CSC 3380 Aymond

Section 1

2/5/2020

Homework Assignment

- Moodle Quiz, Due 11PM 2/6
- Milestone #1 self assessment
- A milestone self assessment will be due 48 hours after each milestone deadline

Project News

- Next Milestone: #2
  - Due Friday 2/21, 11PM
  - Upload to Moodle (1 upload for entire team)
  - Outline is in Project Kickoff Lecture Notes
  - All UML diagrams must be developed in EA

Object Oriented Design

- Source Control++
- **The Design Process**
- Software System Architecture
- Architectural Styles
- Object Oriented Design Principles

Design For Change

- Assume that anything that can change, will change
  - Environment (e.g., OS)
  - External dependencies
    - Data source
    - Communication protocol
    - Connectivity
  - Embedded systems
- Plan to scale up
  - Number of users
  - Size of data
  - Variability in services
- Create general solutions, not specific ones
  - For example, rather than creating specialized functions:
    - Square, rectangle, rhombus, etc.
  - Create a single function that calculates the area of a polygon of any shape.

Desirable Features…

- Fitness for purpose
  - The system must work, and work correctly
  - It should
    - perform the required tasks
    - in the specified manner and
    - within the specified constraints
    - of the specified resources
- Robustness
  - The design should be **stable against changes** such as features as file and data structures, user interface, etc.

Desirable Design Features

- Simplicity
  - The design should be as simple as possible, but no simpler
- Separation of concerns
  - The different concepts and components should be separated out (modular)
- Information hiding
  - Information about the detailed form of such objects as data structures and device interfaces should
    - be kept local to a module or unit
    - Not be directly "visible" outside that unit

Undesirable Features

- Having too much retained state information spread around the system
- Using interfaces that are too complex
- Containing excessively complex control structures
- Involving needless replication

Principles of Design

- Abstraction
- Information Hiding
- Completeness
- Design for reuse
- Modularity
  - Cohesion
    - Relationships among the elements making up a component
  - Coupling
    - Interdependencies between different component
  - Goal: High cohesion/Low Coupling

Degree of Coupling

- Uncoupled
  - components are independent
- Data coupling
  - data is passed between components
- Stamp coupling
  - data structure is passed between components
- Control coupling
  - one component passes controlling parameter to another
- Common coupling
  - common data store
- Content coupling
  - one component modifies another

Degree of Cohesion

- Functional
  - Every module is essential to the single component function
- Sequential
  - Output of one module is input for another module
- Communicational
  - Modules operate on or produce same dataset
- Procedural
  - Modules a only related by sequence
- Temporal
  - Modules are only related by timing
- Logical
  - logically related modules are grouped together
- Coincidental
  - modules are unrelated to one another

Assessing a Design

- Major Issues
  - Completeness and correctness
    - How well the design meets the specification
  - Quality
    - How well-structured the design is
- Any assessment must also consider
  - The static structures of the design
  - The dynamic performance of the implemented system

Design Strategies

- Top-down
  - Functional decomposition
  - Stepwise refinement at the component level
- Bottom up
  - Composition
  - Design pieces in isolation before deciding how they will fit together as a whole
- Stylized
  - Pattern (re)use
  - Good solution already exists

- There is a place for all of these strategies in software and software system design
  - Start with top-down architecture design
  - Components are handed off to development team for bottom-up software design
  - Patterns are reused at both the architecture and software design levels, where appropriate

Object Oriented Design

- Source Control++
- The Design Process
- **Software System Architecture**
- Architectural Styles
- Object Oriented Design Principles

The Software Problem

- "The typical software system is a kludge, resulting in
  - unpredictability
  - poor quality
  - increasingly difficult to change
  - hard to reuse
  - morale problems
  - losing ground in the market"
    - Bredemeyer Consulting, 2002

Kludge

- Without architecture, we get a kludge
- defined in Webster's as:
  - A system, *especially a computer system*, that is constituted of poorly matched elements or of elements originally intended for other applications
  - **A clumsy or inelegant solution to a problem**

Software Architecture

- Software system architecture is the overall ***shared vision*** of the software system
  - The high level design of a software system
- The fundamental organization of a system, embodied in its major *components*
  - Each component represents a broad category of functionality
  - Components represent possibly many classes that work together as one
- Their *relationships* to each other and the environment
  - Components are linked with *connections*
  - Each connection represents potentially numerous communication channels
- The *principles* governing its design and evolution

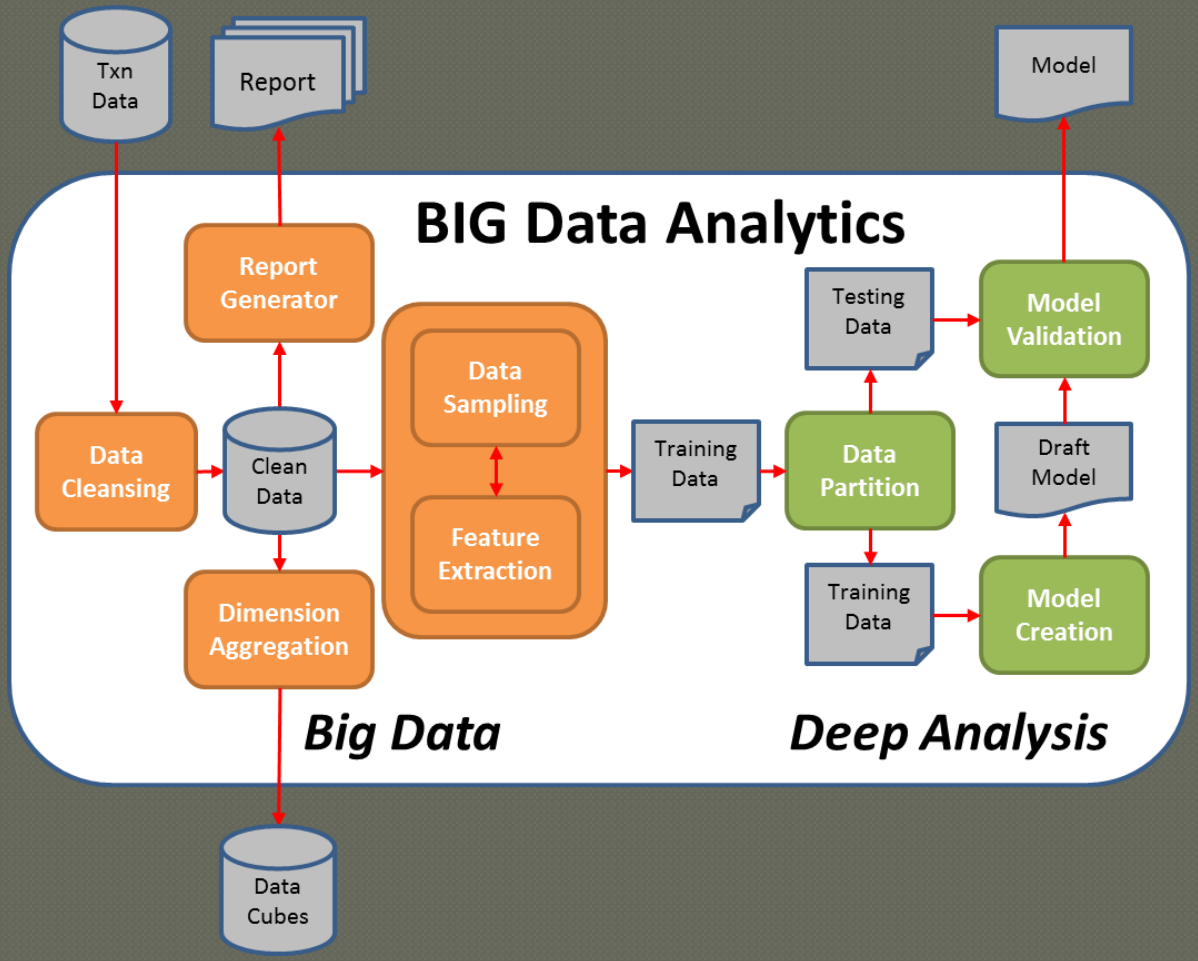
Important Properties

- High-enough level of abstraction that the system can be viewed as a whole
- Structure must support the functionality required of the system
- Structure must conform to the system qualities (e.g. performance, security, reliability flexibility, and extensibility)
- At the architectural level, all ***implementation details are hidden***

Minimalist Architecture Principle

- Keep your architecture decision set as small as it possibly can be, while still meeting your architectural objectives
- If a decision can be delegated to someone with a more narrow scope of responsibility, then do so!

Software System Architecture Example



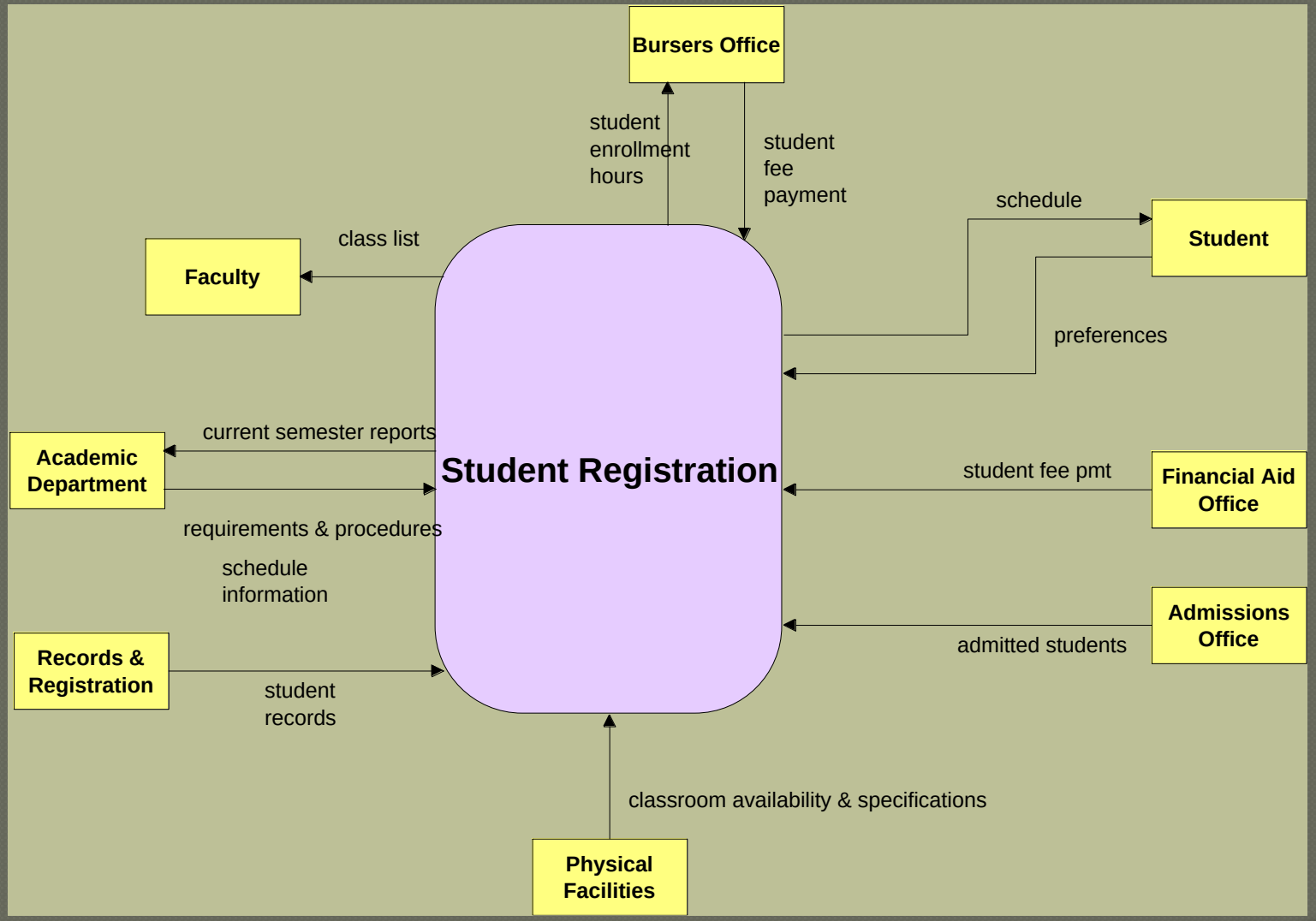
- Best Practice: **Approximately 7 components (± 2)**
- Curved rectangles represent components
- Like-colored components represent coupled components
- Cylinders represent data repositories (usually databases)
- Waved bottom rectangles represent files
- Dog-eared rectangles represent data sets

http://horicky.blogspot.com/2012/08/big-data-analytics.html

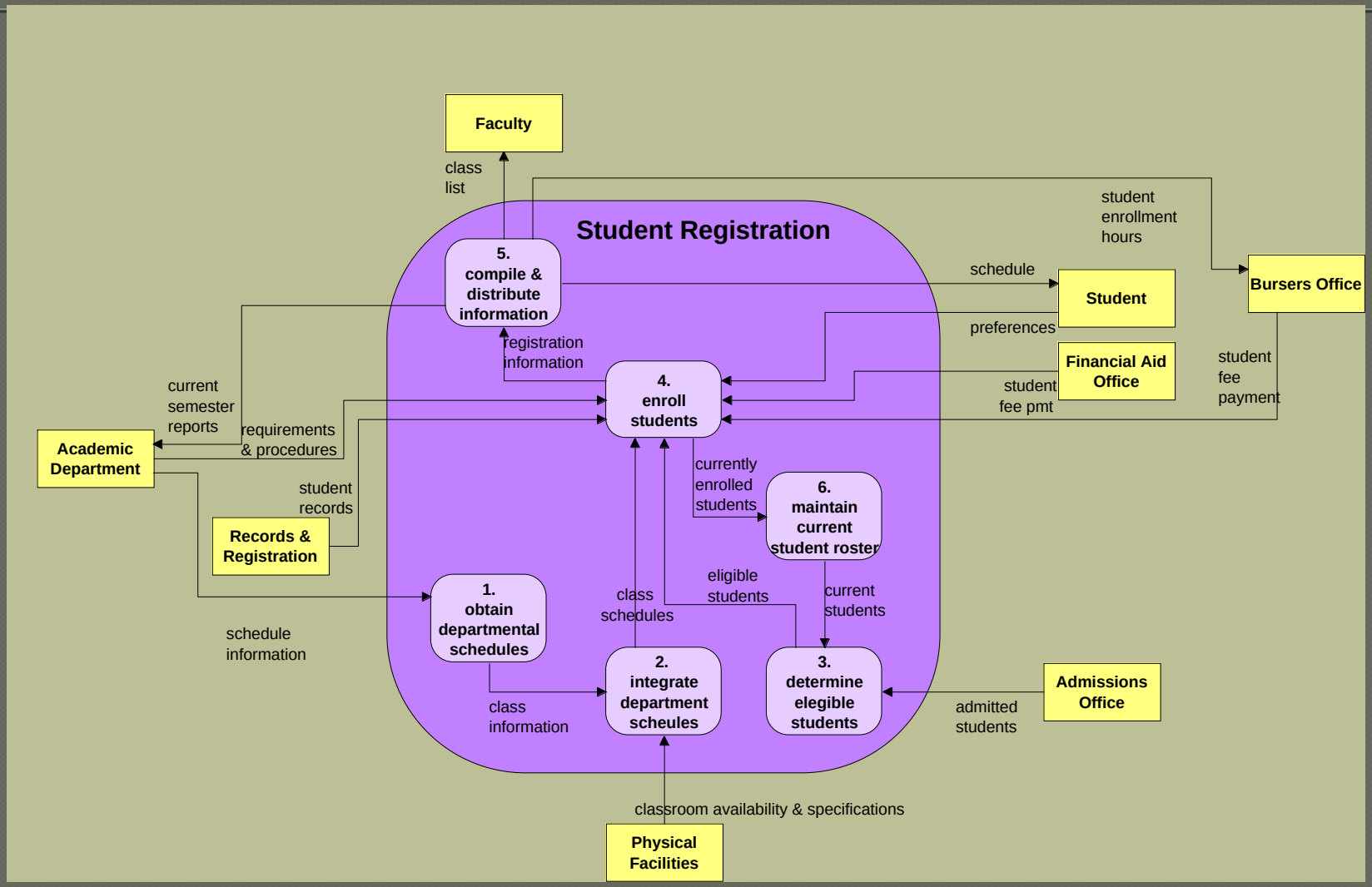
Data Flow Diagrams

- Start with the Architecture Diagram
- Add data that is exchanged between components to all diagram edges
- Refine diagrams sufficiently to hand off to development team
  - Level 0
    - Software system is a black box
    - Data flow is between software system and external entities
  - Level 1
    - Software System Architecture Diagram
      - Numbers are added to components
    - Data flow between system components
  - Level 2
    - Component Design Diagrams
      - Numbers are added to subcomponents
    - Data flow between subcomponents
  - etc.

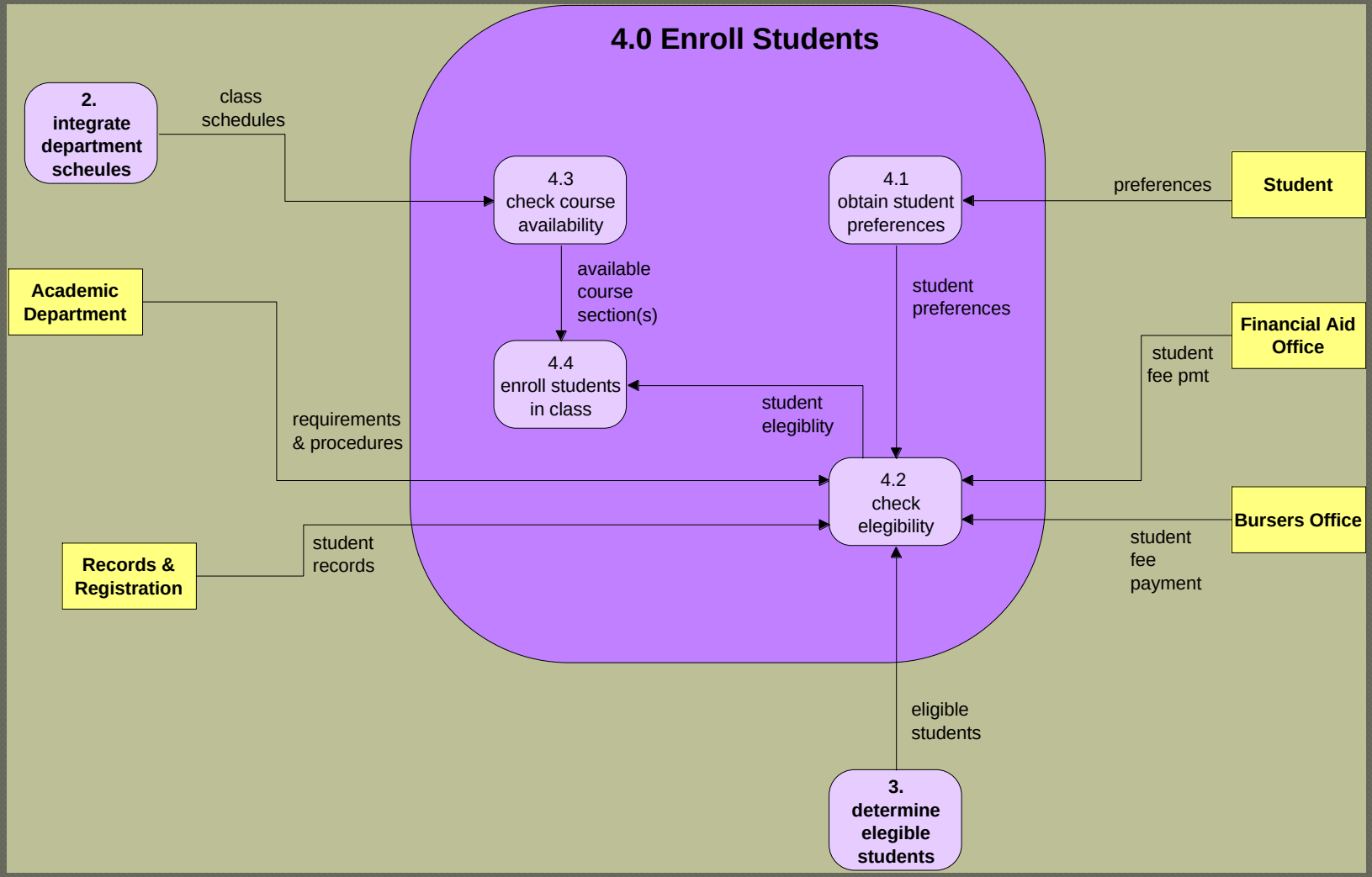
Data Flow Diagram Example: Level 0



Data Flow Diagram Example: Level 1



Data Flow Diagram Example: Level 2



Object Oriented Design

- Source Control++
- The Design Process
- Software System Architecture
- **Architectural Styles**
- Object Oriented Design Principles

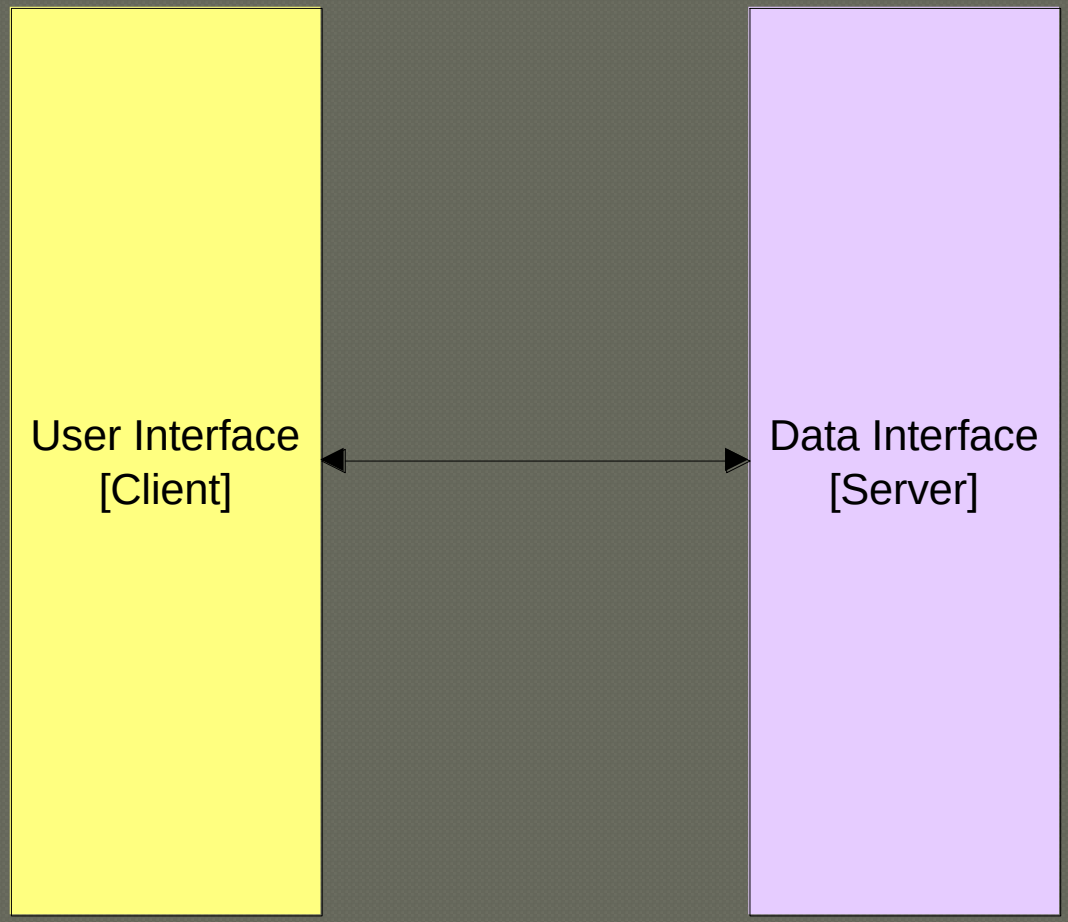
Architectural Styles

- **Client/Server**
- Data Centric
- Peer-to-Peer
- Pipe and Filter
- Model/View/Controller
- Publish/Subscribe

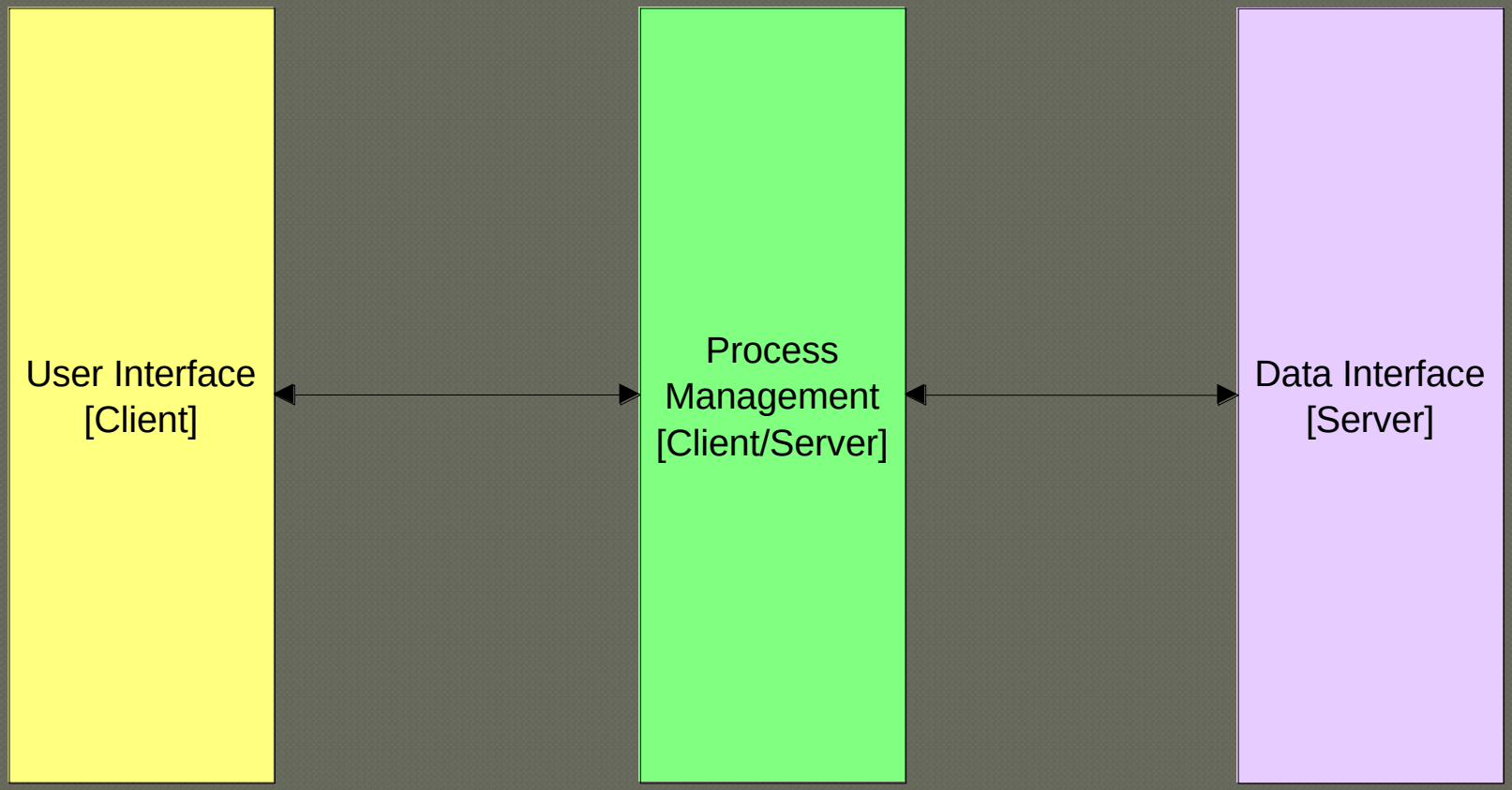
Client/Server

- A network architecture in which each computer or process on the network is either a client or a server.
  - Servers are processes dedicated to managing resources, such as disk drives (file servers), printers (print servers), network traffic (network servers), or computational services
    - The server "guards" access to the important resources that it manages
  - Clients are programs, which are run by users or specialized applications. Clients rely on servers for resources, such as files, devices, data, and computations (e.g., processing power)
    - Clients connect to the server

Client Server Abstraction: 2-tier Architecture

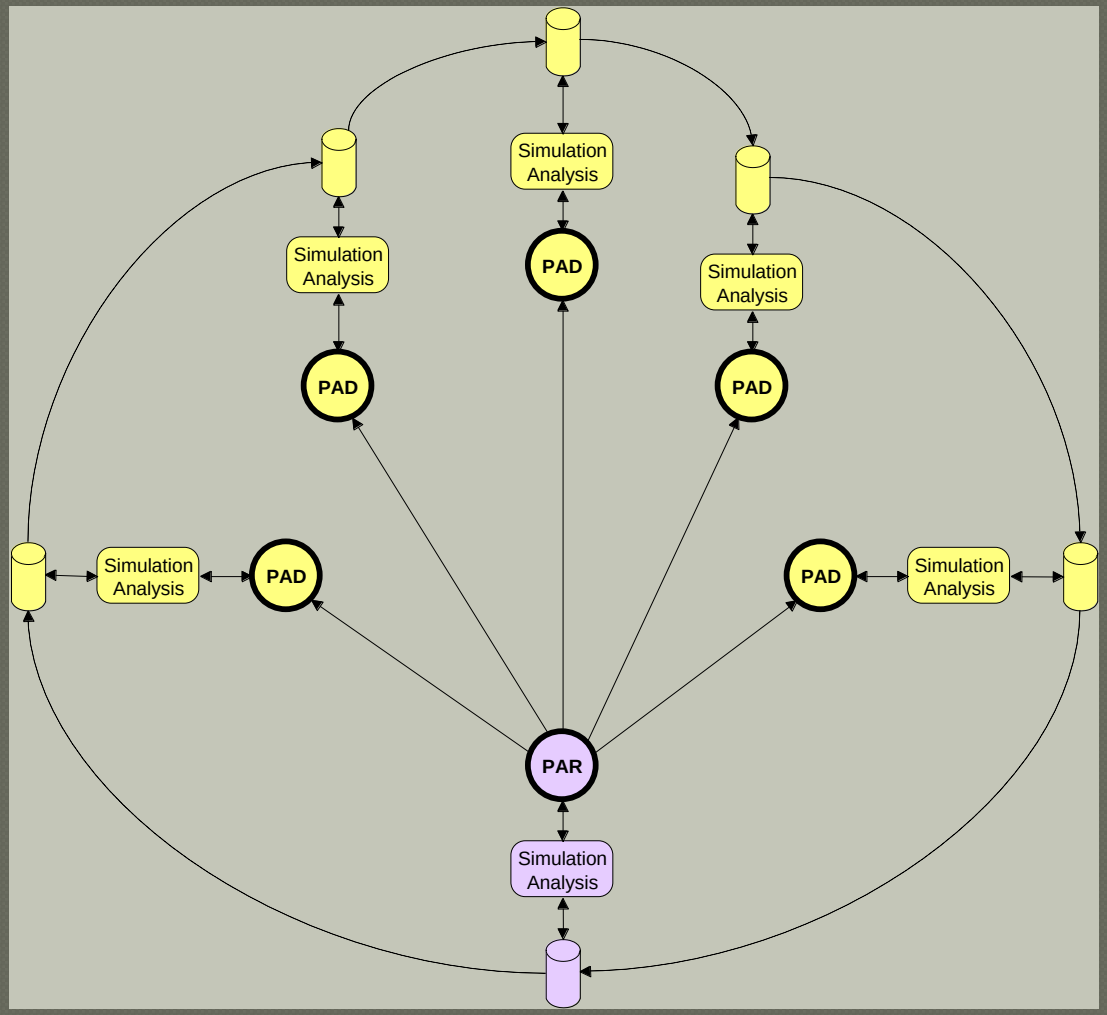


Client Server Abstraction: 3-tier Architecture



- Process Management acts as a server to the User Interface client
- It acts as a client to the Data Interface Server

Client/Server Example
US Army's PAR/PAD Generator



Other Examples of Client/Server

- Multiplayer videogames:
  - Clients absolutely cannot be trusted to change gamestate directly
  - Server validates actions and ensures rules are followed
  - Doesn't help with client side exploits (wallhacks)
- X11 windowing system
  - Allows a windowing application to be separated from a "view"
  - Streams drawing commands over network
  - Allows low-latency screen sharing, but still being replaced

A Note on Using Web Servers

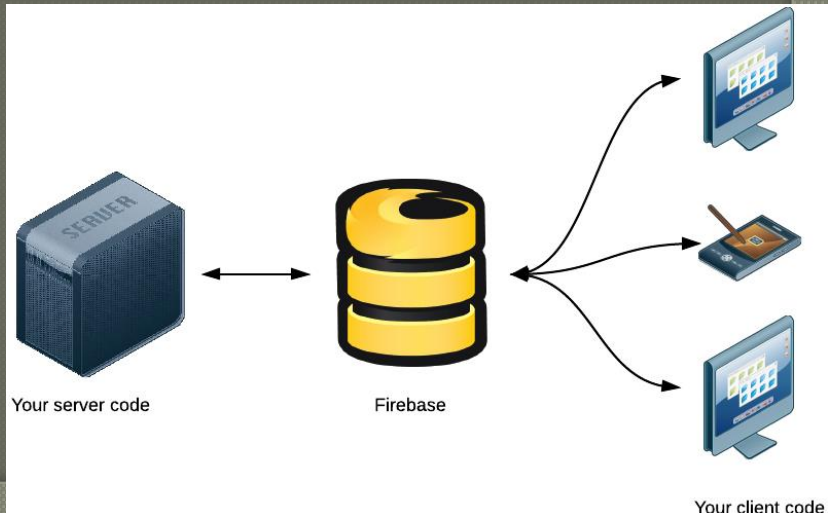
- If you plan on developing a web app, don't rewrite a webserver yourself
- The absolute easiest way to have a web-based server is for it to be a (fast) CGI application
- Install an http server on your host
  - Nginx is good one
  - Apache is widely supported
- Configure it to call your program
- Your server application will return either HTML or JSON

Architectural Styles

- Client/Server
- **Data Centric**
- Peer-to-Peer
- Pipe and Filter
- Model/View/Controller
- Publish/Subscribe

Data Centric vs Client/Server

- In the data centric architecture, both clients and servers modify the same database
  - Firebase apps use a data centric design
- This is different than a client/server model
  - In client/server, clients go through the server, and the server updates the database



Data Centric Benefits

- Lower latency than client/server
- Easy to scale number of servers (major benefit)
- Database handles authentication/security

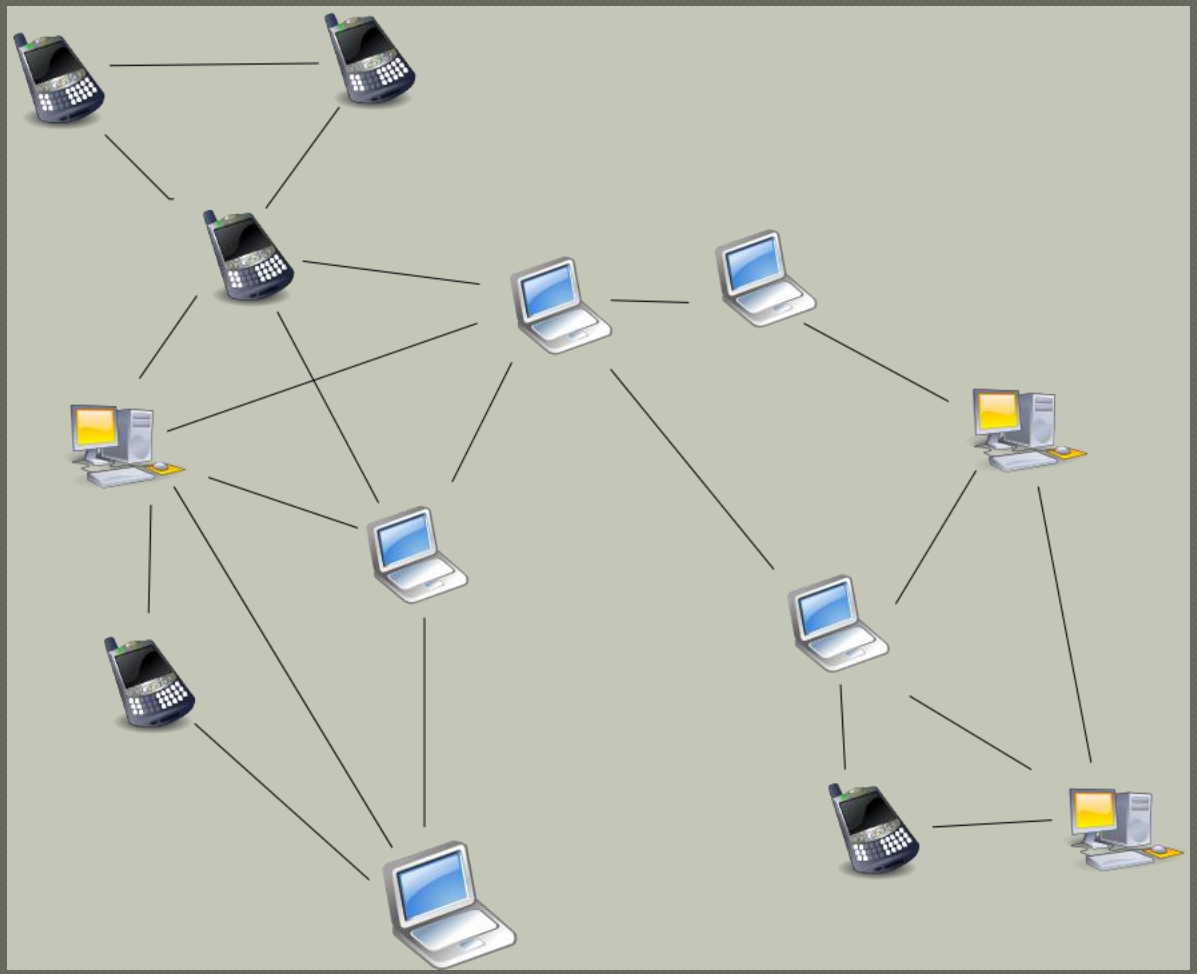
Data Centric Drawbacks

- Some operations are too complicated to allow clients to manage
- Breaks encapsulation
  - adds coupling between client and data
- Uglier communication interface
  - Servers pull requests from database instead of API
- Single point of failure
  - Attractive target for DDoS attacks

Architectural Styles

- Client/Server
- Data Centric
- **Peer-to-Peer**
- Pipe and Filter
- Model/View/Controller
- Publish/Subscribe

Peer to Peer Architecture



Peer to Peer Concept

- No central server, peers broadcast directly to each other
- Each client maintains private state
- Needs to manage:
  - Synchronization
  - Dropout
  - Security
  - Flooding
- Quite complicated

Peer to Peer Benefits

- Robust: no single point of failure
- Don't require central infrastructure
- Useful when communication range is limited (e.g., underwater drones)

Peer to Peer Drawbacks

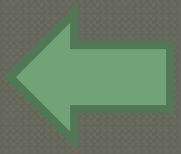
- Synchronization problems
  - What time is it?
- Consensus problems
  - What is the state?
- Byzantine attacks
  - A node becomes infected and provides wrong information to other peers, creating an unstable network

Peer to Peer Examples

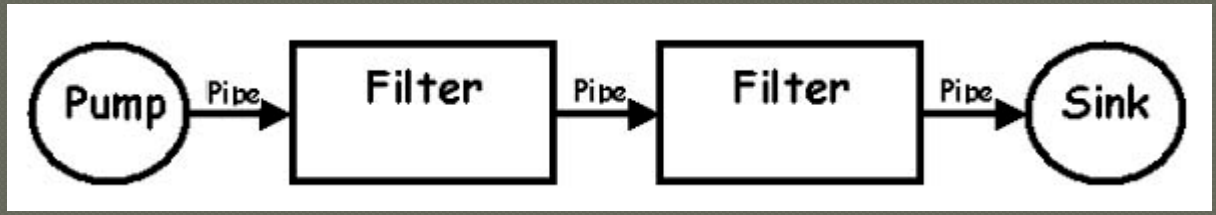
- Cryptocurrency
- Mesh networks
- Automotive networks
- Internet of Things (IOT)
- PC to PC local WiFi hotspot

Architectural Styles

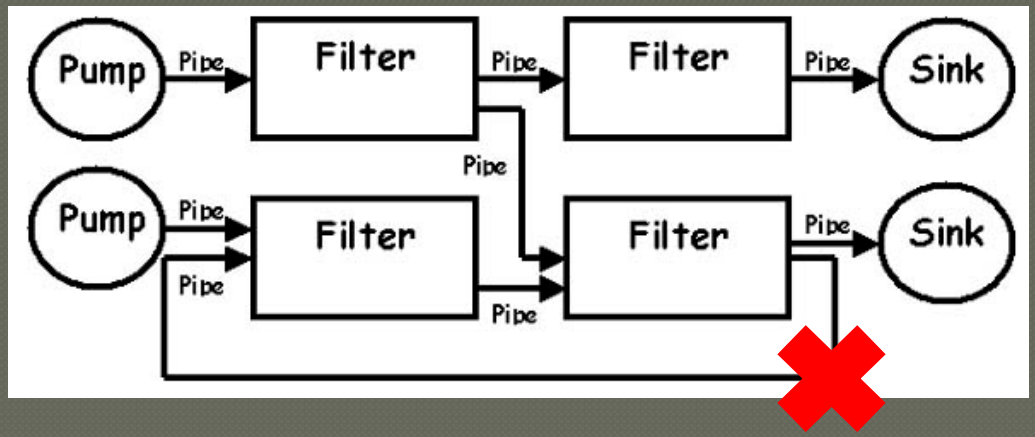
- Client/Server
- Data Centric
- Peer-to-Peer
- **Pipe and Filter**
- Model/View/Controller
- Publish/Subscribe



Pipe & Filter Architecture



- Data flows in one direction.
- Each component "transforms" data
- No loops, 2-way communication, feedback

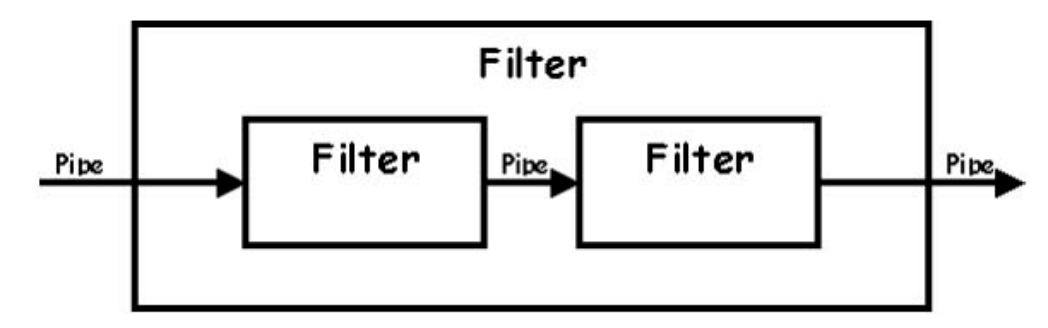


- Most important: no state (massive benefit)

http://www.dossier-andreas.net/software_architecture/pipe_and_filter.html

Pipe & Filter Benefits

- Easy to understand
- Easy to break down problem



- Inherit Parallelization of Processing
- Highly flexible
- Very few bugs caused by state
- If you can use P&F, you probably should

http://www.dossier-andreas.net/software_architecture/pipe_and_filter.html

Pipe & Filter Applications

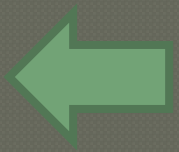
- Natural Language Processing
- Fundamental to the Unix philosophy:
  - Small programs that do one thing
  - Output and input are text
  - Piping programs together
    - `findmnt | grep … | nano`
    - `dmesg | sort | less`
- Particularly suited to functional programming ("components" are actually functions)
- Extremely common in mathematics and engineering

Pipe & Filter Drawbacks

- Implementation necessarily linear
  - Can't skip a step
  - Can't repeat a step

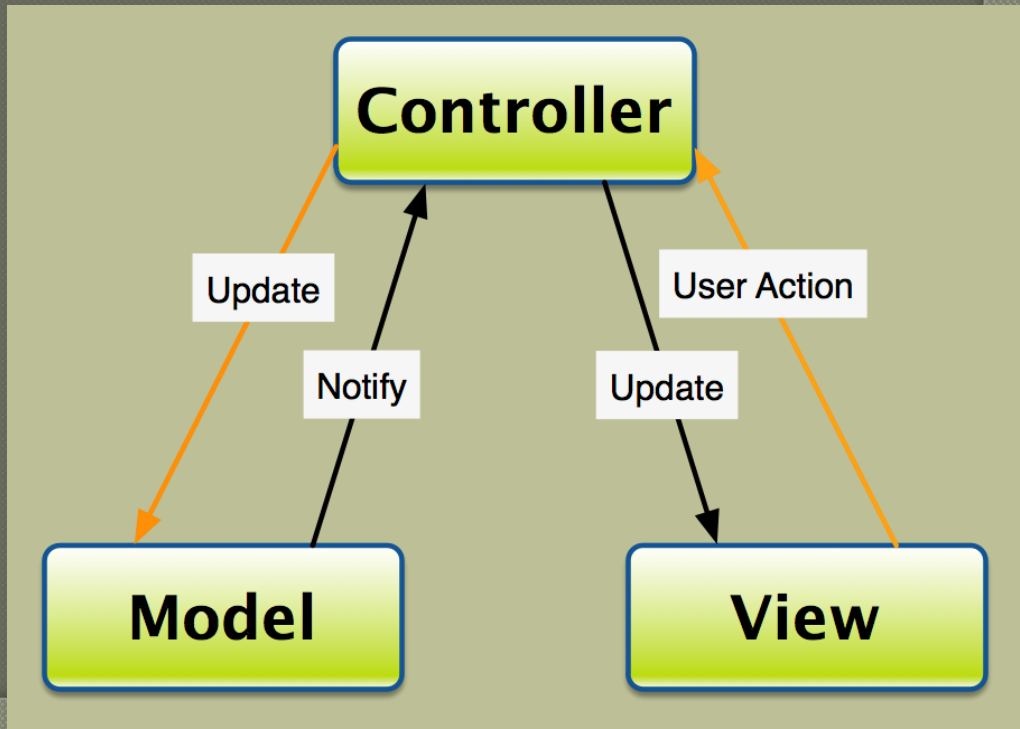
Architectural Styles

- Client/Server
- Data Centric
- Peer-to-Peer
- Pipe and Filter
- **Model/View/Controller**
- Publish/Subscribe



Model/View/Controller (MVC) Concepts

- The **model** represents the data
- The **view** is what the user sees: input/output
- The **controller** mediates between the model and the view and executes business logic (rules).



MVC Example

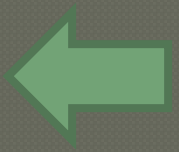
- Timesheet management:
  - The timesheet and employees are stored in the database (models)
  - The interface allows interaction with these objects (views)
  - The business logic makes sure changes are allowed (controller)

MVC Examples

- Extremely common for web apps with conventional business logic (no need for custom server)
- Every GUI app in the world
- Ruby-on-rails apps
- Video games in general (game state, game logic, interface)

Architectural Styles

- Client/Server
- Data Centric
- Peer-to-Peer
- Pipe and Filter
- Model/View/Controller
- **Publish/Subscribe**



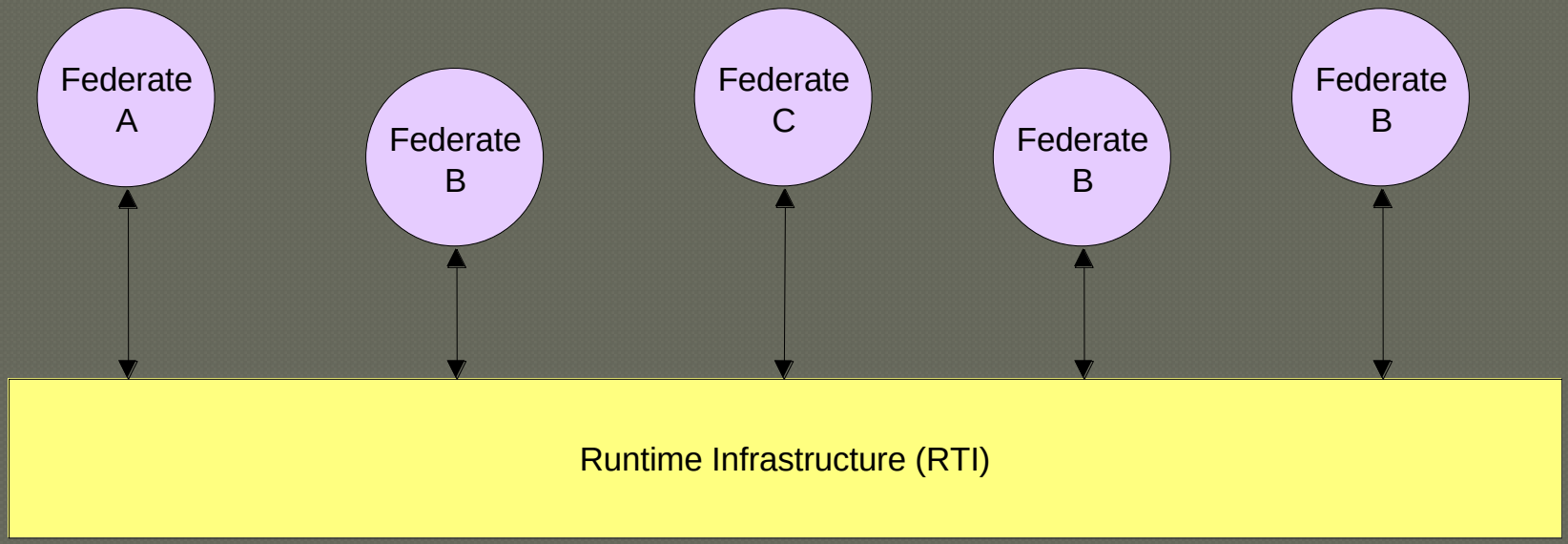
Publish/Subscribe Architecture

- Components do not communicate directly
- Publishers provide information to be used by any subscriber in the system
- A communication broker manages getting most appropriate information from a publisher to match the needs of a subscriber

High Level Architecture (HLA)

- The High Level Architecture (HLA) is a general purpose architecture for simulation reuse and interoperability
- HLA has merited the distinction of preferred architecture for simulation interoperability within the DoD

HLA Framework

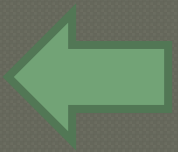


HLA Terms

- Federates
  - simulations, supporting utilities, or interfaces to live systems
- Runtime Infrastructure
  - a special-purpose distributed operating system
- Federation
  - sets of federates working together to support distributed applications
- Object Model Template (OMT)
  - Federates and federations are required to have an object model describing the entities represented in the simulations and the data to be exchanged across the federation
  - Federation Object Model (FOM)
  - Simulation Object Model (SOM)

Object Oriented Design

- Source Control++
- The Design Process
- Software System Architecture
- Architectural Styles
- **Object Oriented Design**



The OO Paradigm

- Abstraction
  - Hidden Data
    - Implementation of Abstract Data Type (ADT) is irrelevant
    - *** **Class members are not (NEVER) accessed directly** ***
      - No public class members
- Encapsulation
  - Data and methods on that data are bundled together
    - A class defines the data implementation, access to the data elements, and methods that act on the data
- Inheritance
  - A class can take on the properties of another class
    - Creates the is-a relationship between the base class and the superclass
- Polymorphism
  - Derived objects (those of a class inherited from another) can behave differently
    - Interface of inherited methods remain the same, but may function differently

The Five Principles of Class Design

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

- Other OO Design Principles
  - YAGNI
  - Once & Only Once

http://wiki.c2.com/?PrinciplesOfObjectOrientedDesign

Single Responsibility Principle

- Each responsibility should be a separate class, because each responsibility is an axis of change
- A class should have one, and only one, reason to change

- If a change to the business rules causes a class to change, then a change to the database schema, GUI, report format, or any other segment of the system should not force that class to change

http://wiki.c2.com/?SingleResponsibilityPrinciple

Open/Closed Principle

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification
- Classes should be designed as if they will persist forever
- The motivation is to prevent the introduction of bugs
- Does this reduce flexibility?

http://wiki.c2.com/?OpenClosedPrinciple

The Liskov Substitution Principle

- “If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T." – Barbara Liskov, Data Abstraction and Hierarchy, SIGPLAN Notices, 23,5 (May, 1988).
- If you substitute an instance of the derived class in a call to a base class method, nothing should break that always works for the base class, as specified by the base class interface

http://wiki.c2.com/?LiskovSubstitutionPrinciple

The Interface Segregation Principle

- The dependency of one class to another one should depend on the smallest possible interface (http://www.objectmentor.com/resources/articles/isp.pdf)

http://wiki.c2.com/?InterfaceSegregationPrinciple

Dependence Inversion Principle

- A. High level modules should not depend upon low level modules. Both should depend upon abstractions.

- B. Abstractions should not depend upon details. Details should depend upon abstractions. (http://www.objectmentor.com/resources/articles/dip.pdf)

http://wiki.c2.com/?DependencyInversionPrinciple

The YAGNI Principle

- "You aren't gonna need it"
- Avoid developing unless you have to:
  - The cheapest code is the code you don't write
- If it's in the requirements you probably do need it

http://c2.com/xp/YouArentGonnaNeedIt.html

Once and Only Once

- We never want to duplicate code
- What if there's an error in the code?
  - Now you have to change it everywhere
  - There is no way to ensure that code is in sync
- **Change** is the only reason why we're doing any of this

Best Practices

- Separate what changes from what stays the same
  - If something stays the same, you won't break it
  - Keeping change limited reduces the amount of analysis
- Coupling vs Cohesion
  - Coupling is bad, cohesion is good.
  - Classes should work together in small, cohesive clusters
  - **You should have high cohesion within modules and low coupling between modules**
- Program to an interface, not an implementation
  - Depend on an interface where practical (instead of class)
  - Allows classes to be swapped out later
  - Even more pragmatic with duck-typing (next slide)

A Word on Duck Typing

- Comes from saying: "If it walks like a duck and quacks like a duck, it's a duck"
- Means that any object can be substituted for any other if it has the correct methods
- Class doesn't matter; don't need to know class in advance.
- Common in dynamic OO languages:
  - Javascript, Lua, Ruby, Python, etc.
- Doesn't exist in static languages, but structural typing similar
  - Go, O'Caml, Scala, etc.

Why is Duck Typing Relevant?

- Duck-typed languages often don't have "interfaces" or "abstract classes"
- Instead, functions take an untyped parameter and simply execute methods on it
- Semantic ambiguities are not discovered until runtime

- Recommendations:
  - Make sure to test for runtime errors
  - Draw base classes and interfaces in UML anyway to capture intended relationships