Notes in Software Architecture

ECE 756-3

- Sources used in preparing these slides:
  - **Lecture slides on Architecture by David Garlan, see http://www-2.cs.cmu.edu/afs/cs/academic/class/17655-s02/www/**
  - **Lecture slides on Architecture by Marc Roper and Murray Wood, see https://www.cis.strath.ac.uk/teaching/ug/classes/52.440/**
  - **M. Shaw and D. Garlan. *Software Architecture: Perspectives on a Emerging Discipline*. Prentice Hall, Englewood Cliffs, NJ, 1996**
  - **F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. John Wiley & Sons Ltd., Chichester, UK, 1996**
  - **B. Bruege, A. Duboit, *Object Oriented Software Engineering Using UML, Patterns, and Java*, Prentice Hall, 2004**
  - **K. Czarneski, Lecture Notes, Software Engineering, ECE 355, U. Waterloo**

Design

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.”

- C.A.R. Hoare

What Is Design?

- Requirements specification was about the WHAT the system will do

- Design is about the HOW the system will perform its functions
  - **provides the overall decomposition of the system**
  - **allows to split the work among a team of developers**
  - **also lays down the groundwork for achieving non-functional requirements (performance, maintainability, reusability, etc.)**
  - **takes target technology into account (e.g., kind of middleware, database design, etc.)**

Software Development Activities

- Requirements Elicitation

- Requirements Analysis (e.g., Structured Analysis, OO Analysis)
  - **analyzing requirements and working towards a *conceptual* model *without* taking the target implementation technology into account**
  - **useful if the conceptual gap between requirements and implementation is large**
  - **part of requirements engineering (but may produce more than what is going to be part of the requirement spec)**

- Design
  - **coming up with solution models *taking* the target implementation technology into account**

- Implementation
- Test
- ...

Levels of Design

- Architectural design (also: high-level design)
  - **architecture - the overall structure: main modules and their connections**
  - **design that covers the main use-cases of the system**
  - **addresses the main non-functional requirements (e.g., throughput, reliability)**
  - **hard to change**

- Detailed design (also: low-level design)
  - **the inner structure of the main modules**
  - **may take the target programming language into account**
  - **detailed enough to be implemented in the programming language**

Why is Design so Difficult?

- *Analysis:* Focuses on the application domain

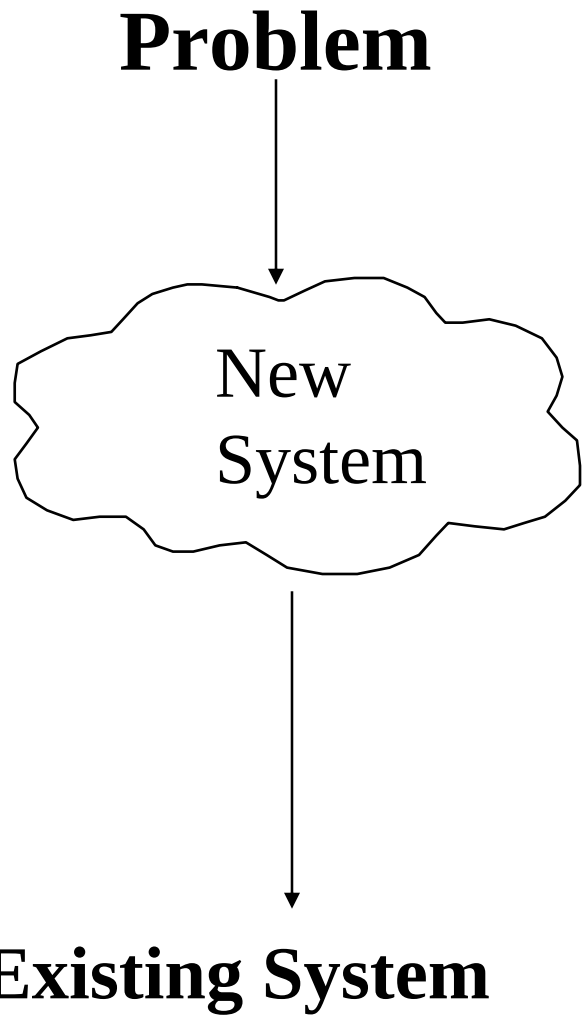
- *Design:* Focuses on the solution domain
  - **Design knowledge is a moving target**
  - **The reasons for design decisions are changing very rapidly**
    - **Halftime knowledge in software engineering: About 3-5 years**
    - **What I teach today will be out of date in 3 years**
    - **Cost of hardware rapidly sinking**

- "Design window":
  - **Time in which design decisions have to be made**

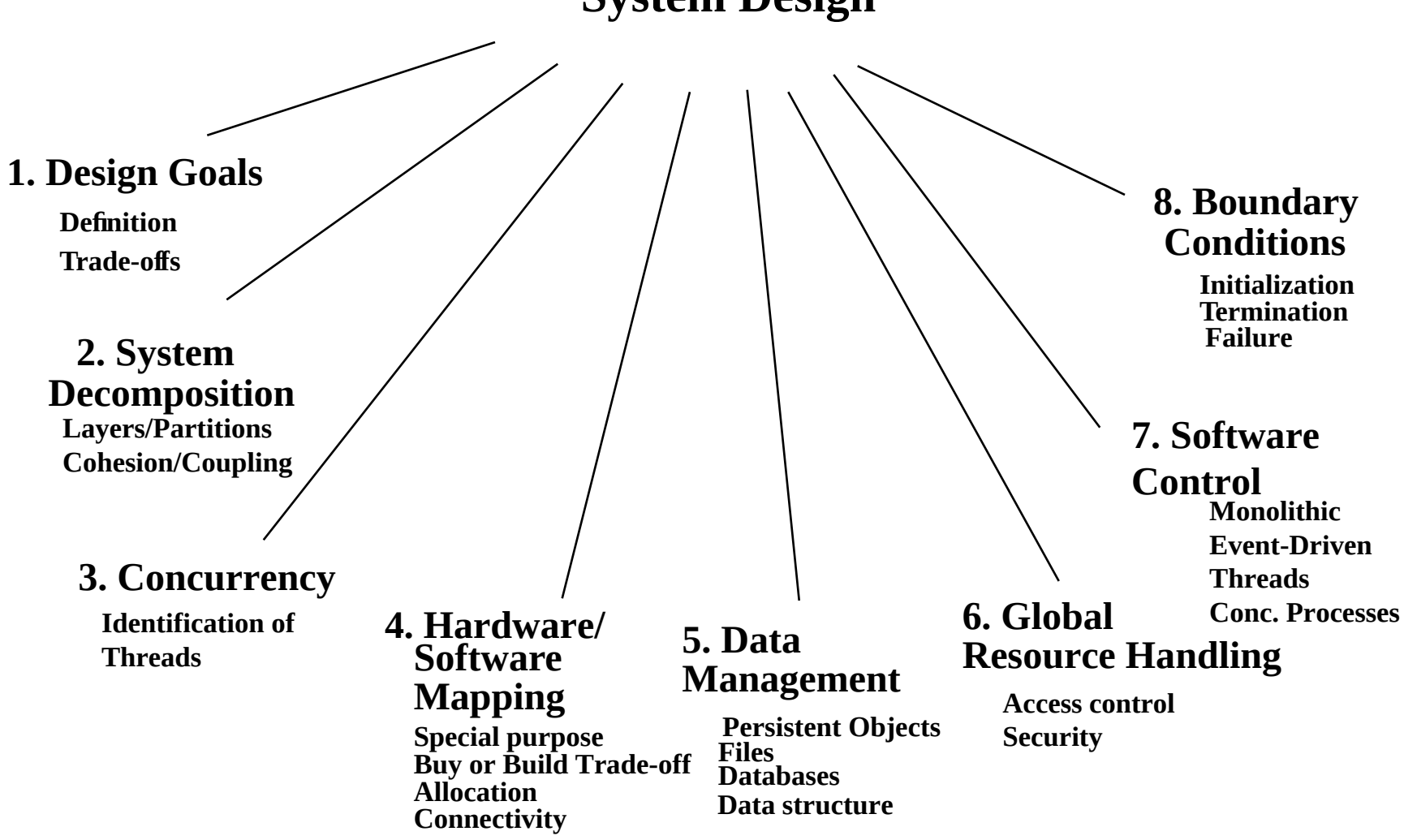
- Technique
  - **Time-boxed prototyping**

The Purpose of System Design

- Bridging the gap between desired and existing system in a manageable way
- Use Divide and Conquer
  - **We model the new system to be developed as a set of subsystems**



System Design



Overview

System Design I (Today)

- **0. Overview of System Design**
- **1. Design Goals**
- **2. Subsystem Decomposition**

System Design II: Addressing Design Goals (next lecture)

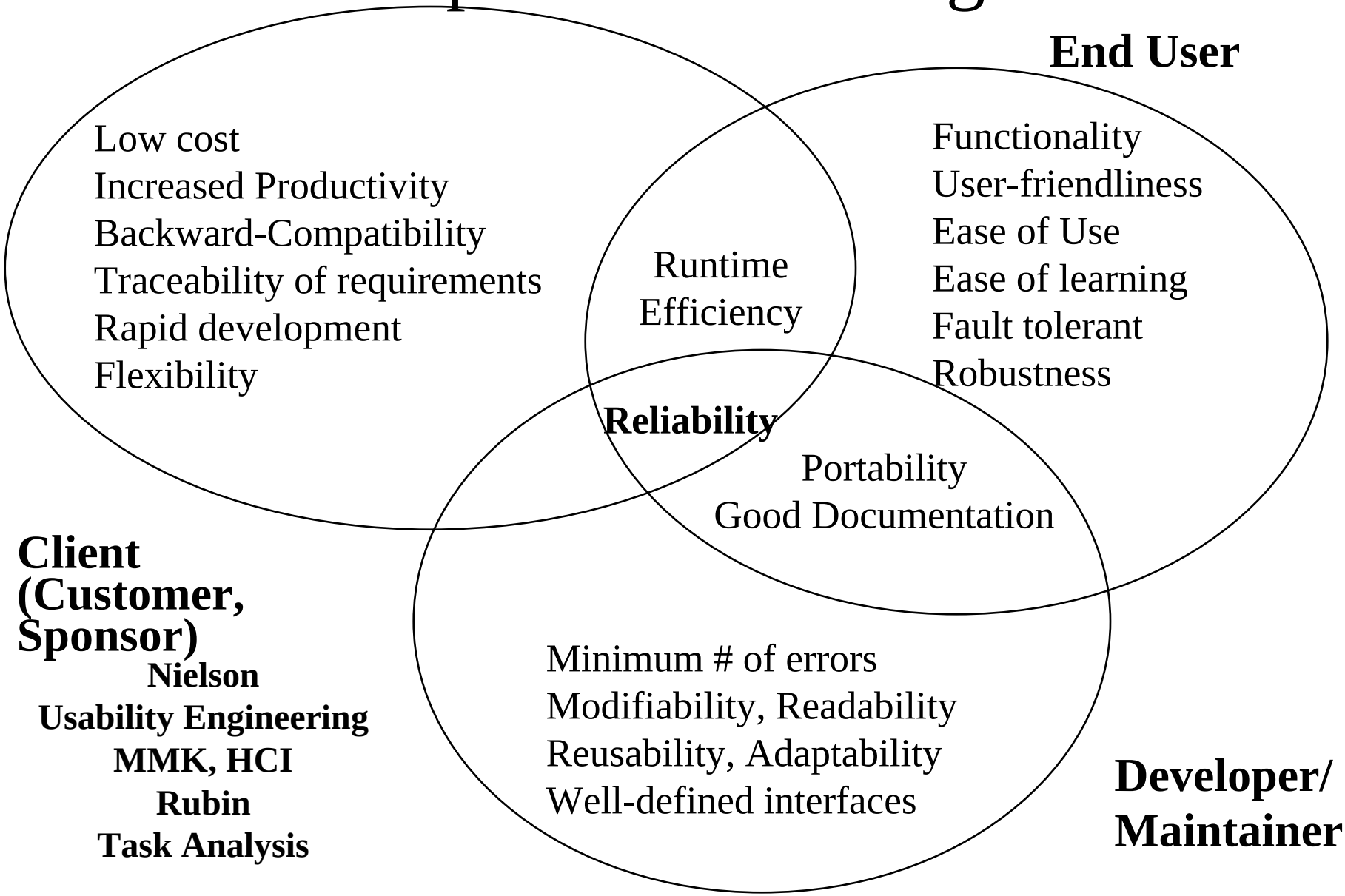
- **3. Concurrency**
- **4. Hardware/Software Mapping**
- **5. Persistent Data Management**
- **6. Global Resource Handling and Access Control**
- **7. Software Control**
- **8. Boundary Conditions**

List of Design Goals

- Reliability
- Modifiability
- Maintainability
- Understandability
- Adaptability
- Reusability
- Efficiency
- Portability
- Traceability of requirements
- Fault tolerance
- Backward-compatibility
- Cost-effectiveness
- Robustness
- High-performance

- Good documentation
- Well-defined interfaces
- User-friendliness
- Reuse of components
- Rapid development
- Minimum # of errors
- Readability
- Ease of learning
- Ease of remembering
- Ease of use
- Increased productivity
- Low-cost
- Flexibility

Relationship Between Design Goals



Typical Design Trade-offs

- Functionality vs. Usability
- Cost vs. Robustness
- Efficiency vs. Portability
- Rapid development vs. Functionality
- Cost vs. Reusability
- Backward Compatibility vs. Readability

NFRs and the use of Design Patterns

- Read the problem statement again
- Use textual clues (similar to Abbot's technique in Analysis) to identify design patterns

- *Text:* "manufacturer independent", "device independent", "must support a family of products"
  - **Abstract Factory Pattern**
- *Text:* "must interface with an existing object"
  - **Adapter Pattern**
- *Text:* "must deal with the interface to several systems, some of them to be developed in the future", " an early prototype must be demonstrated"
  - **Bridge Pattern**

Textual Clues in NFRs

- *Text:* “complex structure”, “must have variable depth and width”
  - **Composite Pattern**
- *Text:* “must interface to an set of existing objects”
  - **Façade Pattern**
- *Text:* “must be location transparent”
  - **Proxy Pattern**
- *Text:* “must be extensible”, “must be scalable”
  - **Observer Pattern**
- *Text:* “must provide a policy independent from the mechanism”
  - **Strategy Pattern**

Section 2. System Decomposition

- Subsystem (UML: Package)
  - **Collection of classes, associations, operations, events and constraints that are interrelated**
  - **Seed for subsystems: UML Objects and Classes.**
- (Subsystem) Service:
  - **Group of operations provided by the subsystem**
  - **Seed for services: Subsystem use cases**
- Service is specified by Subsystem interface:
  - **Specifies interaction and information flow from/to subsystem boundaries, but not inside the subsystem.**
  - **Should be well-defined and small.**
  - **Often called API: Application programmer's interface, but this term should used during implementation, not during System Design**

Why modularize a system?

- **Management**: Partition the overall development effort
  - **divide and conquer (actually: “Divide et impera” = “Divide and rule”)**

- **Evolution**: Decouple parts of a system so that changes to one part are isolated from changes to other parts
  - **Principle of directness (clear allocation of requirements to modules, ideally one requirement (or more) maps to one module)**
  - **Principle of continuity (small change in requirements triggers a change to one module only)**

- **Understanding**: Permit system to be understood
  - **as composition of mind-sized chunks**
    - **e.g., the 7±2 Rule**
  - **with one issue at a time**
    - **Principle of locality, encapsulation, separation of concerns**

- Key issue: what criteria to use for modularization

Information hiding (Parnas)

- Hide secrets. OK, what's a "secret"?
  - **Representation of data**
  - **Properties of a device, other than required properties**
  - **Implementation of world models**
  - **Mechanisms that support policies**

- Try to localize future change
  - **Hide system details likely to change independently**
  - **Separate parts that are likely to have a different rate of change**
  - **Expose in interfaces assumptions unlikely to change**

Further Principles

- Explicit interfaces
  - **make all dependencies between modules explicit (no hidden coupling)**
- Low coupling - few interfaces
  - **minimize the amount of dependencies between modules**
- Small interfaces
  - **keep the interfaces narrow**
    - **combine many parameters into structs/objects**
    - **divide large interfaces into several interfaces**
- High cohesion
  - **a module should encapsulate some well-defined, coherent piece of functionality (more on that later)**

What Is an Interface?

- Whatever is published by a module that clients can depend on

- Syntactic interface
  - **How to call operations**
    - **list of operation signatures**
    - **sometimes also valid orders of calling operations**

- Semantic interfaces
  - **What the operations do, e.g.,**
    - **pre- and post-conditions**
    - **use cases**
  - **performance specification**
  - **...**

Services and Subsystem Interfaces

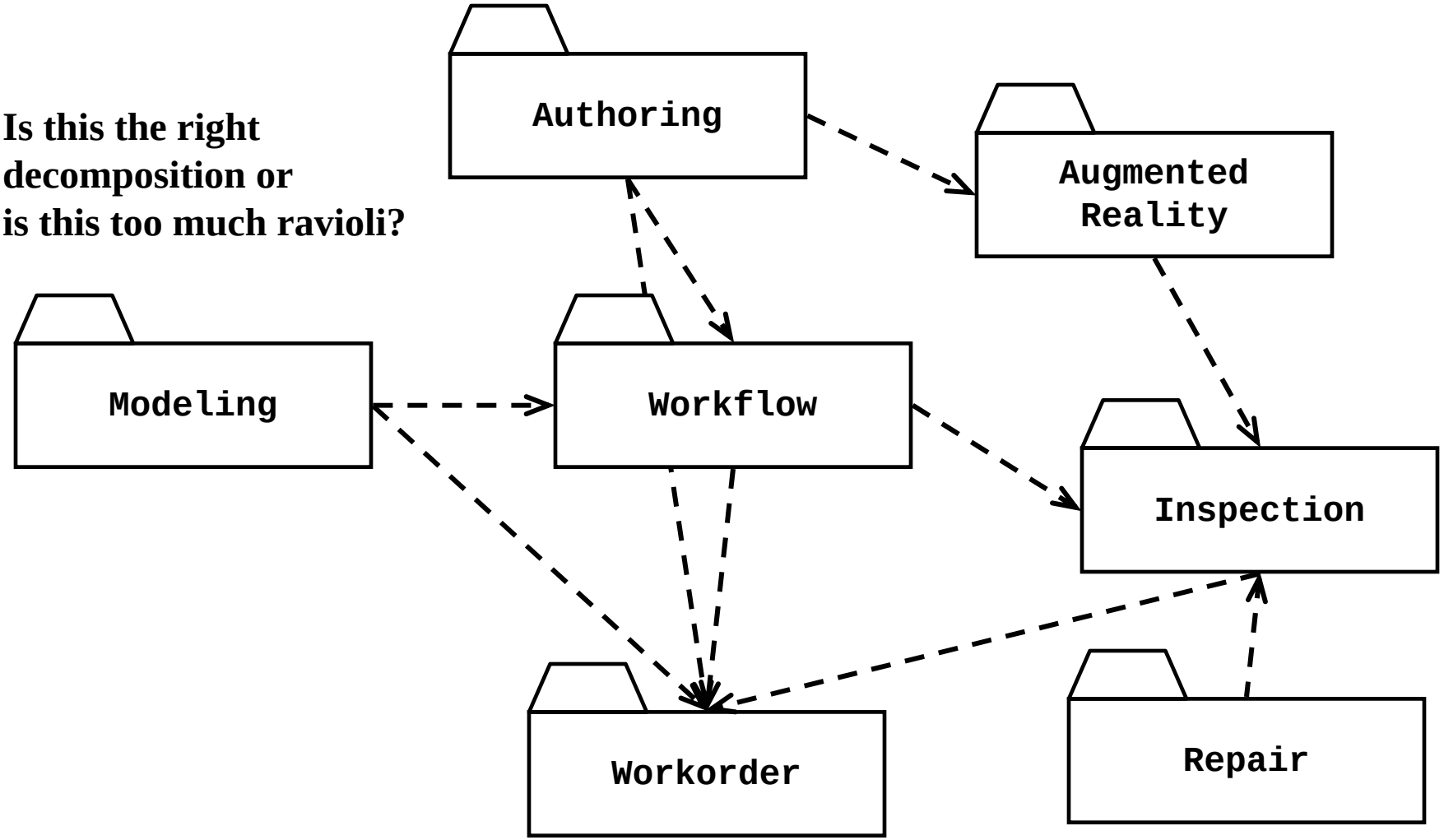
- Service: A set of related operations that share a common purpose
  - **Notification subsystem service:**
    - **LookupChannel()**
    - **SubscribeToChannel()**
    - **SendNotice()**
    - **UnscubscribeFromChannel()**
  - **Services are defined in System Design**
- Subsystem Interface: Set of fully typed related operations.
  - **Subsystem Interfaces are defined in Object Design**
  - **Also called application programmer interface (API)**

Choosing Subsystems

- Criteria for subsystem selection: Most of the interaction should be within subsystems, rather than across subsystem boundaries (High cohesion).
  - **Does one subsystem always call the other for the service?**
  - **Which of the subsystems call each other for service?**
- Primary Question:
  - **What kind of service is provided by the subsystems (subsystem interface)?**
- Secondary Question:
  - **Can the subsystems be hierarchically ordered (layers)?**
- What kind of model is good for describing layers and partitions?

Subsystem Decomposition Example

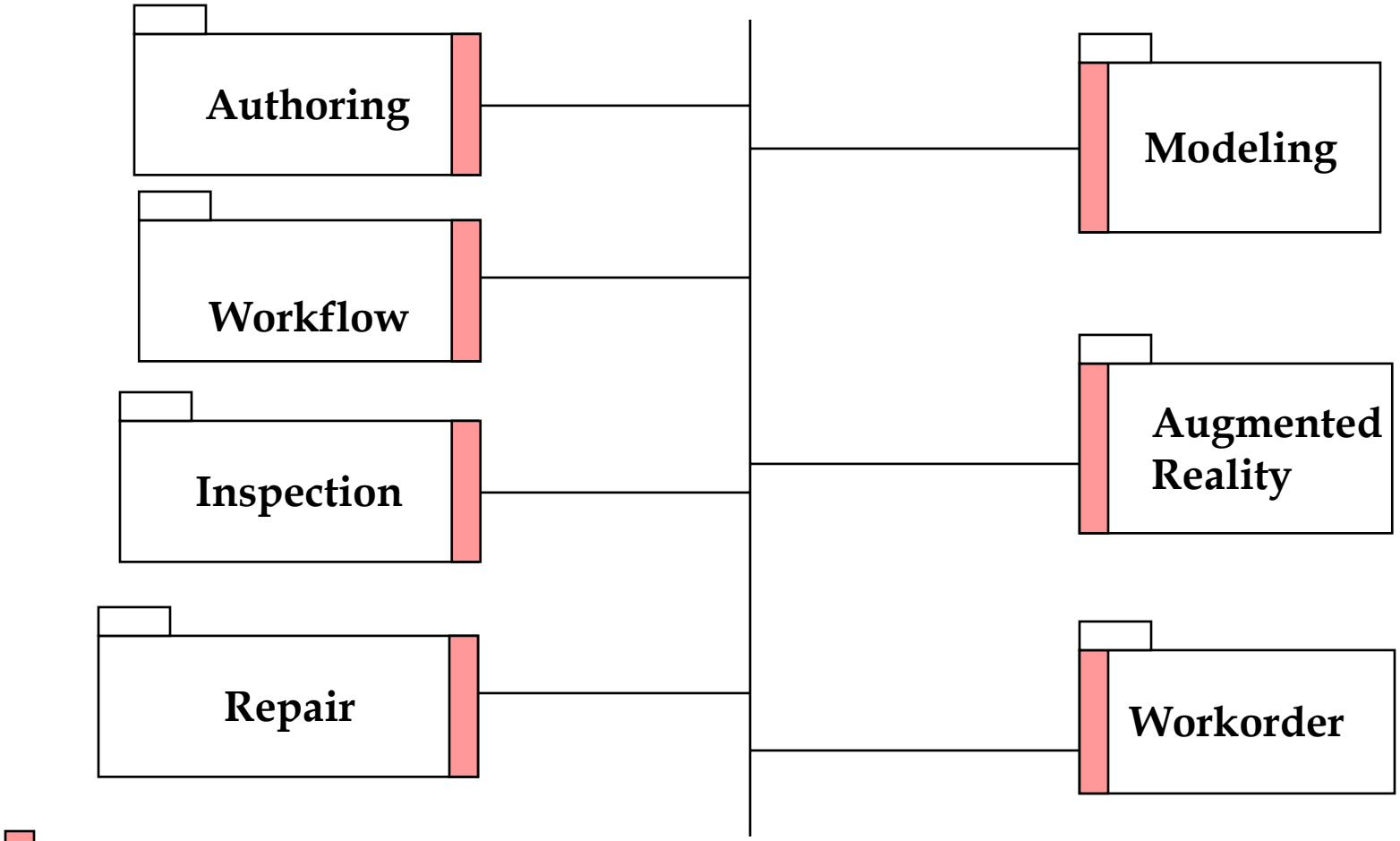
Is this the right decomposition or is this too much ravioli?



Definition: Subsystem Interface Object

- A *Subsystem Interface Object* provides a service
  - **This is the set of public methods provided by the subsystem**
  - **The Subsystem interface describes all the methods of the subsystem interface object**
- Use a Facade pattern for the subsystem interface object

System as a set of subsystems communicating via a software bus



A Subsystem Interface Object publishes the service (= Set of public methods) provided by the subsystem

Coupling and Cohesion

- Goal: Reduction of ***complexity while change occurs***
- Cohesion measures the dependence among classes
  - **High cohesion: The classes in the subsystem perform similar tasks and are related to each other (via associations)**
  - **Low cohesion: Lots of miscellaneous and auxiliary classes, no associations**
- Coupling measures dependencies between subsystems
  - **High coupling: Changes to one subsystem will have high impact on the other subsystem (change of model, massive recompilation, etc.)**
  - **Low coupling: A change in one subsystem does not affect any other subsystem**
- Subsystems should have as maximum cohesion and minimum coupling as possible:
  - **How can we achieve high cohesion?**
  - **How can we achieve loose coupling?**

Degrees of Cohesion

1. Coincidental cohesion
2. Logical cohesion
3. Temporal cohesion
4. Procedural cohesion
5. Communicational cohesion
6. Functional cohesion
7. Informational cohesion

Low cohesion - bad

High cohesion - good

Coincidental cohesion

- The result of *randomly* breaking the project into modules to gain the benefits of having multiple smaller files/modules to work on
  - **Inflexible enforcement of rules such as: “every function/module shall be between 40 and 80 lines in length” can result in coincidental coherence**
- Usually worse than no modularization
  - **Confuses the reader that may infer dependencies that are not there**

Logical cohesion

- A “template” implementation of a number of quite different operations that share some basic course of action
  - **variation is achieved through parameters**
  - **“logic” - here: the internal workings of a module**

- Problems:
  - **Results in hard to understand modules with complicated logic**
  - **Undesirable coupling between operations**

- Usually should be refactored to separate the different operations

Example of Logical Cohesion

```
void function(param1, param2, param3, ..., paramN)
{
        variable declarations....
        code common to all cases... [A]
        if ( param1 == 1 ) [B]
                ...
        else if ( param1 == 2 )
                ...
        else if ( param1 == n )
                ...
        end if
        code common to all cases... [C]
        if ( param == 1) [D]
                ...
        else if ( param1 == 5 )
                ...
        end if
        code common to all cases... [E]
        if ( param1 == 7 )
                ...
}
```

Temporal Cohesion

- Temporal cohesion concerns a module organized to contain all those operations which occur at a similar point in time.

- Consider a product performing the following major steps:
  - **initialization, get user input, run calculations, perform appropriate output, cleanup.**

- Temporal cohesion would lead to five modules named initialize, input, calculate, output and cleanup.

- This division will most probably lead to code duplication across the modules, e.g.,
  - **Each module may have code that manipulates one of the major data structures used in the program.**

Procedural Cohesion

- A module has procedural cohesion if all the operations it performs are related to a sequence of steps performed in the program.
- For example, if one of the sequence of operations in the program was “read input from the keyboard, validate it, and store the answers in global variables”, that would be procedural cohesion.
- Procedural cohesion is essentially temporal cohesion with the added restriction that all the parts of the module correspond to a related action sequence in the program.
- It also leads to code duplication in a similar way.

Procedural Cohesion

Module A

```
operationA()
{ readData(data,filename1);
  processAData(data);
  storeData(data,filename2);
}

readData(data,filename)
{ f := openfile(filename);
  readrecords(f, data);
  closefile(f);
}
storeData(data,filename)
{...}
processAData(data)
{...}
```

Module B

```
operationB()
{ readData(data,filename1);
  processBData(data);
  storeData(data,filename2);
}

readData(data,filename)
{ f := openfile(filename);
  readrecords(f, data);
  closefile(f);
}
storeData(data,filename)
{...}
processBData(data)
{...}
```

Communicational Cohesion

- Communicational cohesion occurs when a module performs operations related to a sequence of steps performed in the program (see procedural cohesion) AND all the actions performed by the module are performed on the same data.

- Communicational cohesion is an improvement on procedural cohesion because all the operations are performed on the same data.

Functional Cohesion

- Module with functional cohesion focuses on exactly one goal or “function”
  - **(in the sense of purpose, not a programming language “function”).**
- Module performing a well-defined operation is more reusable, e.g.,
  - **modules such as: read_file, or draw_graph are more likely to be applicable to another project than one called initialize_data.**
- Another advantage of is fault isolation, e.g.,
  - **If the data is not being read from the file correctly, there is a good chance the error lies in the read_file module/function.**

Informational Cohesion

- Informational cohesion describes a module as performing a number of actions, each with a unique entry point, independent code for each action, and all operations are performed on the same data.
  - **In informational cohesion, each function in a module can perform exactly one action.**

- It corresponds to the definition of an ADT (abstract data type) or object in an object-oriented language.

- Thus, the object-oriented approach naturally produces designs with informational cohesion.
  - **Each object is generally defined in its own source file/module, and all the data definitions and member functions of that object are defined inside that source file (or perhaps one other source file, in the case of a .hpp/.cpp combination).**

Levels of Coupling

5. Content Coupling (High Coupling - Bad)
4. Common Coupling
3. Control Coupling
2. Stamp Coupling
1. Data Coupling (Low Coupling - Good)

(Remember: no coupling is best!)

Content Coupling

- One module directly refers to the content of the other
  - **module 1 modifies a statement of module 2**
    - **assembly languages typically supported this, but not high-level languages**
    - **COBOL, at one time, had a verb called alter which could also create self-modifying code (it could directly change an instruction of some module).**
  - **module 1 refers to local data of module 2 in terms of some kind of offset into the start of module 2.**
    - **This is not a case of knowing the offset of an array entry - this is a direct offset from the start of module 2's data or code section.**
  - **module 1 branches to a local label contained in module 2.**
    - **This is not the same as calling a function inside module 2 - this is a goto to a label contained somewhere inside module 2.**

Common Coupling

- Common coupling exists when two or more modules have read *and* write access to the same global data.

- Common coupling is problematic in several areas of design/maintenance.
  - **Code becomes hard to understand - need to know all places in all modules where a global variable gets modified**
  - **Hampered reusability because of hidden dependencies through global variables**
  - **Possible security breaches (an unauthorized access to a global variable with sensitive information)**

- It’s ok if just one module is writing the global data and all other modules have read-only access to it.

Common Coupling

- Consider the following code fragment:

```
while( global_variable > 0 )
{ switch( global_variable )
   {  case 1: function_a(); break;
      case 2: function_b(); break;
      ...
      case n: ...
   }
   global_variable++;
}
```

Common Coupling

- If function_a(), function_b(), etc can modify the value of global variable, then it can be extremely difficult to track the execution of this loop.

- If they are located in two or more different modules, it becomes even more difficult
  - **potentially all modules of the program have to be searched for references to global variable, if a change or correction is to take place.**

- Another scenario is if all modules in a program have access to a common database in both read and write mode, even if write mode is not required in all cases.

- Sometimes necessary, if a lot of data has to be supplied to each module

Control Coupling

- Two modules are control-coupled if module 1 can directly affect the execution of module 2, e.g.,
  - **module 1 passes a “control parameter” to module 2 with logical cohesion, or**
  - **the return code from a module 2 indicates NOT ONLY success or failure, but also implies some action to be taken on the part of the calling module 1 (such as writing an error message in the case of failure).**

- The biggest problem is in the area of code re-use: the two modules are not independent if they are control coupled.

Stamp Coupling

- It is a case of passing more than the required data values into a module, e.g.,
  - **passing an entire employee record into a function that prints a mailing label for that employee. (The data fields required to print the mailing label are name and address. There is no need for the salary, SIN number, etc.)**
- Making the module depend on the names of data fields in the employee record hinders portability.
  - **If instead, the four or five values needed are passed in as parameters, this module can probably become quite reusable for other projects.**
- As with common coupling, leaving too much information exposed can be dangerous.

Data Coupling

- Data coupling exhibits the properties that all parameters to a module are either simple data types, or in the case of a record being passed as a parameter, all data members of that record are used/required by the module. That is, no extra information is passed to a module at any time.

What Is Software Architecture?

- Captures the gross structure of a system
  - **How it is composed of interacting parts**
  - **How the interactions take place**
  - **Key properties of the parts**
- Provides a way of analysing systems at a high level of abstraction
- Illuminates top-level design decisions

Definition by Shaw and Garlan

- Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among these components. Such a system may in turn be used as a (composite) element in a larger system design. [Garlan&Shaw]

Definition by Buschmann et al.

- A software architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and nonfunctional properties of a software system. The software architecture of a system is an artifact. It is the result of the software development activity. [POSA]

- See http://www.sei.cmu.edu/architecture/definitions.html for 60+ other definitions…

Issues Addressed by an Architectural Design

- Gross decomposition of a system into interacting components
  - **Typically hierarchical**
  - **Using rich abstractions for "glue"**
  - **Often using common design idioms/styles**
- Emergent system properties
  - **Performance, throughput, latencies**
  - **Reliability, security, fault tolerance, evolvability**
- Rationale
  - **Relates requirements and implementations**
- Envelope of allowed change
  - **"Load-bearing walls"**
  - **Design idioms and styles**

Good Properties of an Architecture

- Good architecture (like much good design):
  - **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 product lifetime**
  - **Reuse of established engineering knowledge**

Architecture Development

- Unified Process:
  - **Focus on implementing the most valuable and critical use cases first**
  - **Produce an architectural description by taking those design elements that are needed to explain how the system realizes these use cases at a high level**

- Use past and proven experience by applying architectural styles and patterns

Architectural Styles

- The architecture of a system includes
  - **Components: define the locus of computation**
    - **Examples: filters, databases, objects, ADTs**
  - **Connectors: define the interactions between components**
    - **Examples: procedure call, pipes, event announce**

- An architectural style defines a family of architectures constrained by
  - **Component/connector vocabulary**
  - **Topology**
  - **Semantic constraints**

Architectural Styles and Patterns

- An *architectural style* defines a family of architectures constrained by
  - **Component/connector vocabulary, e.g.,**
    - **layers and calls between them**
  - **Topology, e.g.,**
    - **stack of layers**
  - **Semantic constraints, e.g.,**
    - **a layer may only talk to its adjacent layers**

- For each architectural style, an *architectural pattern* can be defined
  - **It's basically the architectural style cast into the pattern form**
  - **The pattern form focuses on identifying a problem, context of a problem with its forces, and a solution with its consequences and tradeoffs; it also explicitly highlights the composition of patterns**

Catalogues of Architectural Styles and Patterns

- Architectural styles
  - **[Garlan&Shaw] M. Shaw and D. Garlan. *Software Architecture: Perspectives on a Emerging Discipline*. Prentice Hall, Englewood Cliffs, NJ, 1996**
- Architectural Patterns
  - **[POSA] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. John Wiley & Sons Ltd., Chichester, UK, 1996**

Taxonomy of Architectural Styles

- Data flow
- Call-and-return
- Interacting processes
- Data-oriented repository
- Data-sharing
- Hierarchical
- …

“Pure” Form of Styles

- When we introduce a new style, we will typically first examine its “pure” form.
  - **Pure data flow styles (or any other architectural style) are rarely found in practice**
  - **Systems in practice**
    - **Regularly deviate from the academic definitions of these systems**
    - **Typically feature many architectural styles simultaneously**
  - **As an architect you must understand the “pure” styles to understand the strength and weaknesses of the style as well as the consequences of deviating from the style**

Overview

- Context
- What is software architecture?
- Example: Mobile Robotics
- Architectural styles and patterns
  - ➔ **Data flow**
  - **Call-and-return**
  - **Interacting processes**
  - **Data-oriented repository**
  - **Data-sharing**
  - **Hierarchical**
  - **Other**
- Heterogeneous architectures

Data Flow

- A data flow system is one in which:
  - **The availability of data controls the computation**
  - **The structure of the design is determined by the orderly motion of data from component to component**
  - **The pattern of data flow is explicit**
  - **This is the only form of communication between components**

- There are variety of variations on this general theme:
  - **How control is exerted (e.g., push versus pull)**
  - **Degree of concurrency between processes**
  - **Topology**

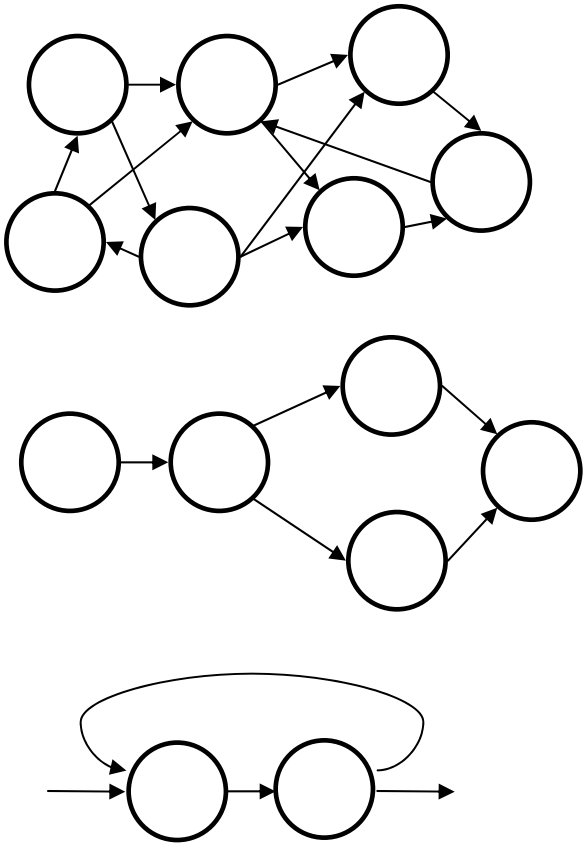
Data Flow

- Components: Data Flow Components
  - **Interfaces are input ports and output ports**
  - **Input ports read data; output ports write data**
  - **Computational model: read data from input ports, compute, 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**

- Systems
  - **Arbitrary graphs**
  - **Computational model: functional composition**

Patterns of Data Flow in Systems



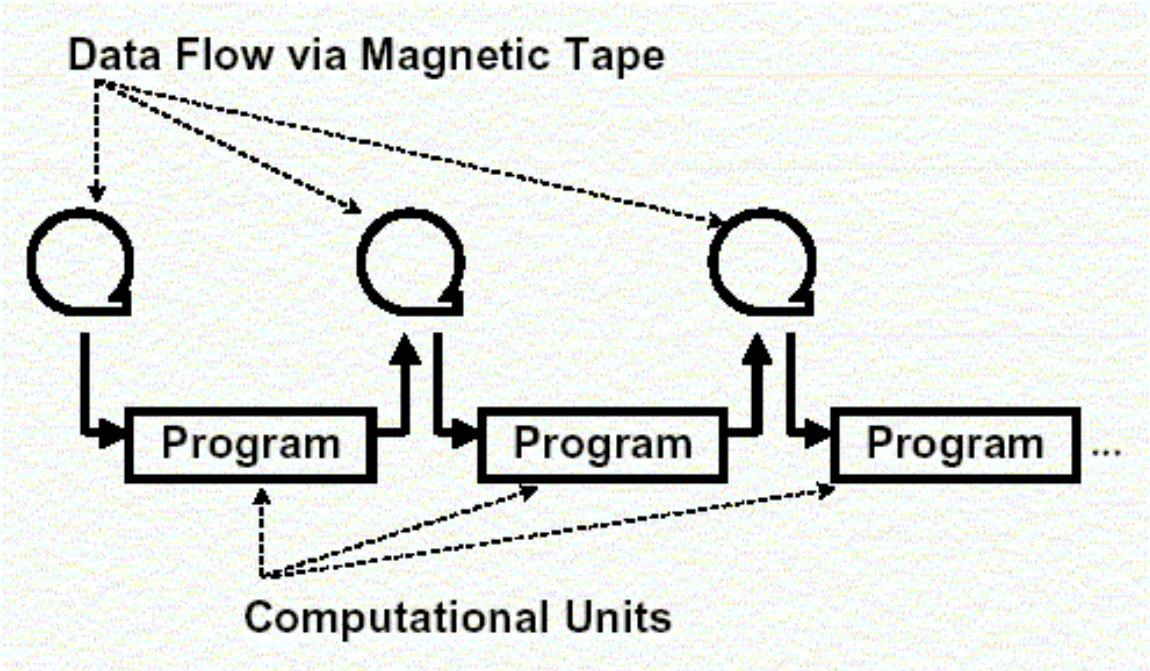
- Data can flow in arbitrary patterns
- Primarily we are interested in linear data flow patterns
- ...or in simple, constrained cyclical patterns...

Kinds of Data Flow Architectures

- ➔ Batch sequential
- ♦ Dataflow network (pipes&filters)
  - ◆ **acyclic, fanout, pipeline, Unix, etc.**
- ♦ Closed loop control

Characteristics of Batch Sequential Systems

- 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



Characteristics of Batch Sequential Systems

- 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**

Characteristics of Batch Sequential Systems

- Transformational data analysis
  - **Raw data is gathered and analyzed in a step-wise, batch-oriented fashion**
- Typical applications: non real-time, batch oriented computations such as:
  - **Payroll computations**
  - **IRS tax return computations**

Kinds of Data Flow Architectures

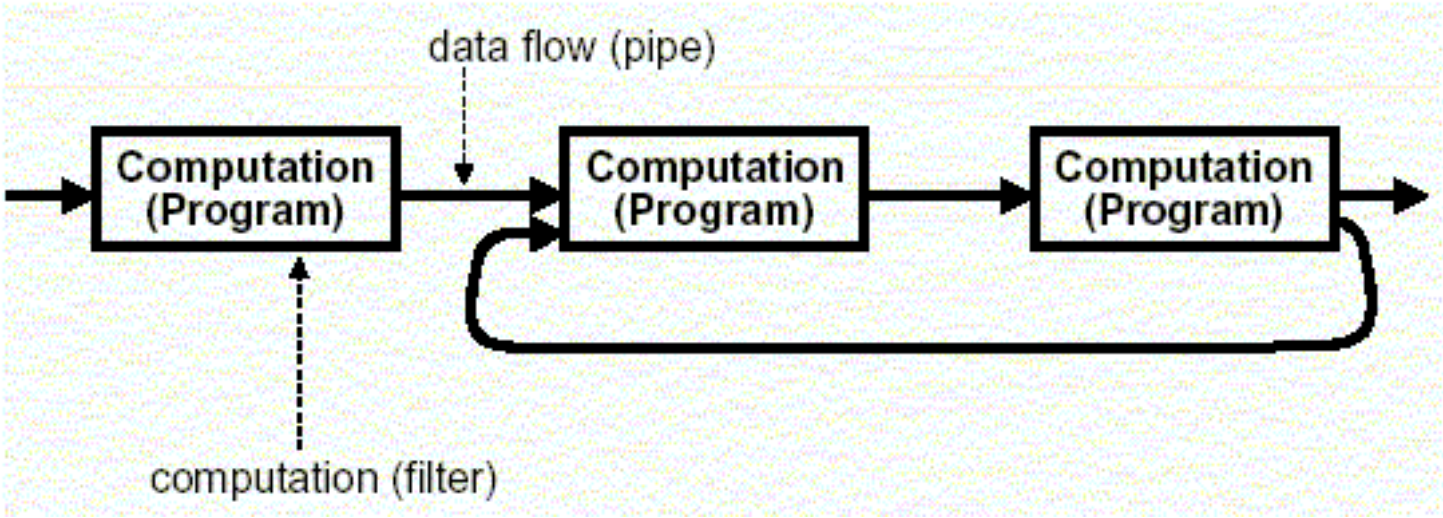
- Batch sequential

➔ Dataflow network (pipes&filters)
  - **acyclic, fanout, pipeline, Unix, etc.**

- Closed loop control

Pipes and Filters

- The tape of the batch sequential system, morphed into a language and operating system construct
- Compared to the batch-sequential style, data in the pipe&filter style is processed *incrementally*



Pipes and Filters

- “The Pipes and Filters architectural pattern [style] provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.” [POSA p53]

Pipes and Filters

- Components (Filters)
  - **Read streams of data on input producing streams of data on output**
  - **Local incremental transformation to input stream (e.g., filter, enrich, change representation, etc.)**
  - **Data is processed as it arrives, not gathered then processed**
  - **Output usually begins before input is consumed**
- Connectors (Pipes)
  - **Conduits for streams, e.g., first-in-first-out buffer**
  - **Transmit outputs from one filter to input of other**

Pipes and Filters

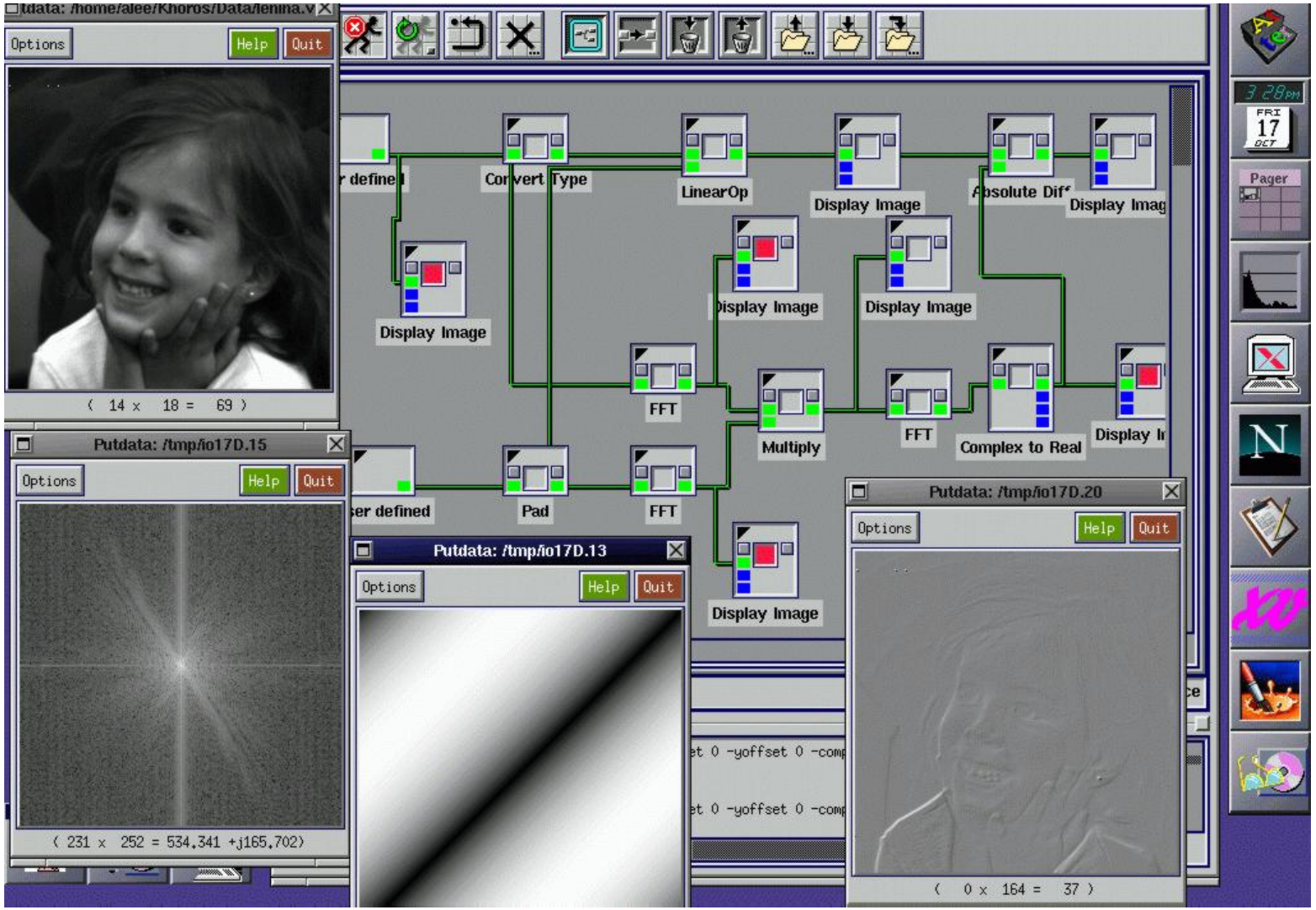
- Invariants
  - **Filters must be independent, no shared state**
  - **filters don’t know upstream or downstream filter identity**
  - **Correctness of output from network must not depend on order in which individual filters provide their incremental processing**

- Common specializations
  - **Pipelines: linear sequence of filters**
  - **Bounded and typed pipes …**

Example Pipe-and-Filter Systems

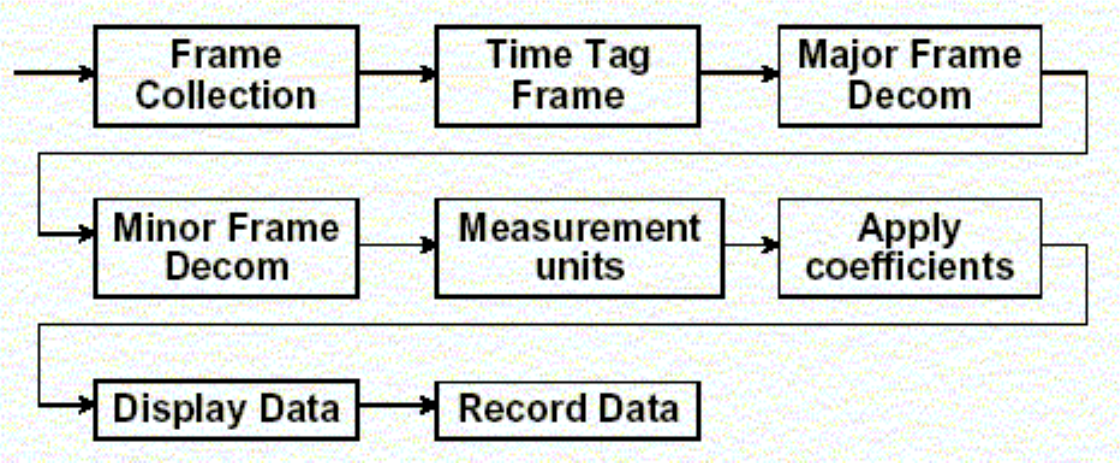
- lex/yacc-based compiler (scan, parse, generate code, ..)
- Unix pipes
- Image processing
- Signal processing
- Voice and video streaming
- …

Example Pipe-and-Filter System: Khors–An Image-Processing Workbench



Example Pipe-and-Filter System

- Telemetry Data Collection Systems
  - **Receives telemetry stream, decom frames, applies coefficients, stores data**



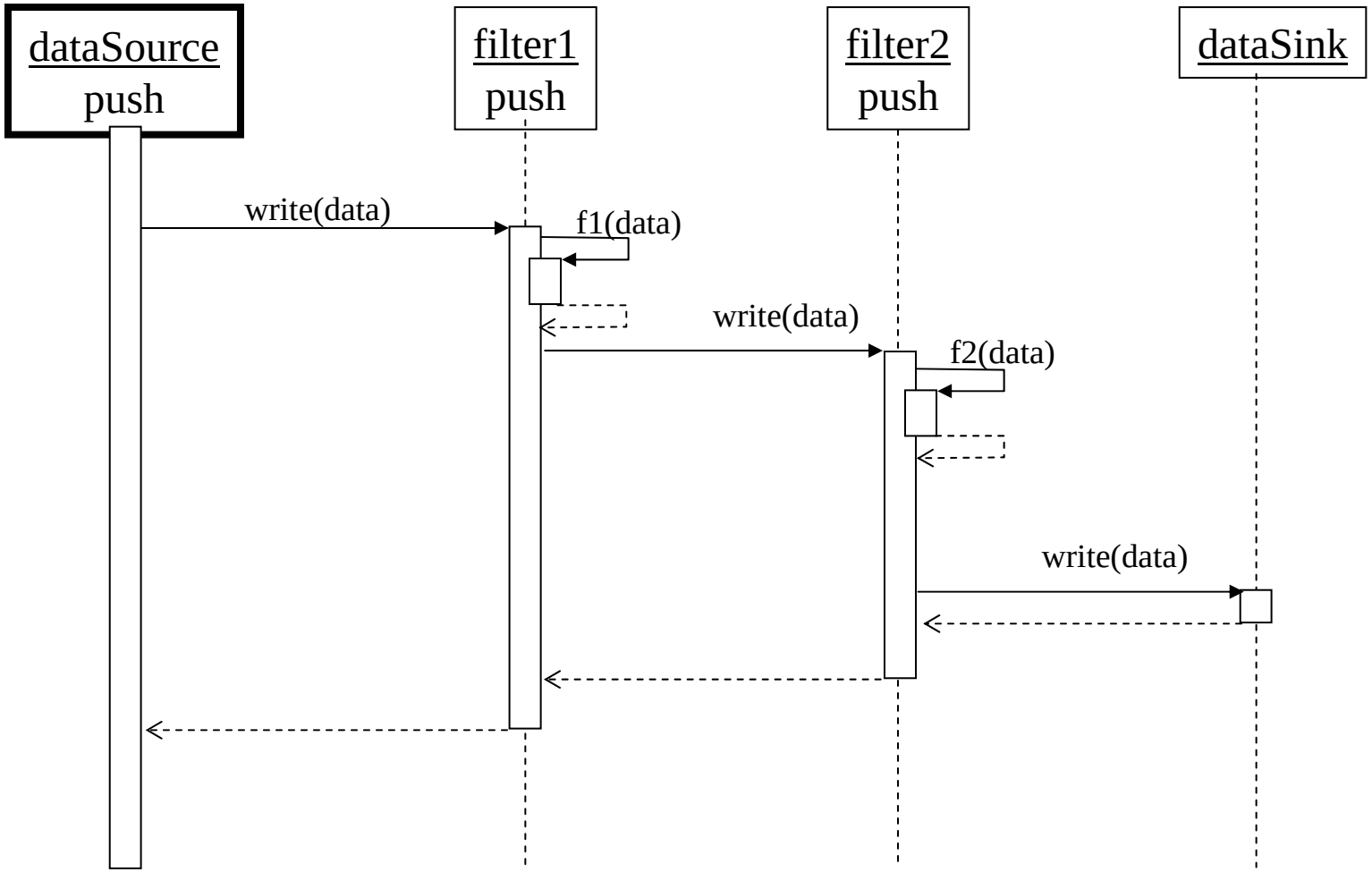
Data Pulling and Data Pushing

- What is the force that makes the data flow?

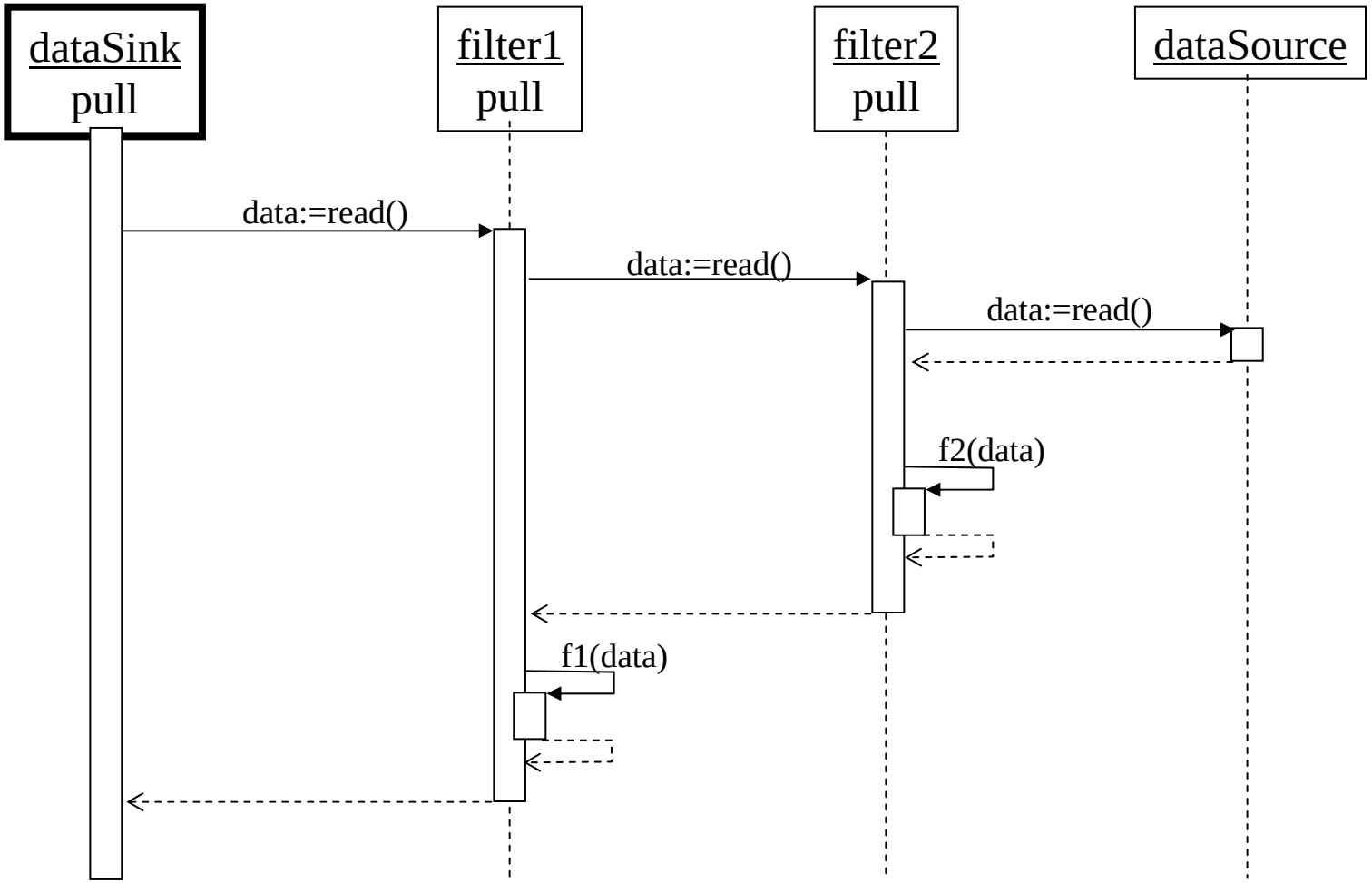
- Four choices:
  - **Push: data source pushes data in a downstream direction**
  - **Pull: data sink pulls data from an upstream direction**
  - **Push/pull: a filter is actively pulling data from a stream, performing computations, and pushing the data downstream**
  - **Passive: don’t do either, act as a sink or source for data**

- Combinations may be complex and may make the “plumber’s” job more difficult
  - **if more than one filter is pushing/pulling, synchronization is needed**

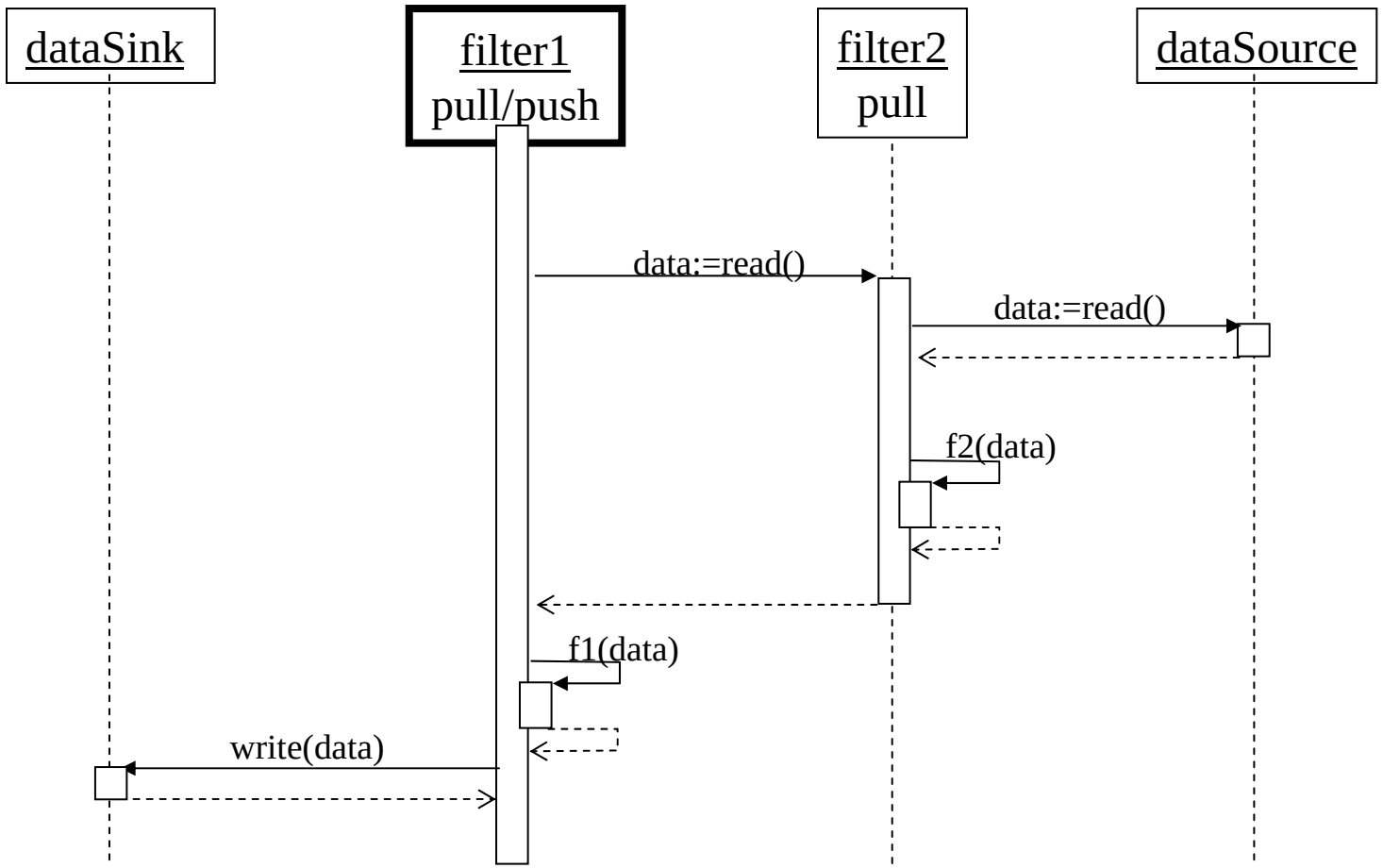
A Push Pipeline With an Active Source



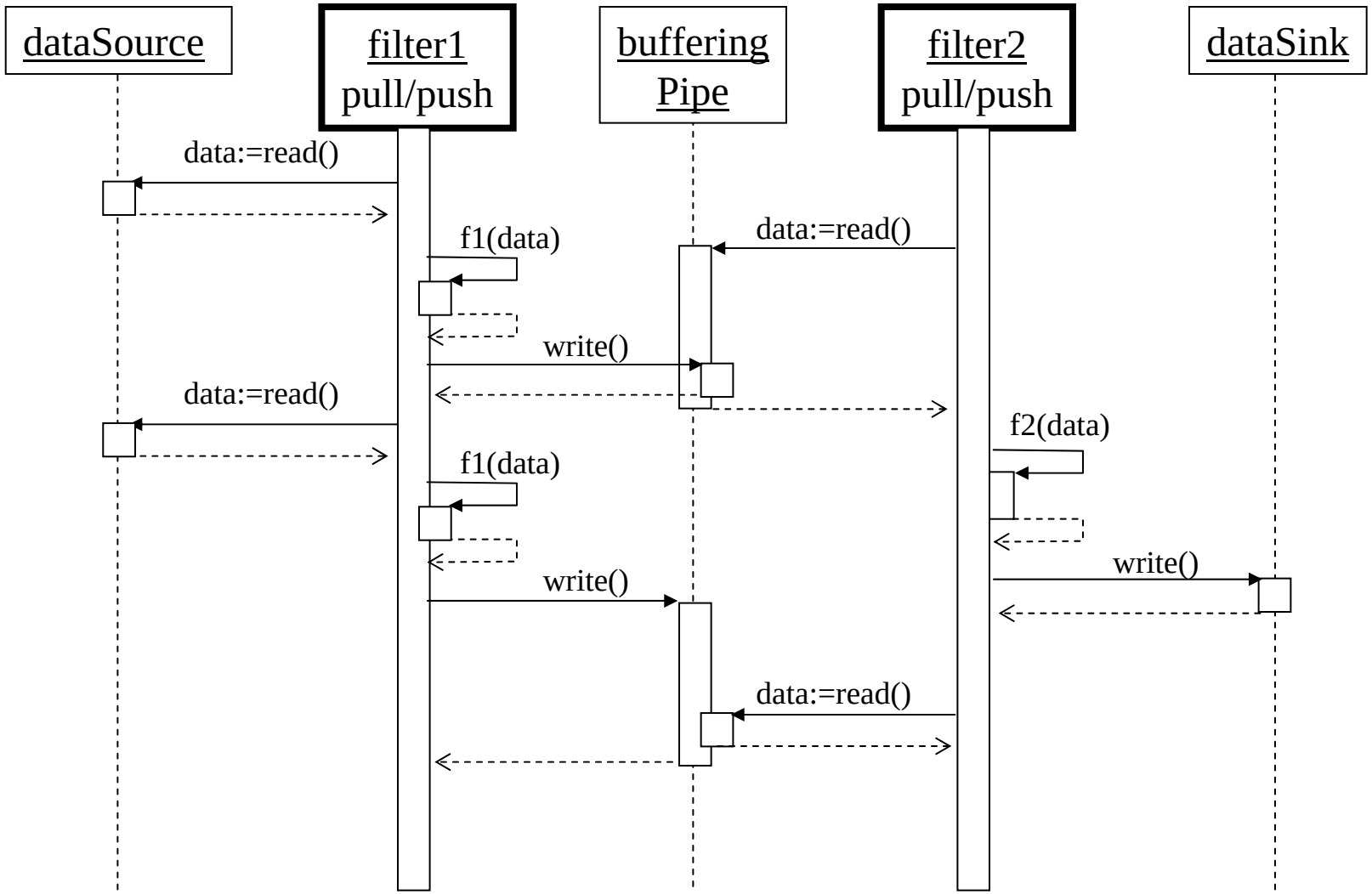
A Pull Pipeline With an Active Sink



A Mixed Push-pull Pipeline With Pasive Source and Sink



A Pipeline With Active Filters and Synchronizing Buffering Pipes



Pipe and Filter: Strengths

- Overall behaviour is a simple composition of behaviour of individual filters.
- Reuse - any two filters can be connected if they agree on that data format that is transmitted.
- Ease of maintenance - filters can be added or replaced.
- Prototyping e.g. Unix shell scripts are famously powerful and flexible, using filters such as sed and awk.
- Architecture supports formal analysis - throughput and deadlock detection.
- Potential for parallelism - filters implemented as separate tasks, consuming and producing data incrementally.

Pipe and Filter: Weaknesses

- Can degenerate to ‘batch processing’ - filter processes 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 systems, one of which is Model-View-Controller.
- Synchronisation of streams will constrain architecture.
- Error handling is Achilles heel e.g. 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 e.g. 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

Overview

- Context
- What is software architecture?
- Example: Mobile Robotics
- Architectural styles and patterns
  - **Data flow**
  - ➔ **Call-and-return**
  - **Interacting processes**
  - **Data-oriented repository**
  - **Data-sharing**
  - **Hierarchical**
  - **Other**
- Heterogeneous architectures

Call-and-return

- Main program/subroutines
- Information hiding
  - **ADT, object, naive client/server**

Main Program + Subroutine Architecture

- Classic style since 60s - pre-OO.
- Hierarchical decomposition into subroutines (Components) each solving a well defined task/function.
- Data passed around as parameters.
- Main driver provides a control loop for sequencing through subroutines.

Data Abstraction / Object Oriented

- Widely used architectural style
- Components:
  - **Objects or abstract data types**
- Connections:
  - **Messages or function/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 Object-Oriented - no inheritance.

Object-Oriented Strengths/Weaknesses

- Strengths:
  - **Change implementation without affecting clients (assuming interface doesn't change)**
  - **Can break problems into interacting agents (distributed across multiple machine / networks).**
- Weaknesses:
  - **To interact objects must know each other's identity (in contrast 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 C effects on B can be unexpected to A (and vice-versa).**
  - **Complex dynamic interactions – distributed functionality.**

Overview

- Context
- What is software architecture?
- Example: Mobile Robotics
- Architectural styles and patterns
  - **Data flow**
  - **Call-and-return**
  - ➔ **Interacting processes**
  - **Data-oriented repository**
  - **Data-sharing**
  - **Hierarchical**
  - **Other**
- Heterogeneous architectures

Interacting processes

- Communicating processes
  - **LW processes, distributed objects, …**
- Event systems
  - **implicit invocation, pure events, …**

Event-Based, Implicit Invocation

- This architectural style (pattern) is characterised by the style of communication between components:
  - **Rather than invoking a procedure directly or sending a message a component announces, or broadcasts, one or more events.**
- Basically, components communicate using a generalised Observer Design Pattern style of communication.
- BUT this is a different architectural style from Object-Oriented
  - **Communications are broadcast-based and components are not necessarily objects.**

Implicit Invocation Example

- Components register interest in an event by associating a procedure with the event.
- When the event is announced the system implicitly invokes all procedures that have been registered for the event.
- Common style for integrating tools in a shared environment, e.g.,
  - **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.**

Implicit Invocation Example

- Examples:
  - **Editor announces it has finished editing a module, compiler registers for such announcements and automatically re-compiles module.**
  - **Debugger announces it has reached a breakpoint, editor registers interest in such announcements and automatically scrolls to relevant source line.**

Implicit Invocation

- Components
  - **Modules whose interfaces provide a collection of procedures/methods and a set of events that it may announce**
- Connectors
  - **Bindings between event announcements and procedure/method calls**
  - **Traditional procedure/method calls (to bypass implicit invocation)**

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 Examples (Shaw and Garlan textbook)
  - **Programming environment tool integration**
  - **User interfaces - Model-View-Controller**
  - **Syntax-directed editors to support incremental semantic checking**

Implicit Invocation

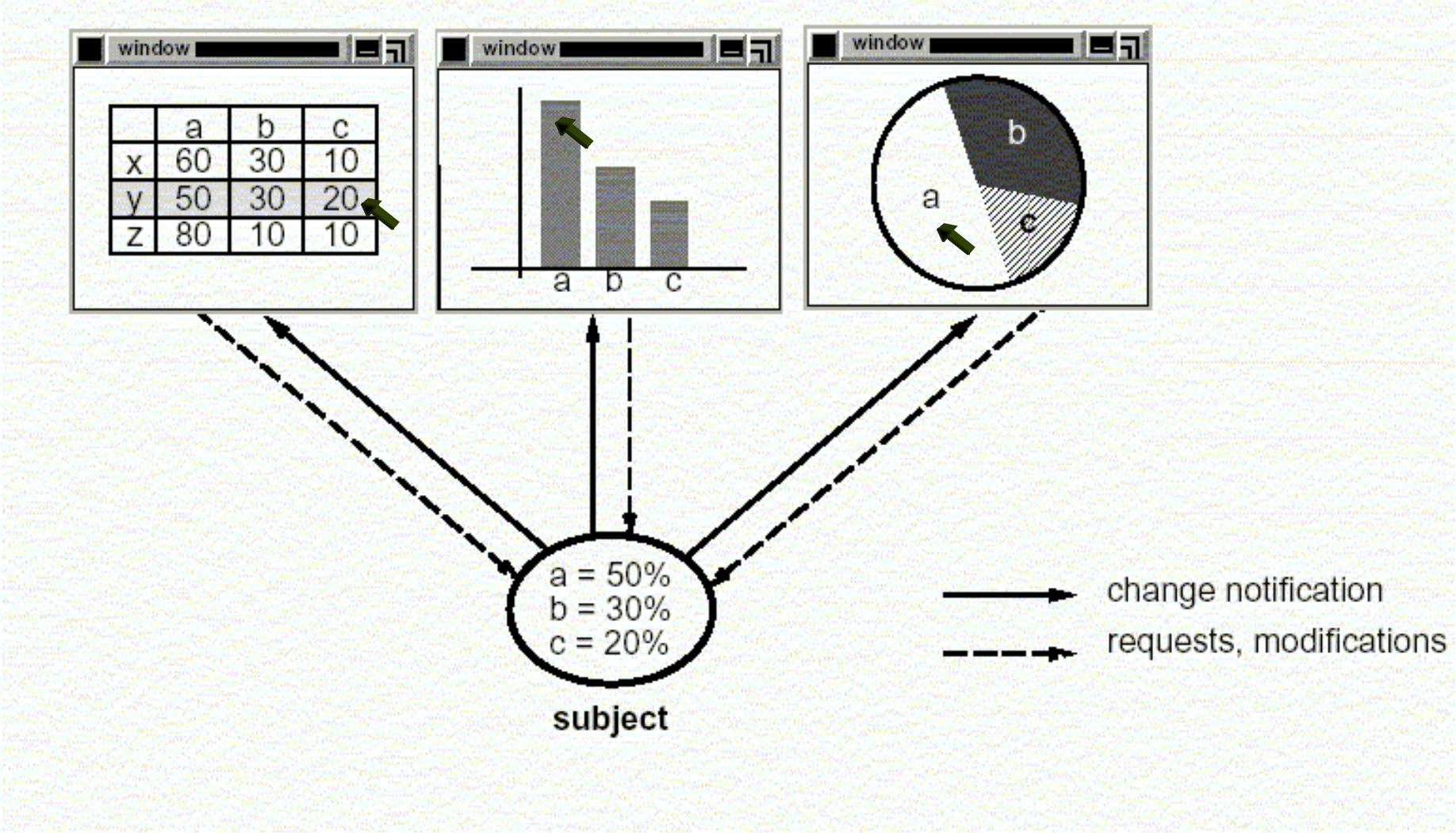
- Strengths
  - **Strong support for reuse - plug in new components by registering it for events**
  - **Maintenance - add and replace components with minimum affect on other components in the system.**

Implicit Invocation

- Weaknesses
  - **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 because it depends on context in which invoked. Unpredictable interactions.**
  - **Sharing data - see the Observer Design Pattern**

- Hence explicit invocation is usually provided as well as implicit invocation. In practice architectural styles are combined.

Model-View-Controller



Model-View-Controller

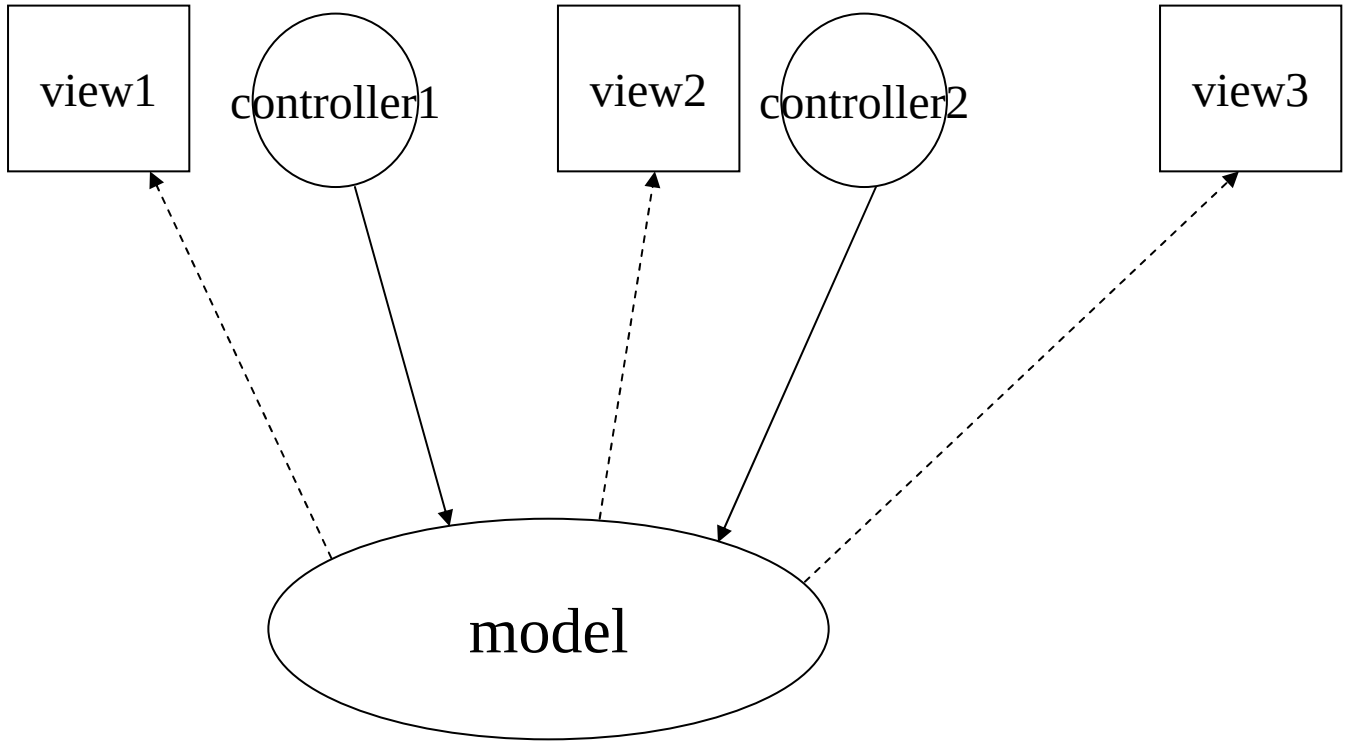
- A decomposition of an interactive system into three components:
  - **A model containing the core functionality and data,**
  - **One or more views displaying information to the user, and**
  - **One or more controllers that handle user input.**

- A change-propagation mechanism (i.e., observer) ensures consistency between user interface and model, e.g.,
  - **If the user changes the model through the controller of one view, the other views will be updated automatically**

- Sometimes the need for the controller to operate in the context of a given view may mandate combining the view and the controller into one component

- The division into the MVC components improves maintainability

Model-View-Controller



Overview

- Context
- What is software architecture?
- Example: Mobile Robotics
- Architectural styles and patterns
  - **Data flow**
  - **Call-and-return**
  - **Interacting processes**
  - ➔ **Data-oriented repository**
  - **Data-sharing**
  - **Hierarchical**
  - **Other**
- Heterogeneous architectures

Data-Oriented Repository

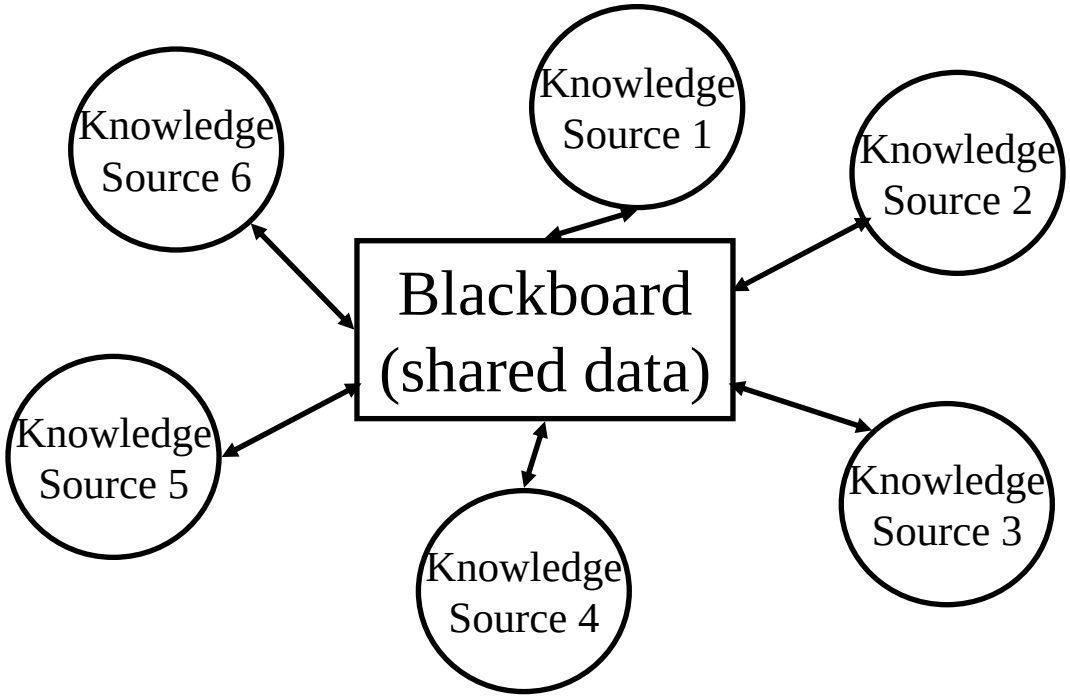
- Transactional databases
  - **True client/server**
- Blackboard
- Modern compiler

Repositories / Data Centred

- Characterised by a central data store component representing systems state and a collection of independent components that operate on the data store.
- Connections between data store and external components vary considerably in this style:
  - ***Transactional databases*: Incoming stream of transactions trigger processes to act on data store. Passive.**
  - ***Blackboard architecture*: Current state of data store triggers processes. Active.**

Blackboard

- Characteristics: cooperating ‘partial solution solvers’ *collaborating but not following a pre-defined strategy.*
- Current state of the solution stored in the blackboard.
- Processing triggered by the state of the blackboard.



Examples of Blackboard Architectures

- 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 (see POSA case study)**
  - **Modern compilers act on shared data: symbol table, abstract syntax tree (see Garlan and Shaw case study)**

Overview

- Context
- What is software architecture?
- Example: Mobile Robotics
- Architectural styles and patterns
  - **Data flow**
  - **Call-and-return**
  - **Interacting processes**
  - **Data-oriented repository**
  - ➔ **Data-sharing**
  - **Hierarchical**
  - **Other**
- Heterogeneous architectures

Data-sharing

- Compound documents
- Hypertext
- Fortran COMMON
- LW processes

Overview

- Context
- What is software architecture?
- Example: Mobile Robotics
- Architectural styles and patterns
  - **Data flow**
  - **Call-and-return**
  - **Interacting processes**
  - **Data-oriented repository**
  - **Data-sharing**
  - ➔ **Hierarchical**
  - **Other**
- Heterogeneous architectures

Hierarchical

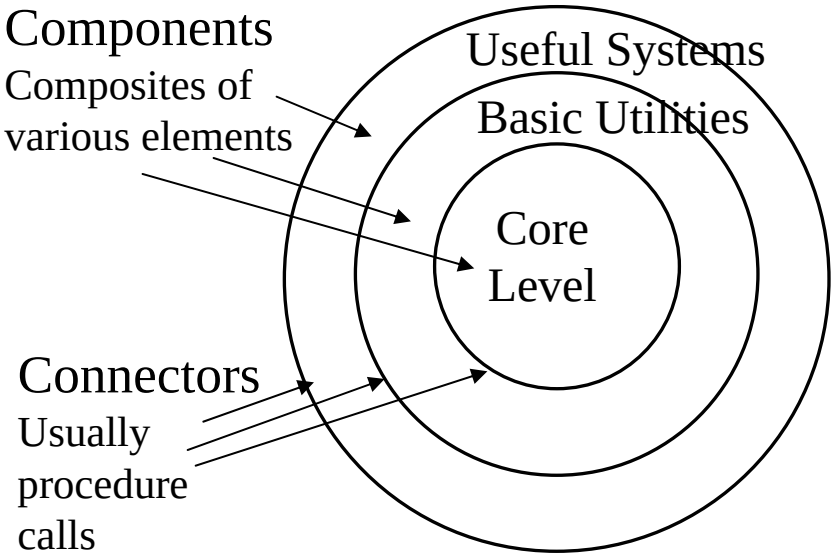
- Layered
  - **Interpreter**
  - **Tiered**

Layered Systems

- “A layered system is organised hierarchically, each layer providing service to the layer above it and serving as a client to the layer below.” (Garlan and Shaw)
- Each layer collects services at a particular level of abstraction.
- In a pure layered system: Layers are hidden to all except adjacent layers.

Layered Systems

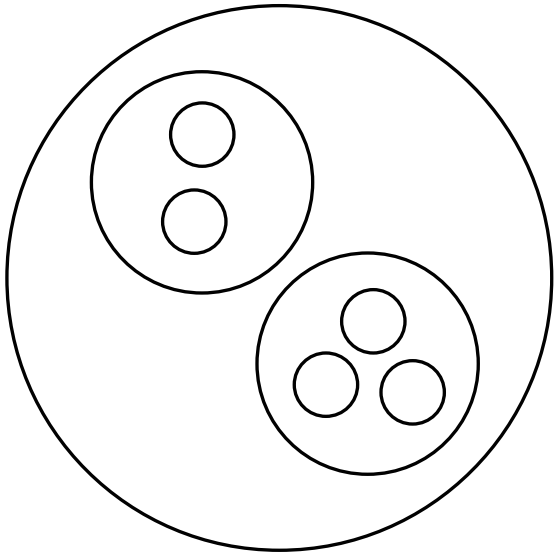
- “Onion Skin model”…

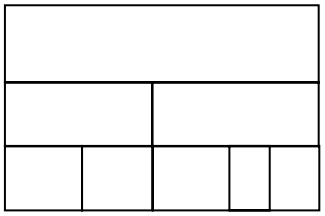


- corresponds to a stack of layers.

Useful Systems
Basic Utilities
Core Level

Hierarchical systems





- Hierarchical systems can be tree-like in general

Layered Systems

- Applicability
  - **A large system that is characterised 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**

Layered Systems

- Invariants
  - **Limit layer (component) interactions to adjacent layers (in practice this may be relaxed for efficiency reasons)**
- Typical variant relaxing the pure style
  - **A layer may access services of all layers below it**
- Common Examples
  - **Communication protocols: each level supports communication at a level of abstraction, lower levels provide lower levels of communication, the lowest level being hardware communications.**

Layered System Examples

- Example 1: ISO defined the OSI 7-layer architectural model with layers: Application, Presentation, …, Data, Physical.
  - **Protocol specifies behaviour at each level of abstraction (layer).**
  - **Each layer deals with specific level of communication and uses services of the next lower level.**

- Example 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 a network communicate ‘virtually’.

- Example 3: Operating systems e.g. hardware layer, …, kernel, resource management, … user level “Onion Skin model”.
- ...

Layered Systems

- Strengths
  - **Increasing levels of abstraction as we move up through layers – partitions complex problems**
  - **Maintenance - in theory, a layer only interacts with layers above and below. Change has minimum effect.**
  - **Reuse - different implementations of the same level can be interchanged**
  - **Standardisation based on layers e.g. OSI**

Layered Systems

- Weaknesses
  - **Not all systems are easily structured in layers (e.g., mobile robotics)**
  - **Performance - communicating down through layers and back up, hence bypassing may occur for efficiency reasons**

- Similar strengths to data abstraction / OO but with multiple levels of abstraction (e.g. well-defined interfaces, implementation hidden).

- Similar to pipelines, e.g., communication with at most one component at either side, but with 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 a virtual machine produced in software.
- Special kind of a 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.
- Example: Java Virtual Machine. Java code translated to platform independent bytecodes. JVM is platform specific and interprets (or compiles - JIT) the bytecodes.

Tiered Architectures

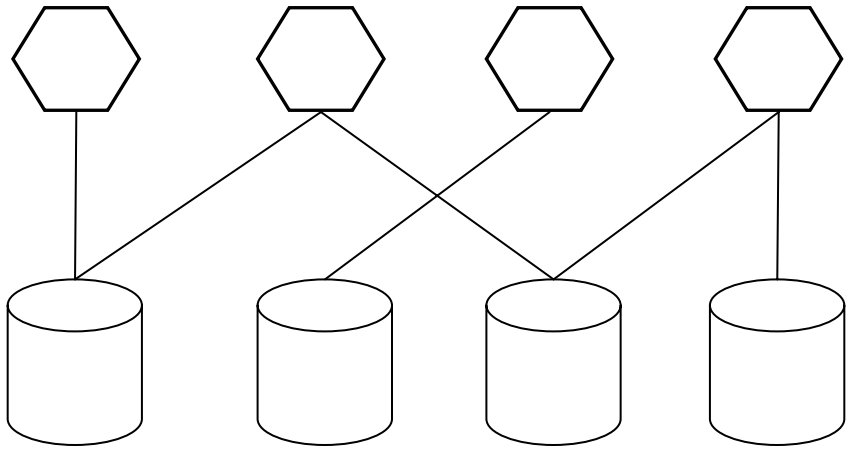
- Special kind of layered architecture for enterprise applications
- Evolution
  - **Two Tier**
  - **Three Tier**
  - **Multi Tier**

Two Tier Client Server Architecture Design

- Developed in the 1980s 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
- See http://www.sei.cmu.edu/str/descriptions/twotier.html#512860

Client tier
User System Interface
+ Some Processing
Management

Server tier
Database Management
+ Some Processing
Management



Three Tier Client Server Architecture Design

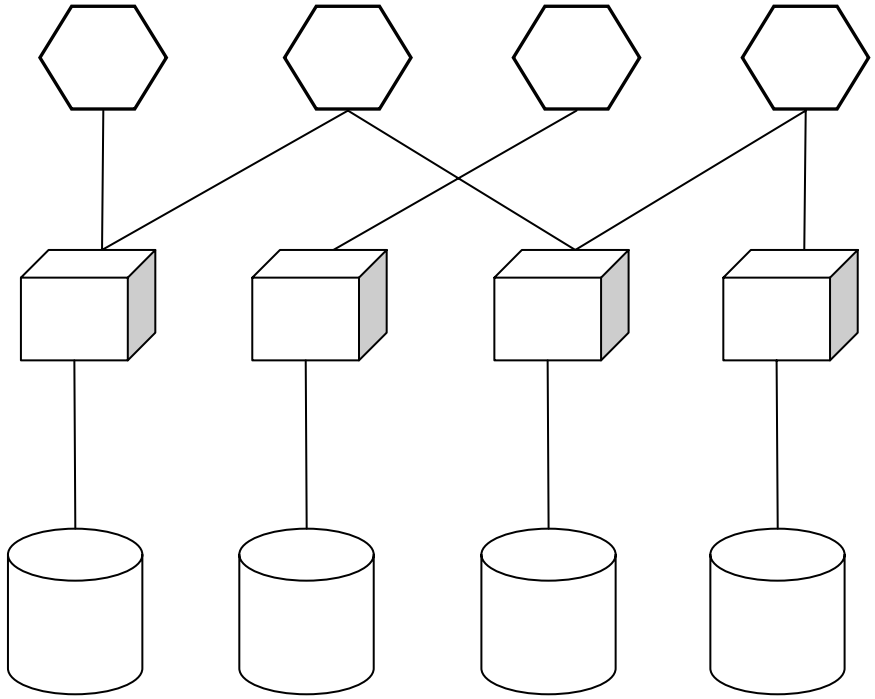
- Emerged in the 1990s to overcome the limitations of the two tier architecture by adding an additional middle tier.
- This middle tier provides process management where business logic and rules are executed and can accommodate hundreds of users by providing generic services such as queuing, application 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 from the user.
- See http://www.sei.cmu.edu/str/descriptions/threetier.html

Three Tier Client Server Architecture Design

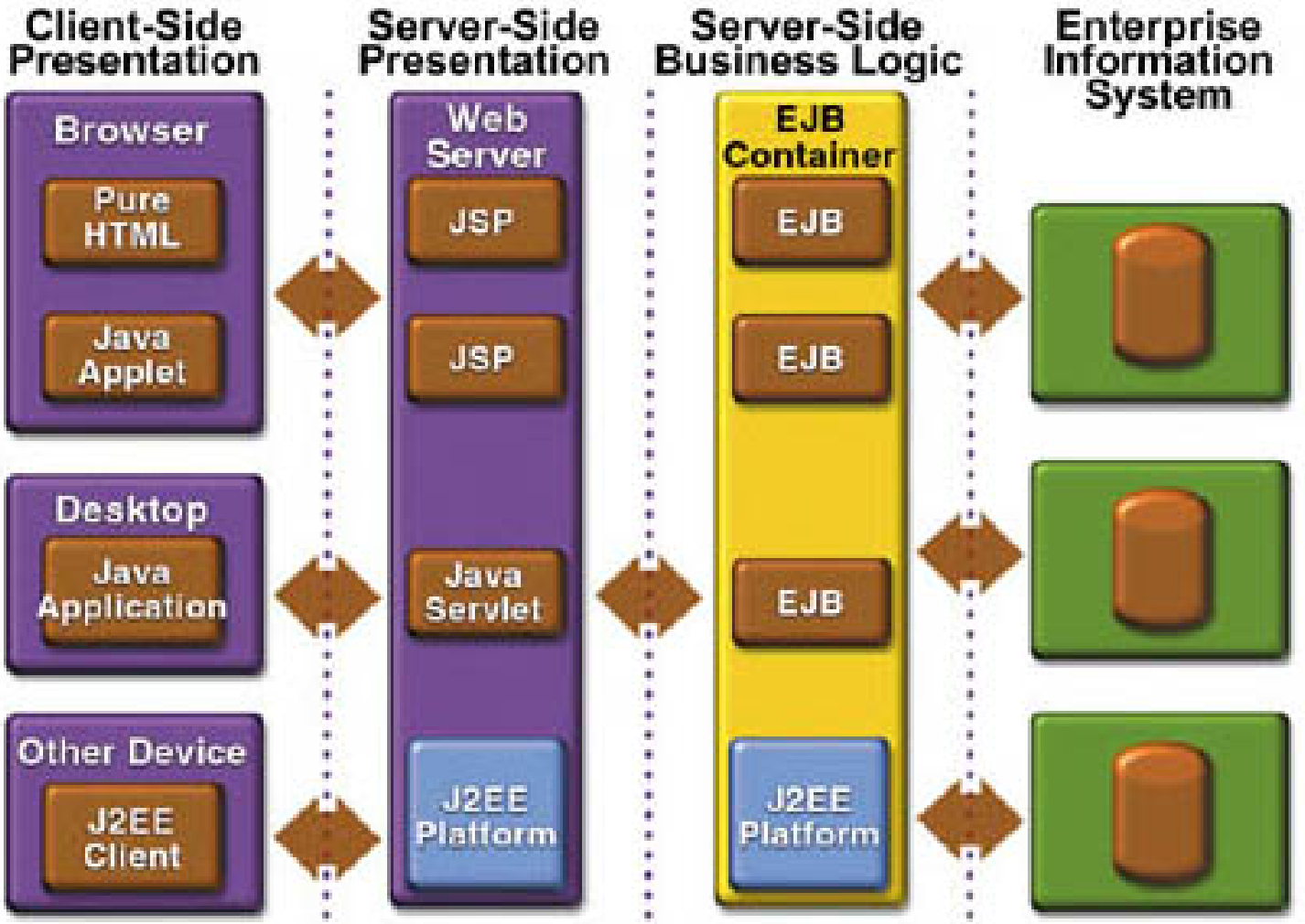
User System Interface

Processing Management

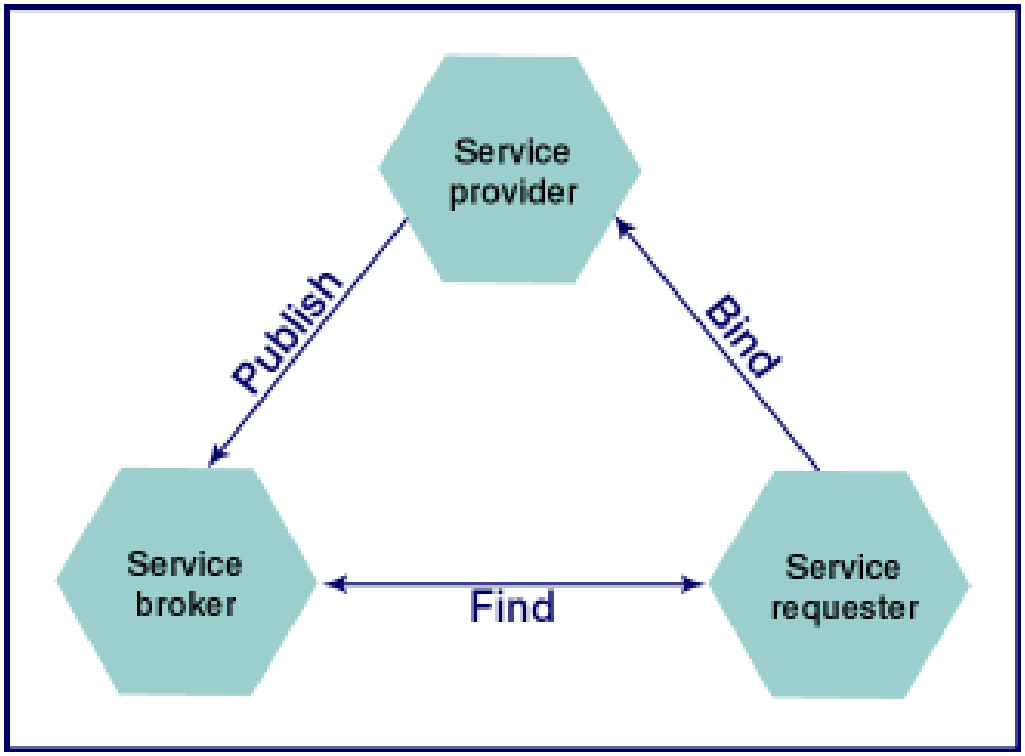
Database Management



Example of a Multi Tier Architecture: Java 2 Platform, Enterprise Edition (J2EE)



Service Oriented Architecture (SOA)



Overview

- Context
- What is software architecture?
- Example: Mobile Robotics
- Architectural styles and patterns
  - **Data flow**
  - **Call-and-return**
  - **Interacting processes**
  - **Data-oriented repository**
  - **Data-sharing**
  - **Hierarchical**
  - ➔ **Other**
- Heterogeneous architectures

Other Architectures...

- Distributed Systems:
  - **Common organisations for multi-process systems characterised either by topological organisation e.g. ring or star, and inter-process protocols e.g. client-server architectures.**
  - **Broker pattern: An arrangement where decoupled components interact by remote service invocations. A broker component is responsible for coordinating communication and for transmitting results and exceptions.**
- Process Control Systems:
  - **Dynamic control of physical processes based on a feedback loop.**

POSA Architectural Patterns

- These were already discussed:
  - **Layers pattern**
  - **Pipes and filters pattern**
  - **Blackboard pattern**
  - **Model-view-controller pattern**
  - **Broker pattern**

POSA Architectural Patterns for Adaptable Systems

- Microkernel pattern
  - **An arrangement that separates a minimal functional core from extended functionality and customer-specific parts.**
  - **The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.**
- Reflection pattern
  - **Organize a system into a base level performing the actual functionality and a meta-level providing a runtime, explicit configuration model of the base level.**
  - **The metalevel makes software self-aware by allowing to inspect and possibly reconfigure itself through the metalevel**

Overview

- Context
- What is software architecture?
- Example: Mobile Robotics
- Architectural styles and patterns
  - **Data flow**
  - **Call-and-return**
  - **Interacting processes**
  - **Data-oriented repository**
  - **Data-sharing**
  - **Hierarchical**
  - **Other**

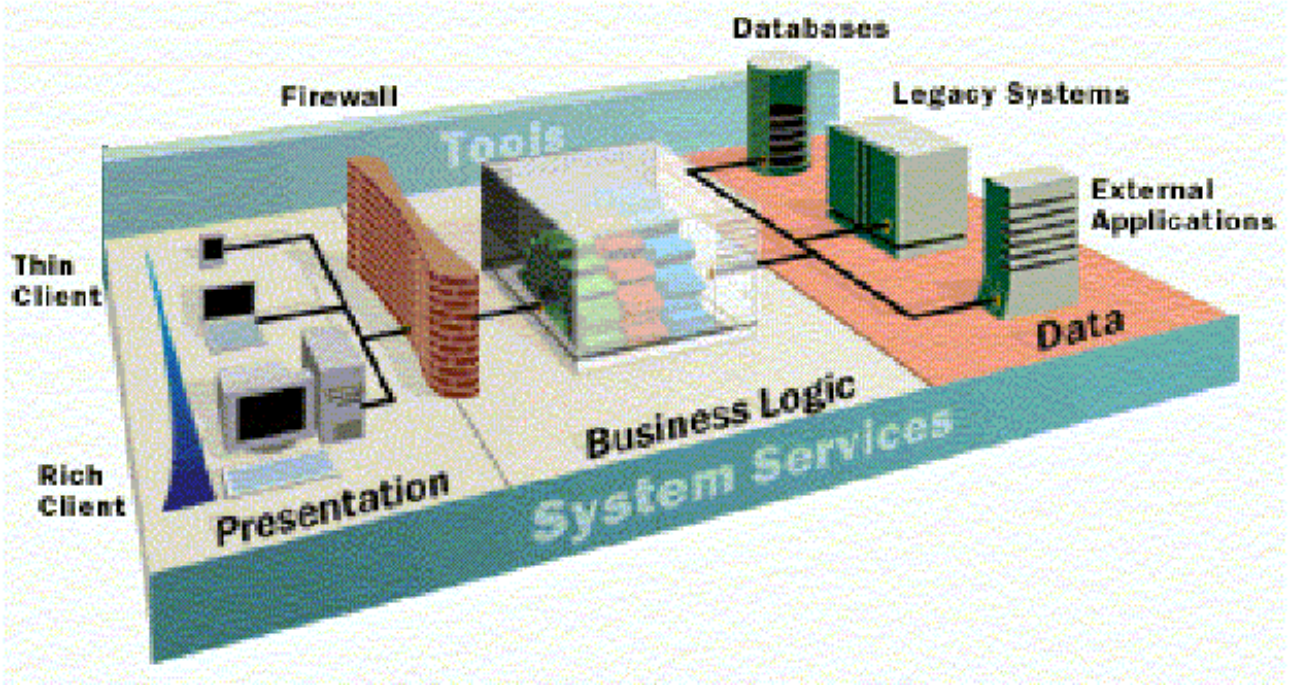
➔ Heterogeneous architectures

Heterogeneous Architectures

- In practice the architecture of large-scale system is a combination of architectural styles:
  - **('Hierarchical heterogeneous') A Component in one style may have an internal style developed in a completely different style (e.g, pipe component developed in OO style, implicit invocation module with a layered internal structure, etc.)**
  - **('Locational heterogeneous') Overall architecture at same level is a combination of different styles (e.g., repository (database) and mainprogram-subroutine, etc.)**
    Here individual components may connect using a mixture of architectural connectors - message invocation and implicit invocation.
  - **('Perspective heterogeneous') Different architecture in different perspectives (e.g., structure of the logical view, structure of the physical view, etc.)**

Example of Heterogeneous Architectures: Enterprise Architectures

- Multi tier (at the highest level), distributed (including broker pattern), transactional databases, event-based communication, implicit invocation, object-oriented, MVC (e.g., for presentation in the client), dataflow for workflow, etc.



Overview

- Context
- What is software architecture?
- Architectural styles and patterns
  - **Data flow**
  - **Call-and-return**
  - **Interacting processes**
  - **Data-oriented repository**
  - **Data-sharing**
  - **Hierarchical**
  - **Other**
- ➔ Heterogeneous architectures