

* When we define architecture style, we usually present its simple form
  + In practice, we rarely use a sftwr architecture style in its simple form. Combine appropriate architectural styles to design a new sys
* As new sftwr architects, we should have a good understanding of the different styles in their simple form, know the pros and cons of each style and understand what the impact of changes or modifications on an architecture style

**Data Flow Style**

* The availability of data controls the computation
* The structure of design is determined by the orderly motion of data from component to component
* The pattern of data control is explicit
* This is the only form of communication b/w components
* There are variety of variations on this general theme
  + How control is exerted (eg push vs pull)
  + Degree of concurrency b/w processes
  + Topology
* **Components**: data flow components
  + Interfaces are input and output ports
    - Input ports read data
    - Output ports write data
  + Computational model
    - Read data from input ports, computes, write data to output ports
* **Connectors** (Data Streams)
  + Uni-directional - Usually asynchronous, buffered
  + Interfaces are reader and writer roles
  + Computational model - Transport data from writer roles to reader roles
* Syss
  + Arbitrary graphs
  + Computational model – funcal composition
* Patterns in syss
  + Data flow can be arbitrary patterns
  + Primarily we are interested in linear data flow patterns
  + Or in simple, constrained cyclical patterns
* Kinds of data flow architectures
* **Batch sequential**
  + - **Components** (processing steps) are independent programs
    - **Connectors** are some type of media – traditionally magnetic tape
    - Each step runs to completion before the next step begins
    - History
      * Mainframes and magnetic tape
      * Limited disk space
      * Block scheduling of CPU processing time
    - Business data processing
      * Discrete transactions of predetermined type and occurring at periodic intervals
      * Creation of periodic reports based on data periodic data updates
    - Transformational data analysis
      * Raw data is gathered and analyzed into a step-wise, batch oriented fashion
    - Typical applications: non real time, batch oriented computations such as
      * Payroll computations
      * CRA tax return computations
* **Dataflow network (pipes & filters) – acyclic, fanout, pipeline, Unix, etc.**
  + - The tape of the batch sequence sys, morphed into a language and operating sys construct
    - Compared to the batch sequential style, data in the pipe & filter style is processed incrementally
    - the pipes and filters architectural pattern [style] provides a structure for syss that process a stream of data. Each processing step is encapsulated in a filter component. Data passed through pipes b/w adjacent filters. Recombining filters allows you to build families of related syss
    - **components** (filters)
      * read streams of data on input producing streams of data on output
      * local incremental transformation to input stream (eg filter, enrich, change representation etc.)
      * data is processed as it arrives, not gathered them processed
      * output usually begins before input is consumed
    - **connectors** (pipes)
      * conduits for streams (eg first in first out buffer)
      * transmit outputs from one filter to input of other
    - invariants
      * filters must be independent, no shared state
      * filters don’t know upstream or downstream filter identity
      * correctness of output from network must not depend on order in which individuals filters provide their incremental processing
    - common specializations
      * pipelines – linear sequence of filters
      * bounded and types pipes
    - exs of pipe and filter syss
      * lex/yacc-based compiler (scan, parse, generate code,..)
      * unix pipes
      * image processing
      * signal processing
      * voice and video streaming
    - data pulling and data pushing
      * what is the force that makes the data flow? 4 choices
        + push – data source pushes data into a downstream direction
        + pull – data sink pulls data from an upstream direction
        + push/pull – a filter is actively pulling data from a steam, performing computations and pushing the data downstream
        + passive – don’t do either, act as a sink or source for data
      * combinations may be complex and make the “plumbers” job more difficult
        + if more than one filter is pushing/pulling, synchronization is needed

|  |  |
| --- | --- |
| * + - a push pipeline w an active sourceA screenshot of a cell phone        Description automatically generated | * a pull pipeline w an active sinkA screenshot of a cell phone    Description automatically generated |
| * a mixed push-pull pipeline w a passive source and sinkA screenshot of a cell phone    Description automatically generated | * a pipeline w active filters and synchronizing buffering pipesA screenshot of a social media post    Description automatically generated |

|  |  |
| --- | --- |
| Strengths | Weaknesses |
| * + - * overall behavior is a simple composition of behavior of individual filters       * reuse – any two filters can be connected if they agree on the data format that is transmitted       * ease of maintenance – filters can be added or replaced       * prototyping eg Unix shell scripts are famously powerful and flexible, using filters such as a sed and awk       * architecture supports formal analysis – throughput and deadlock detection       * potential for parallelism – filters implemented as separate tasks consuming and producing data incrementally | * + - * can degenerate to “branch processing” – filter process all of its data before passing on (rather than incrementally)       * sharing global data is expensive or limiting       * can be difficult to design incremental filters       * not appropriate for interactive applications – doesn’t split into sequential stages. POSA book has specific styles for interactive syss, one of which is Model-View-Controller       * synchronization of streams will constrain architecture       * error handling is Achilles heel         + eg. filter has consumed three quarters of its input and produced half its output and some intermediate filter crashes. generally restart pipeline (POSA)       * implementation may force lowest common denominator on data transmission         + eg Unix scripts everything is ASCII |

**Pipe and Filter vs Batch Sequential**

* both decompose the task into a fixed sequence of computations (components) interacting only through data passed from one to another

|  |  |
| --- | --- |
| Batch Sequential | Pipe and Filter |
| * course grained * high latency * external access to input * no concurrency * non interactive | * fine grained * results starts processing * localized input * concurrency possible * interactive awkward but possible |

**Call and Return**

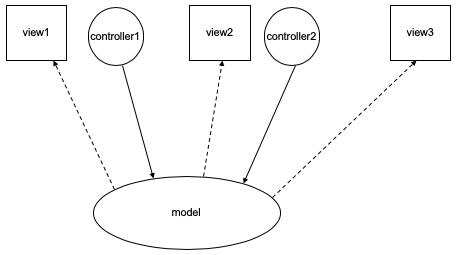
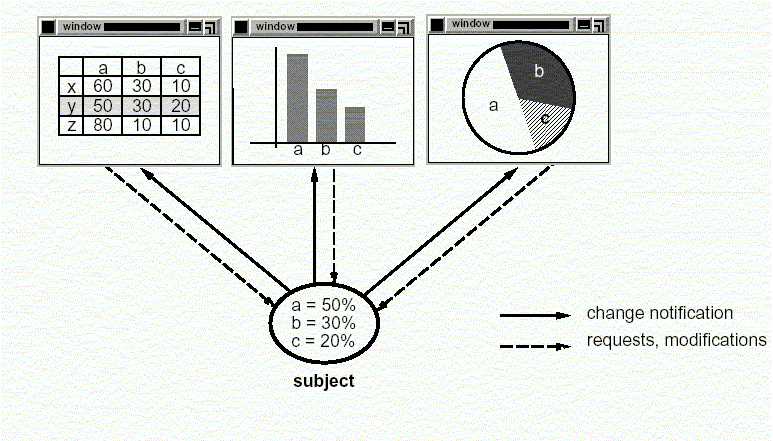
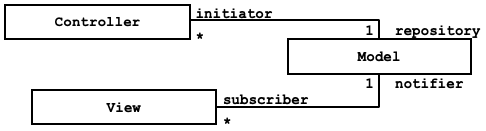
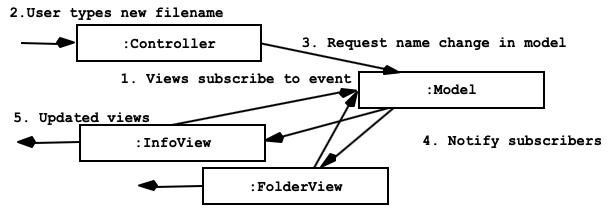
* main program/subroutines
  + hierarchical decomposition into subroutines (components) each solving a well-defined task/func
  + data passed around as parameters
  + main driver provides a control loop for sequencing through subroutines
* data abstraction/ OO (information hiding) - widely used architectural style
  + **components**
    - objects or abstract data types
  + **connections**
    - messages or func/procedure invocations
  + key aspects
    - object preserves integrity of representation – no direct access
    - representation is hidden from objects
  + variations
    - objects as concurrent tasks
    - multiple interfaces for objects (Java!)
  + note that data abstraction is different from OO – no inheritance

|  |  |
| --- | --- |
| Strengths | Weaknesses |
| * + - change implementation w/out affecting clients (assuming interface doesn’t change)     - can break problems into interacting agents (distributed across multiple machine/networks | * + - to interact objects must know each others identity (in contras to pipe and filter)     - when identity changes, objects that explicitly invoke it must change (java interfaces help though)     - side effect problems: if A uses B and C uses B, then V effects on B can be unexpected to A (and vice versa)     - complex dynamic interactions – distributed funcality |

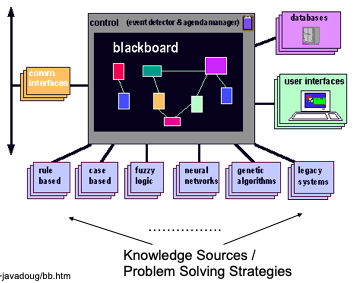
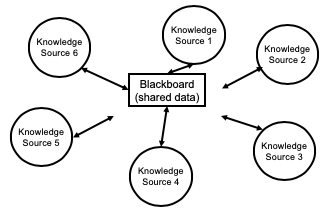
1. **Interacting processes (communication processes)**

* style of communication b/w components
  + rather than invoking a procedure directly or sending a message a component announces or broadcasts, one or more events - communications are broadcast-based and components are not necessarily objects
* **implicit invocation** 
  + components register interest in an event by associating a procedure w the event
  + when the event is announced the sys implicitly invokes all procedures that have been registered for the event
  + common style for integrating tools in a shared environment. eg
    - tools communicate by broadcasting interesting events
    - other tools register patterns that indicate which events should be routed to them and which method/procedure should be invoked when an event matches that pattern
    - pattern matcher responsible for invoking appropriate methods when each event is announced
  + editor announces it has finished editing a module, compiler registers for such announcements and automatically recompiles module
  + debugger announces it has reached a breakpoint, editor registers interest in such announcements and automatically scrolls to relevant source line
  + **components**
    - modules whose interfaces provide a collection of procedures/methods and a set of events that it may announce
  + **connectors**
    - bindings b/w event announcements and procedure/method calls
    - traditional procedure/method calls (to bypass implicit invocation)
  + invariants
    - announcers of events do not know which components will be affected by those events
    - components cannot make assumptions about ordering of processing or what processing will occur as a result of their events
  + common exs
    - programming environment tools integration
    - user interfaces -Model-View-Controller
    - syntax directed editors to support incremental semantic checking

|  |  |
| --- | --- |
| Strengths | weaknesses |
| * + - * string support for reuse – plug in new components by registering it for events       * maintenance – add and replace components w minimum affect on other components in the sys | * + - * loss of control         + when a component announces an event, it has no idea what components will respond to it         + cannot rely on order that these components will be invoked         + cannot tell when they are finished       * ensuring correctness is difficult bc it depends on context in which invoked. unpredictable interactions       * sharing data – see the Observer Design Pattern |

* + **Model-View-Controller (MVC)**  
    
    - a decomposition of an interactive sys into three components
      * a model containing the core funcality of data
      * one or more views displaying info to the user
      * one or more controllers that handle user input
    - a change propagation mechanism (ie. observer) ensure consistency b/w user interface and model
      * eg if the user changes the model through the controller of one view, the other views will be updated automatically
    - the division into the MVC components improves maintainability
    - in this architecture under the style of implicit invocation there are these different types of components
      * model components - denote and implement the domain application data
      * viewer components - denote and implement the different ways data are presented to the user or to other syss
      * controller components - implement sftwr logic that alter the data in the model (i.e. in the model components)
    - communication diagram MVC  
      

1. **Data-Oriented Repository**

* transactional data bases
  + true client/server
* modern compiler
* repositories/data centered
  + characterized by a central store component representing syss state and a collection of independent **components** that operate on the data store
  + **connections** b/w data store and external components vary considerably in this style
    - transactional data bases
      * incoming stream of transactions trigger processes to act on data store. passive
    - blackboard architecture
      * current state of data store triggers processes. active
* **blackboard**
  + cooperating “partial solution solvers” collaborating but not following a pre-defined strategy
  + current state of the solution stored in the black board
  + processing triggered by the state of the blackboard
  + blackboard architectural style ex  
    
    - problems for which no deterministic solution strategy is known, but many different approaches (often alternative ones) exist and are used to build a partial or approximate solution
      * AI: vision, speech and pattern recognition
      * modern compilers act on shared data: symbol table, abstract syntax tree

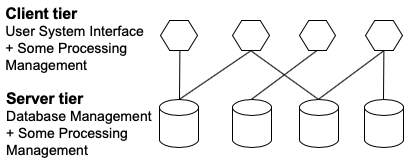
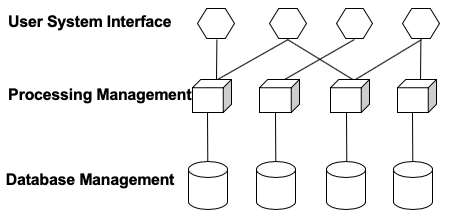
1. **Data Sharing**

* compound documents
* hypertext
* Fortran COMMON
* LW processes

1. **Hierarchical** layered

* “a layered sys is organized hierarchically, each later providing service to the layer above it and serving as a client to the layer below”
* each layer collects services at a particular level of abstraction
* in a pure layered sys: layers are hidden to all except adjacent layers
* applicability
  + a large sys that is characterized by a mix of high and low level issues, where high level issues depend on lower level ones
* **components**
  + group of subtasks which implement a “virtual machine” at some layer in the hierarchy
* **connectors**
  + protocols / interface that define how the layers will interact
* invariants
  + limit layer (component) interactions to adjust layers (in practive this may be relaxed for efficiency reasons)
* typical variant relaxing the pure style
  + a layer may access services of all layers below it
* common exs
  + communication protocols
    - each level supports communication at a level of abstraction, lower levels provide lower levels of communication and lowest level being hardware communications
  + Ex 1
    - ISO defined in the OSI 7-layer architectural model w layers: Application, Presentation,…, Data, Physical
      * protocol specifies behavior at each level of abstraction (layer)
      * each layer deals w specific level of communication and uses services of the next lower level
  + Ex 2
    - TCP/IP is the basic communications protocol used on the internet. POSA book describes 4 layers: ftp, tcp, ip, Ethernet. The same layers in the network communicate “virtually”
  + Ex 3
    - operating syss
      * eg hardware, …, kernel, resource management, … user level “onion skin model”

|  |  |
| --- | --- |
| Strengths | weaknesses |
| * + - increasing levels of abstraction as we move up through layers – partitions complex problems     - maintenance – in theory, a layer only interacts w layers above and below. change has minimum effect     - reuse – different implementations of the same level can be interchanged     - standardization based on layers eg OSI | * + - not all syss are easily structured in layers       * eg mobile robotics     - performance communicating down through layers and back up, hence bypassing may occur efficiency reasons |

* similar to strengths to data abstraction/OO but w multiple levels of abstraction
  + eg well defined interfaces, implementation hiding
* similar to pipelines
  + eg communication w at most one component at either side, but w richer form of communication
* a layer can be viewed as aka “virtual machine” providing a standardized interface to the one above it
* **interpreter**
  + architecture is based on virtual machine produced in sftwr
  + special kind of layered architecture where a layer is implemented as a true language interpreter
  + components are “program” being executed, its data, the interpretation engine and its state
    - eg. Java virtual machine. Java code translated to platform independent bytecodes. JVM is platform specific and interprets (or compiles – JIT) the bytecodes
* **tiered architectures**
  + special kind of layered architecture for enterprise applications
  + two tier client server architecture design
    - to decouple (typically form/based) user interface from the storage of data
    - improved maintainability (changes to UI and database can be made independently); scales up to 100 users
  + three tier client server architecture design
    - to overcome the limitations of the two tier architecture by adding an additional middle tier
    - this middle tier provides process management where business logic and rules are executed and can accommodate hundreds of users by providing generic services such as queuing, application execution and database staging
    - an effective distributed client/server design that provides increased performance, flexibility, maintainability, reusability and scalability, while hiding the complexity of distributed processing form the user