ENPM 613: SOFTWARE DESIGN AND IMPLEMENTATION

GENERATE ARCHITECTURE: METHODS, STYLES, TACTICS

Dr. Tony D. Barber / Prof. Sahana Kini

Fall, 2022

TODAY'S CLASS OBJECTIVES AND OUTCOME

- Explain approaches for generating architecture
- Define architecture styles and tactics
- List features, advantages, and disadvantages of architecture styles
- Recommend styles appropriate for specific situation
- List tactics associated with quality attributes
- Explain designing with and for reuse
- Identify architecture anti-patterns

OUTLINE

- Lecture
  - Architecture design concerns
  - Architecture generation approaches and strategies
  - Design patterns / architecture styles
  - Quality driven design mechanisms/tactics
    - Application software security design
  - Reuse: components, frameworks
  - Anti-patterns
- Class exercises: styles and tactics
- Assignments

RESOURCES

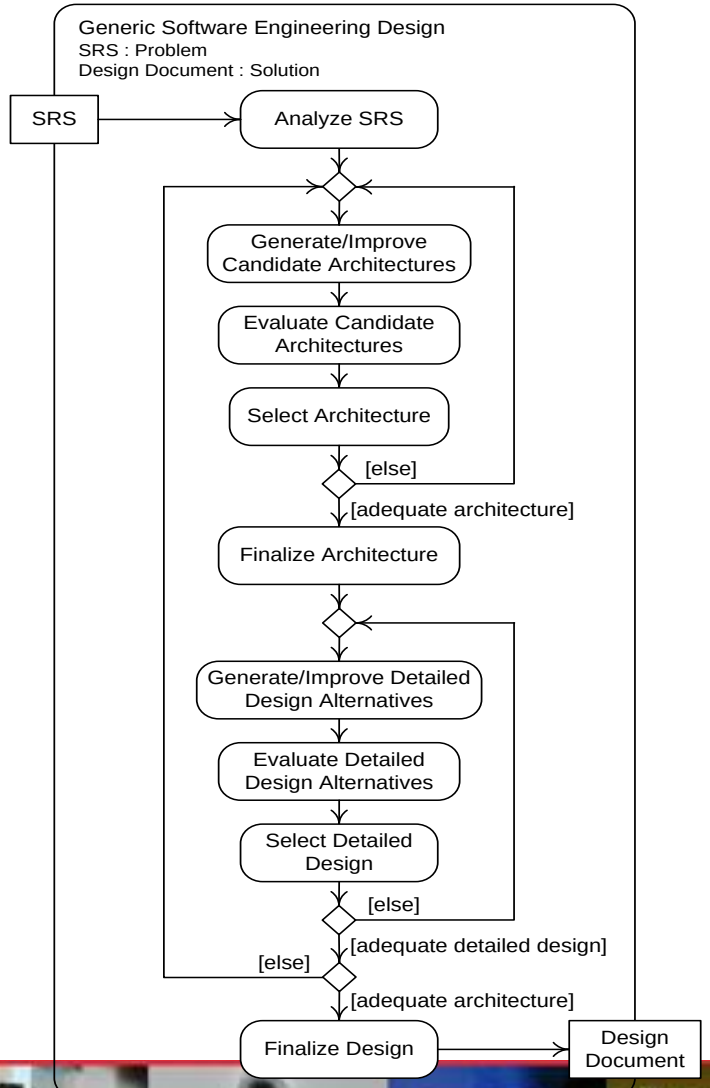
- **Books**
  - Textbook Chapters 9, 10, 11, 15
  - *Software Engineering*, Ian Sommerville
  - *Software Design Methodology - From principles to Architectural Styles*, by Hong Zhu
  - *Software Design,* by David Budgen
  - *Software Architecture in Practice, (SEI Series in Software Engineering),* by Len Bass and Paul Clements
  - *Software Modeling and Design - UML, Use Cases, Patterns, and Software Architectures*, By Hassan Gomaa
  - *Architecting Software Intensive Systems*, by Anthony Lattanze
- **Papers**:
  - *An Introduction to Software Architecture*, David Garlan, Mary Shaw
  - *Architectural Blueprints—The "4+1" View Model of Software Architecture*, by Philippe Kruchten, IEEE Software 12 (6) November 1995
- SEI web site: http://www.sei.cmu.edu/architecture/
- Microsoft Software Architecture and Design guidance
- Video: Making Architecture Matter - Martin Fowler Keynote

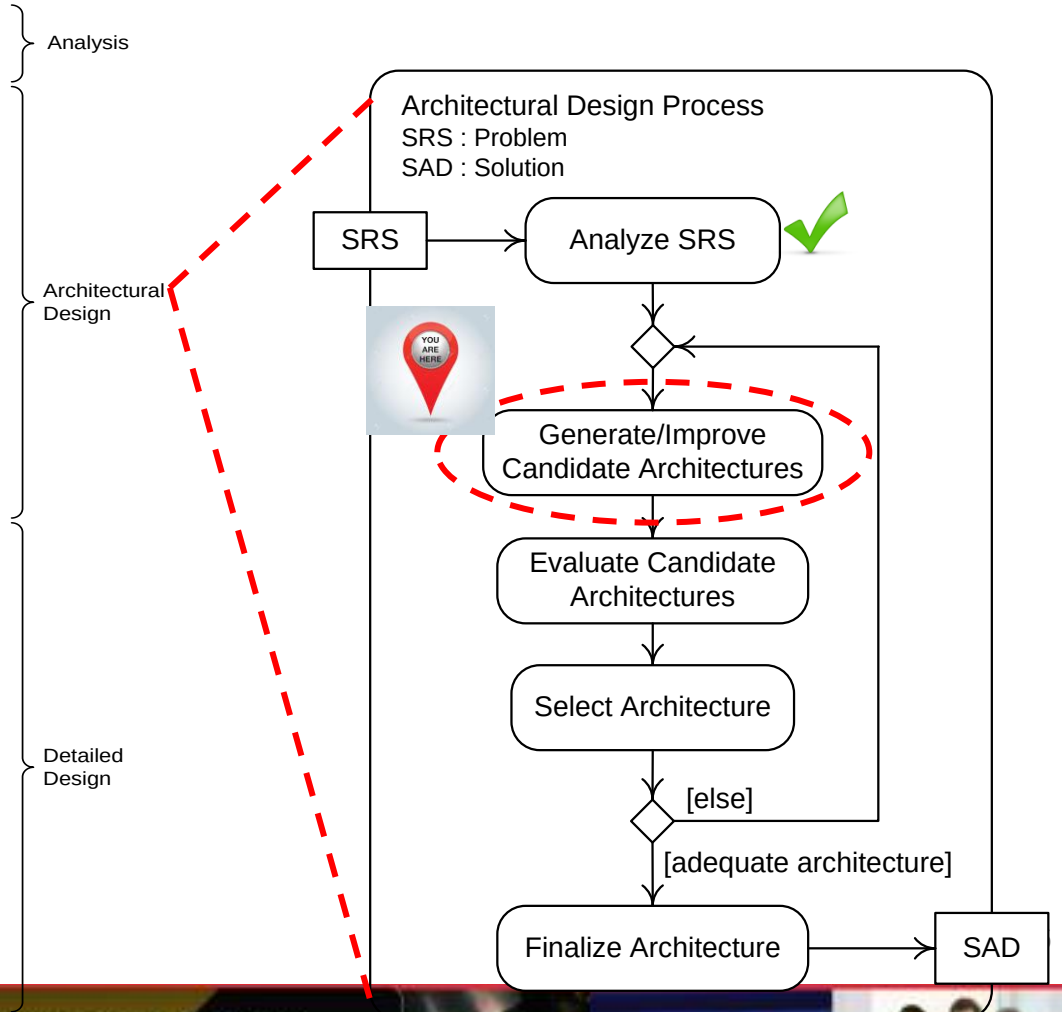
OUTLINE

We are here

- Lecture
  - Architecture design concerns
  - Architecture generation approaches and strategies
  - Design patterns / architecture styles
  - Quality driven design mechanisms/tactics
    - Application software security design
  - Reuse: components, frameworks
  - Anti-patterns
- Class exercises: styles and tactics
- Assignments

SOFTWARE DESIGN PROCESS





SOFTWARE ARCHITECTURE CONCERNS

- Decomposition of system functionality
- Allocation of functions to components
- Component interfaces
- Communication and interaction among components
- Component and system properties: qualities, constraints
- Reuse of common design styles

OUTLINE

- Lecture
  - Architecture design concerns
  - Architecture generation approaches and strategies
  - Design patterns / architecture styles
  - Quality driven design mechanisms/tactics
    - Application software security design
  - Reuse: components, frameworks
  - Anti-patterns
- Class exercises: styles and tactics
- Assignments

We are here

DESIGN APPROACHES/STRATEGIES

- How do we design (generate) the architecture?

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

ARCHITECTURE GENERATION TECHNIQUES

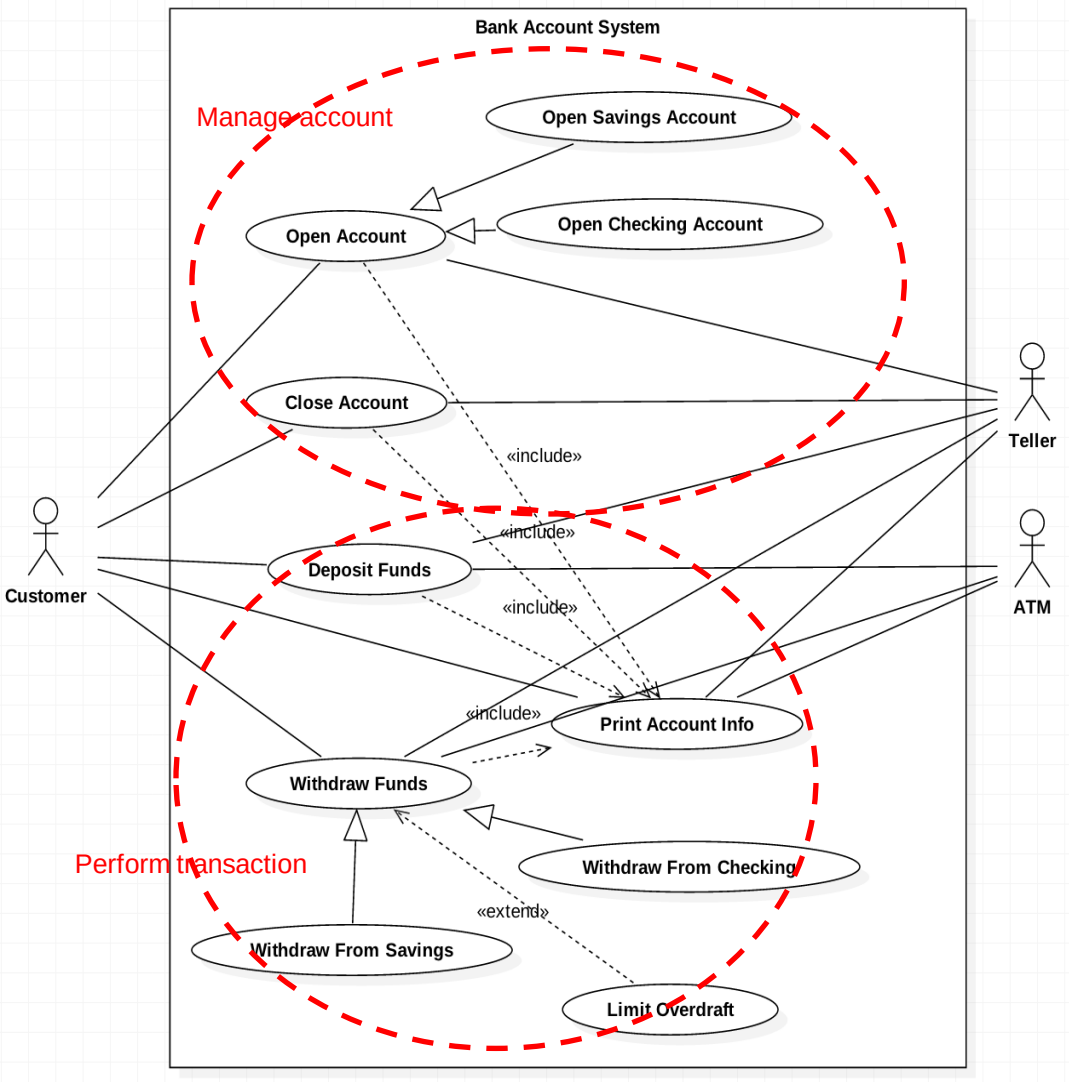
- Determine *functional* components
  - Create components responsible for realizing coherent collections of *functional* and *data requirements*
- Determine components based on *quality attributes*
  - Form components to *meet non-functional requirements*
  - Then add components to fill functional and data requirements gaps
- Modify an existing architecture
  - Alter an architecture for a similar program
- Elaborate an Architectural Style
  - An **architectural style** is a paradigm of program or system constituent types and their interactions (more on this later)
  - Elaborate a style to form an architecture
- Transform a Conceptual Model
  - Modify a conceptual model from a problem to a solution description

SOFTWARE DECOMPOSITION

- Purpose: to handle complexity – "Divide and conquer"
- Decomposition criteria:
  - Functional
    - Partitions functions or requirements into modules
  - Object-oriented
    - Assigns objects to architecture elements
  - Data-oriented
    - Focuses on how data will be partitioned into modules
  - Process-oriented
    - Partitions the system into concurrent processes
  - Feature-oriented
    - High-level design describes the system in terms of a service and a collection of features
    - Lower-level designs describe how each feature augments the service and identifies interactions among features
  - Event-oriented
    - Focuses on the events the system must handle and assigns responsibility for events to different modules

EXAMPLE OF FUNCTIONAL DECOMPOSITION – BANK SYSTEM

Description (model) of the needs/problem

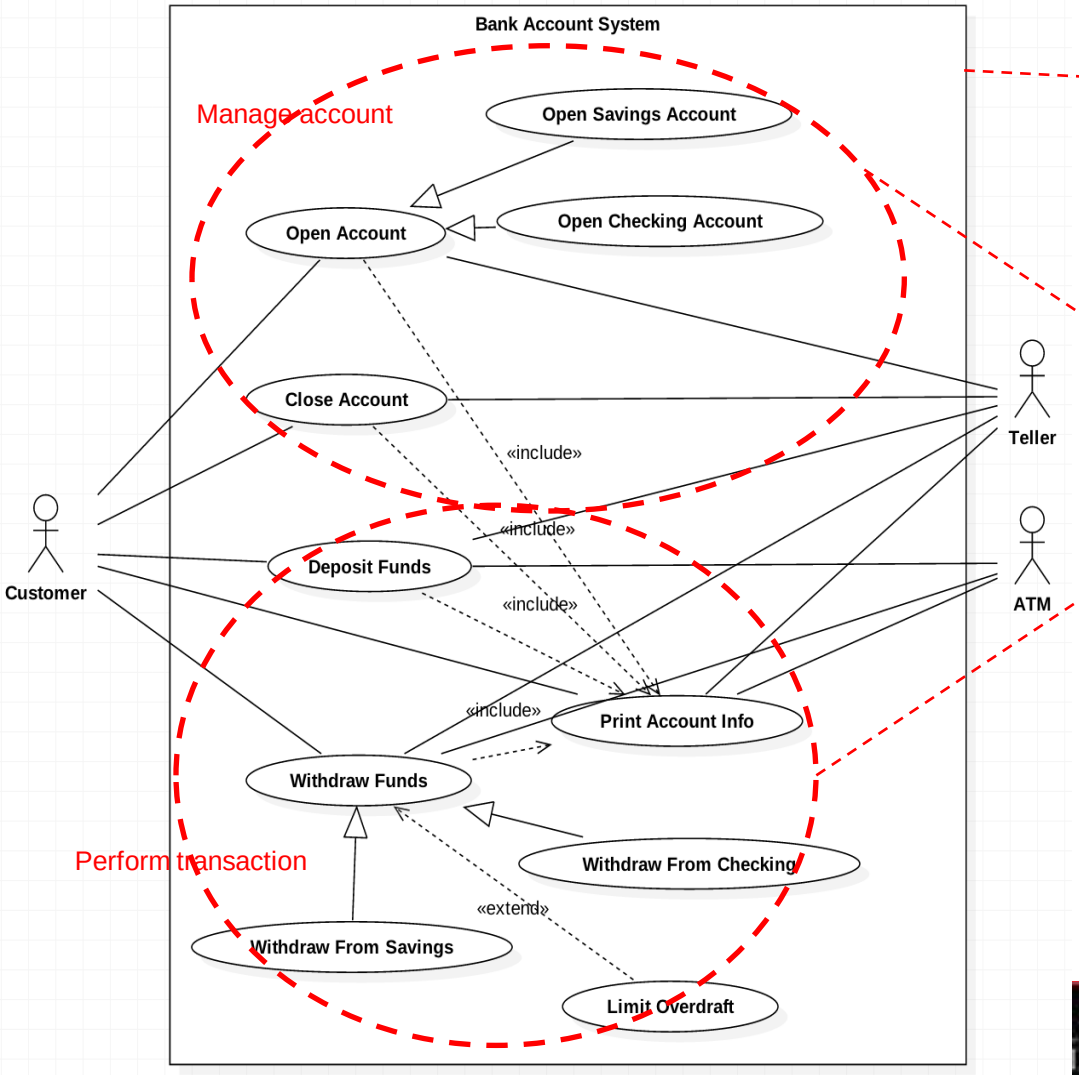


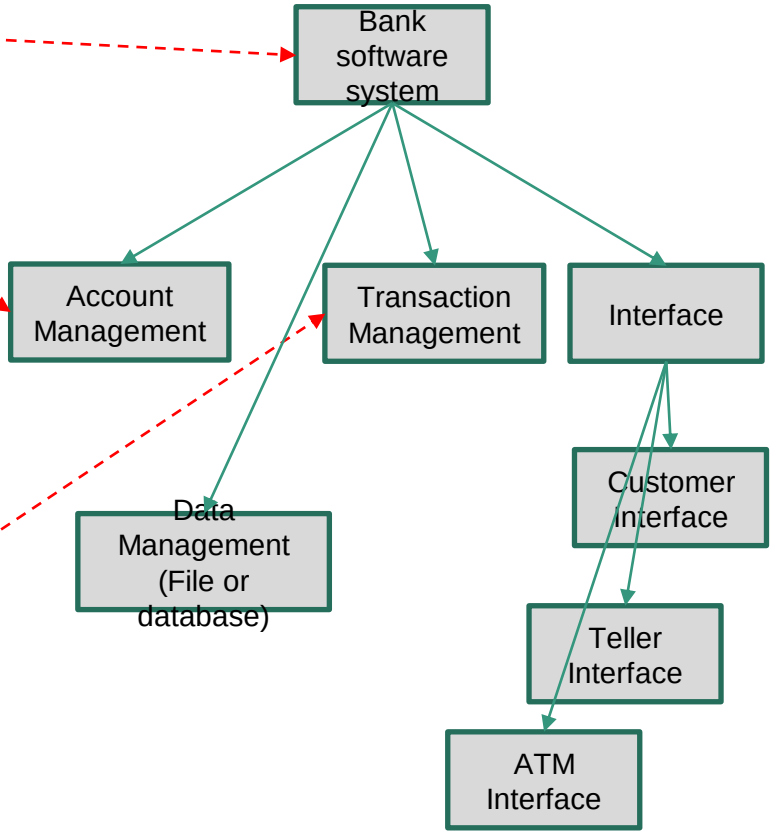
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

EXAMPLE OF FUNCTIONAL DECOMPOSITION – BANK SYSTEM

Description (model) of the needs/problem

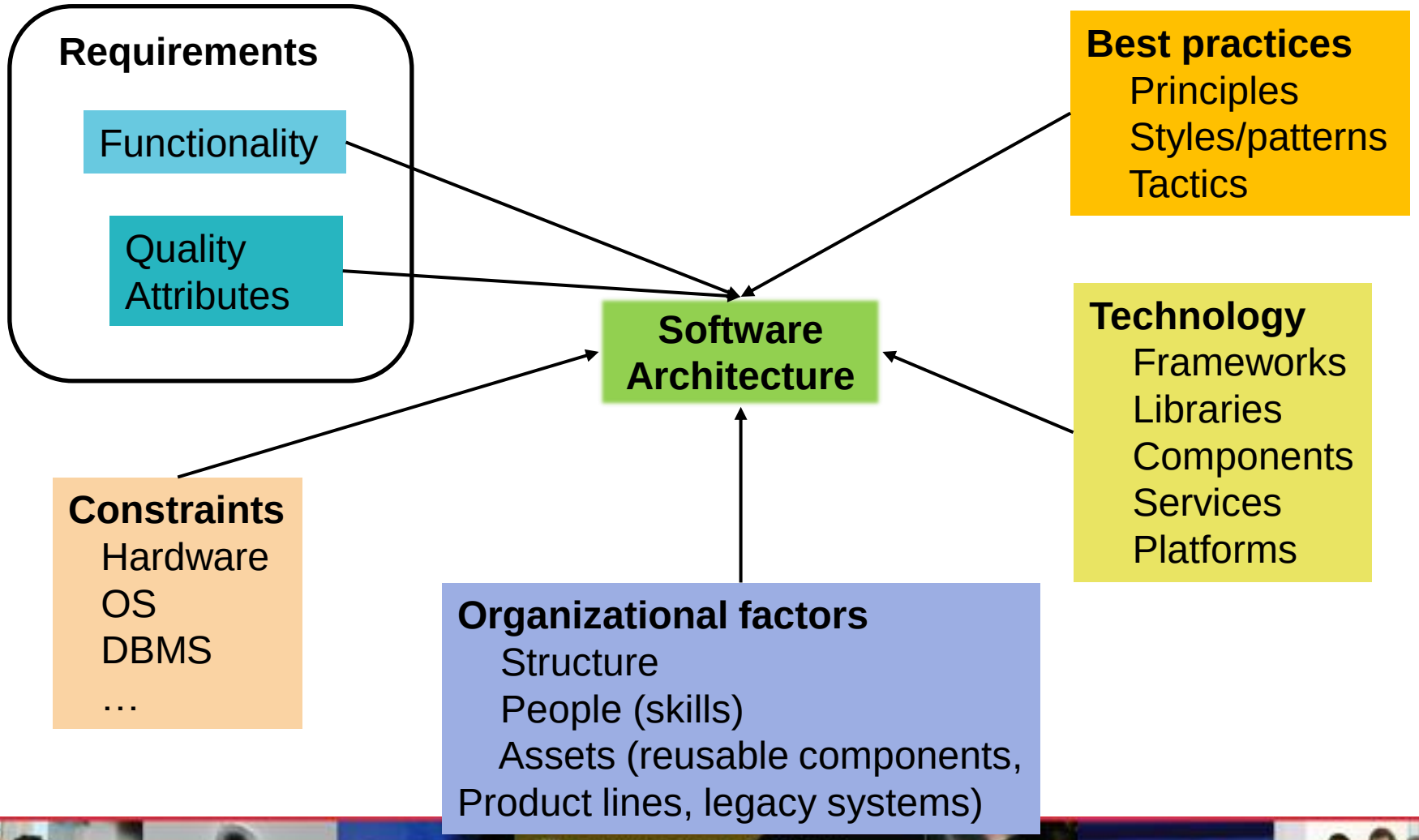
Description (model) of the solution architecture





Legend:
System Decomposition (design)
Functionality mapping

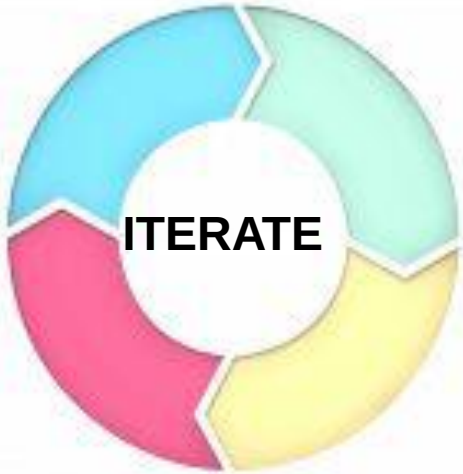
SOFTWARE ARCHITECTURE *DRIVERS*



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

EXAMPLE ARCHITECTURE GENERATION STEPS

1. *Identify* the design priorities (driving factors)
2. *Review* current system (if exists) and/or reusable assets
3. *Decompose* the system - *identify* initial components (use decomposition criteria on previous slide)
4. *Select* system styles and tactics/mechanisms
5. *Select* data persistence (e.g., DBs, files)
6. *Define* an architecture
   a) *Define* subsystems/components
   b) *Define* subsystems/components interfaces
7. Iterate 2-6
8. Iterate again (2-3 times)



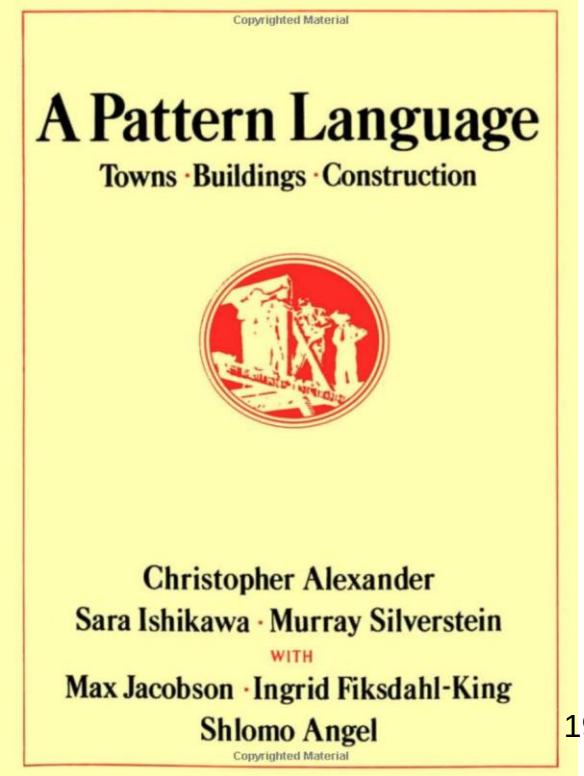
OUTLINE

- Lecture
  - Architecture design concerns
  - Architecture generation approaches and strategies
  - Design patterns / architecture styles
  - Quality driven design mechanisms/tactics
    - Application software security design
  - Reuse: components, frameworks
  - Anti-patterns
- Class exercises: styles and tactics
- Assignments

We are here

PATTERNS

"Patterns describe a problem and then offer a solution."

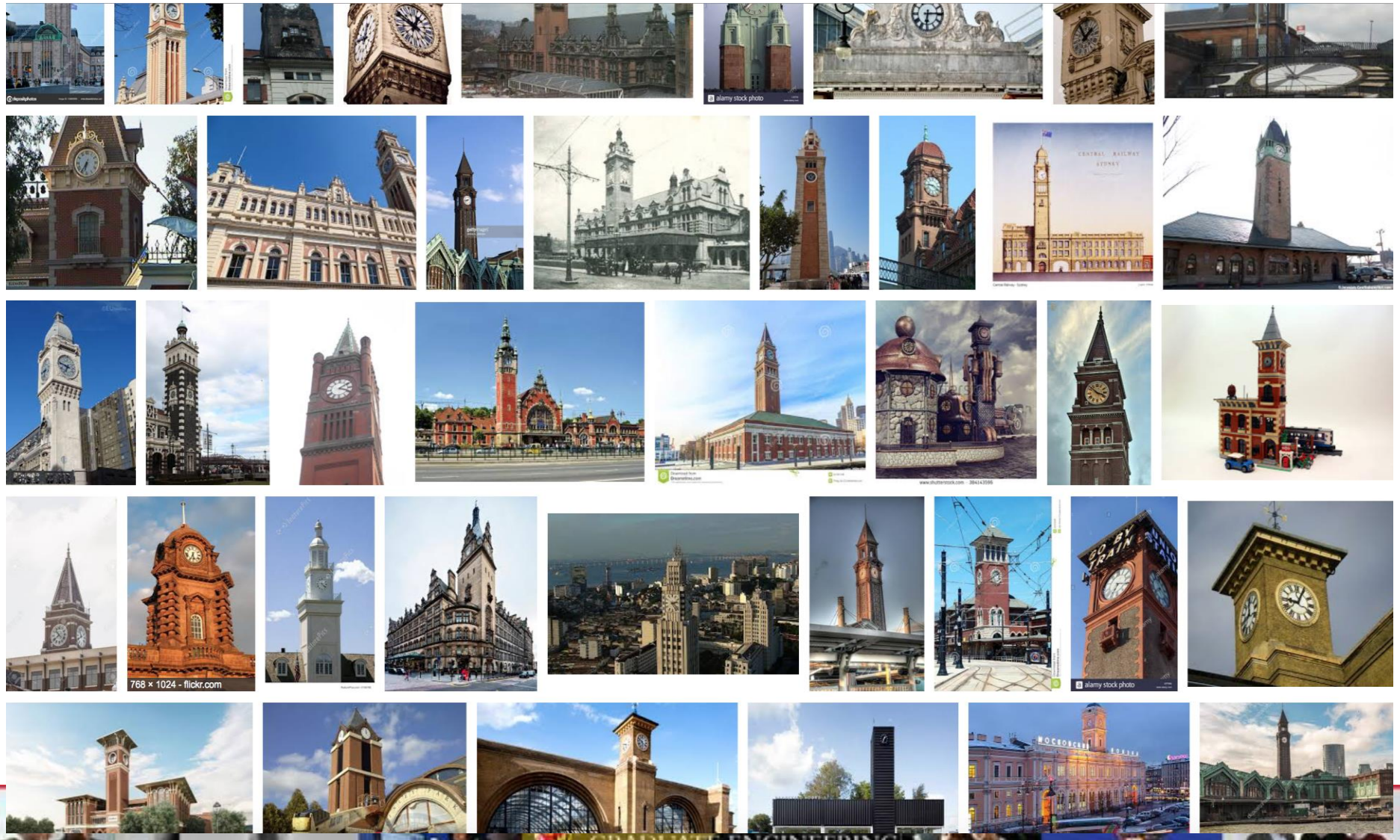


1977

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

PATTERNS IN BUILDINGS' ARCHITECTURE – BEACH HOTELS

PATTERNS IN BUILDINGS' ARCHITECTURE – TRAIN STATIONS



PATTERNS IN BUILDINGS ARCHITECTURE – CASTLES

SOFTWARE DESIGN PATTERNS

- A software **design pattern** is
  - A general, repeatable/reusable (“canned”) solution to a commonly occurring problem (need) in software design
  - A model proposed for imitation in solving a software design problem

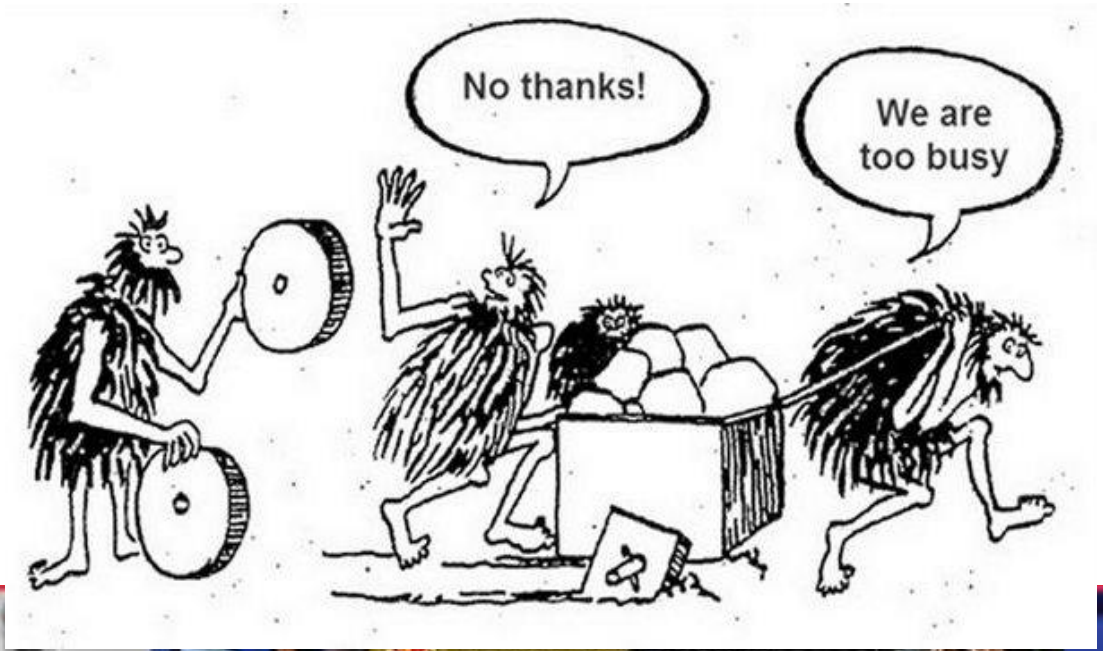
- The patterns movement began in 80s and grew in the 90s
- Patterns are now a well-established part of software design

WHY DESIGN PATTERNS?

- Reuse
- Capture expertise, transfer knowledge
  - Expert designers behave differently from novices
    - What do experts know that novices do not?
    - Experts have successful design patterns from **past experience** that they **apply to new problems**.

WHY DESIGN PATTERNS?

- Reuse
- Capture expertise, transfer knowledge
  - Expert designers behave differently from novices
    - What do experts know that novices do not?
    - Experts have successful design patterns from **past experience** that they **apply to new problems**.



GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

ADVANTAGES OF USING PATTERNS

- ***Support reuse***
  - Patterns and their implementations can be reused extensively
- *Promote communication*
  - Pattern names, as well as knowledge regarding advantages and disadvantages speeds communication
- *Streamline documentation*
  - Pattern form and behavior need not be elaborated
- *Increase efficiency*
  - Tool/technology support for patterns makes development faster
    - E.g. Java classes, libraries, and frameworks
- *Provide ideas*
  - Patterns can be the starting point for design or a basis for improvements

DESIGN PATTERNS GRANULARITY

Granularity

- *Architectural styles (or patterns)* are for entire systems and sub-systems
  - Kinds of architectural components and interactions
  - E.g., Layered, Model-View-Controller, Pipe and Filter
- *Design patterns* involve several interacting functions or classes
  - Collaboration between detailed design components
  - E.g., Iterator, Façade, Factory, Strategy
- *Data structures and algorithms* are low-level patterns
  - Store, retrieve, manipulate data
  - E.g., lists, hash tables, trees; binary search, tree traversal, quicksort
- *Idioms* are ways of doing things in particular programming languages

ARCHITECTURE STYLES IN BUILDINGS (E.G., IN HOMES)

- Why are there different architectural styles?
- Name some architecture styles and their characteristics
- What users' ne[illegible]'e or restrict?

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

ARCHITECTURE STYLES

- An architectural style is a paradigm of program or system *constituent types* and their *interactions*
- An abstraction for a set of architectures that have a set of common architectural features
- "An architectural style determines the *vocabulary* of *components* and *connectors* that can be used in instances of that style, together with a set of *constraints* on how they can be *combined*." (Garlan and Shaw)
  - Constraints
    - Topological on architectural descriptions (e.g., no cycles)
    - Execution semantics

ARCHITECTURE STYLES (CONTINUED)

- Architectural style elements:
  - A set of *component types*
  - A set of *connectors*
  - A *topological structure*
  - A set of *semantic constraints*

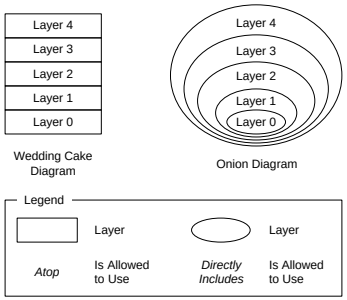
ARCHITECTURE STYLES

- **Structure**
  - Layered
  - Pipes and filters
  - Component-based
  - Monolithic application
  - Model view controller
- **Shared memory**
  - Data-centric
  - Blackboard
  - Rule-based
- **Messaging**
  - Publish-subscribe
  - Event-driven (aka implicit invocation)
  - Asynchronous messaging
- **Adaptive systems**
  - Plug-ins
  - Microkernel
  - Reflection
  - Domain specific languages
- **Distributed systems**
  - Client-server
  - Shared nothing architecture
  - Space-based architecture
  - Object request broker
  - Peer-to-peer
  - Representational state transfer (REST)
  - Service-oriented
  - Cloud computing patterns
- **Services**
  - Monolith
  - Microservices

LAYERED STYLE CHARACTERISTICS AND RULES

- Is a ***structural*** style
- The software is organized into *layers* that each provide a *cohesive set of services* with a *well-defined interface*.
- Each layer is allowed to use:
  - Only the layer directly below it (Strict Layered style) or
  - Multiple (all) layers below it (Relaxed Layered style).
- Typically, components of the lowest-level layer interact with the underlying OS or Hardware

GENERIC LAYER STYLE REPRESENTATIONS



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

LAYERED STYLE DESIGN

- Criteria for forming layers:
  - Levels of abstraction
    - Example: Network communication layers
  - Information hiding, decoupling, etc.
    - Examples: User interface layers, virtual device layers
  - Virtual machines
    - Examples: Operating systems, interpreters

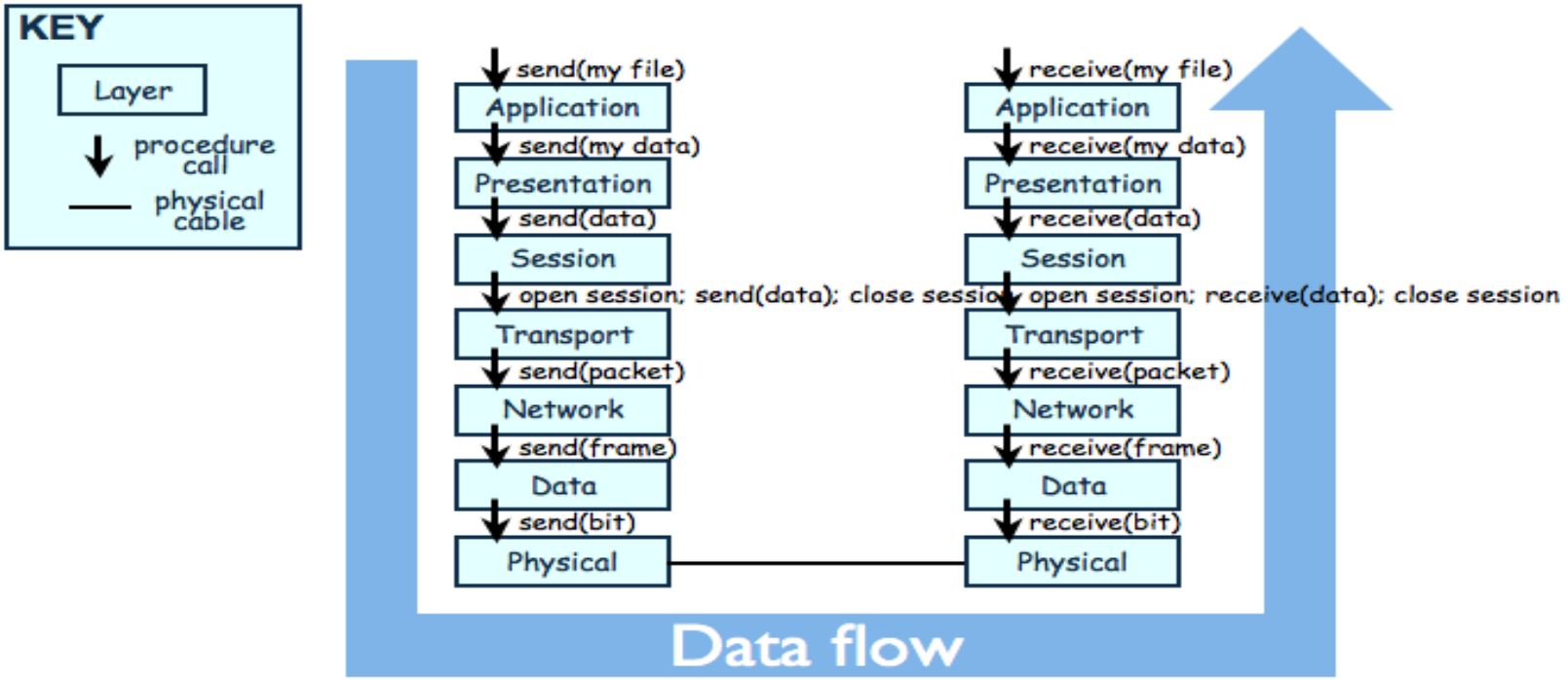
LAYERED STYLE ADVANTAGES

- Layers are highly *cohesive* and promote *information hiding*
- Layers are not strongly coupled to layers above them, *reducing overall coupling*
- Layers help decompose programs, *reducing complexity*
- Layers are *easy to alter* or fix by replacing entire layers, and *easy to enhance* by adding functionality to a layer
- Layers are usually *easy to reuse*
- Enables incremental testing from the bottom to top layer
- Plug-out and Plug-in of a layer with a new layer conforming to the same API
  - Helpful in producing variants of a product
- Helps to distribute the work to different teams
  - Often different teams work on different layers
- Supports *reuse, modifiability, portability*

LAYERED STYLE DISADVANTAGES

- Passing everything through many layers can complicate systems and negatively affect performance
- Debugging through multiple layers can be difficult
- Getting the layers right can be difficult
  - What to abstract and what-not-to abstract requires experience and business knowledge
- Defining API's for each layer is challenging
  - Especially hiding implementation details at the interface level
- Layer constraints may have to be violated to achieve unforeseen functionality

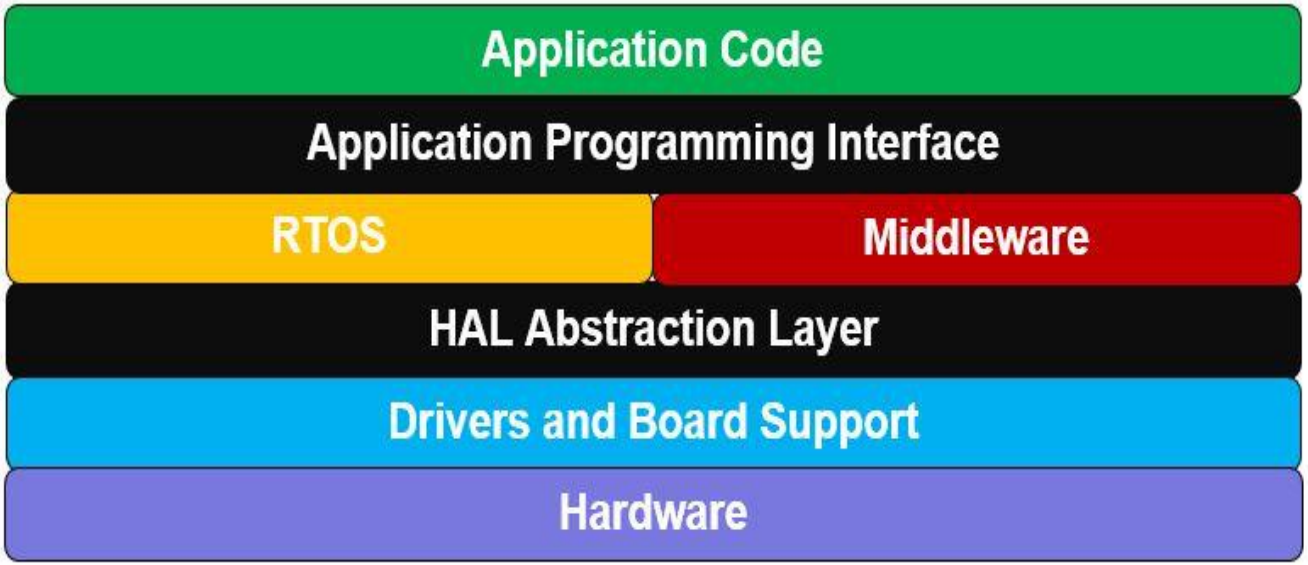
LAYERED STYLE EXAMPLE – THE OSI MODEL



- Open Systems Interconnection model (OSI model) is a conceptual model that characterizes and standardizes the communication functions of a telecommunication or computing system without regard to their underlying internal structure and technology.
- Its goal is the **interoperability** of diverse communication systems with standard protocols.
- The model partitions a communication system into abstraction layers

GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

LAYERED STYLE EXAMPLE - EMBEDDED MICROCONTROLLERS

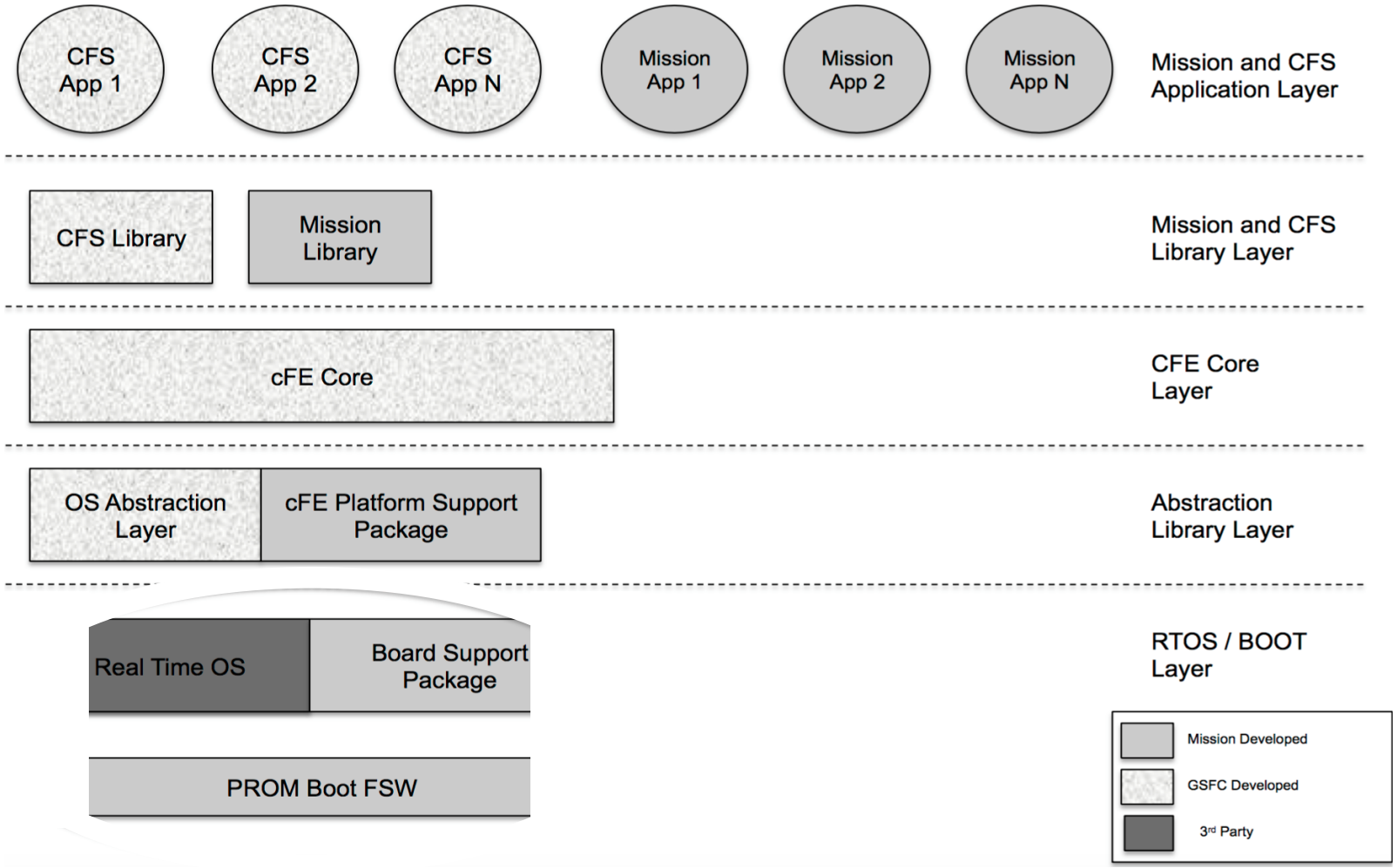


- **Microcontroller** (system on a chip or SoC) contains one or more CPUs (processor cores) along with memory and programmable input/output peripherals
  - Are used in automatically controlled products and devices (automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, toys)
- **HAL** is a hardware abstraction layer that defines a set of routines, protocols and tools for interacting with the hardware
- **API** is an application programming interface that defines a set of routines, protocols and tools for creating an application

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

LAYERED STYLE EXAMPLE – NASA CORE FLIGHT SW

Supports **reusability** and **portability**



[From: *NASA/GSFC's Flight Software Architecture: Core Flight Executive and Core Flight System*, by Alan Cudmore, http://flightsoftware.jhuapl.edu/files/2011/FSW11_Cudmore.pdf]

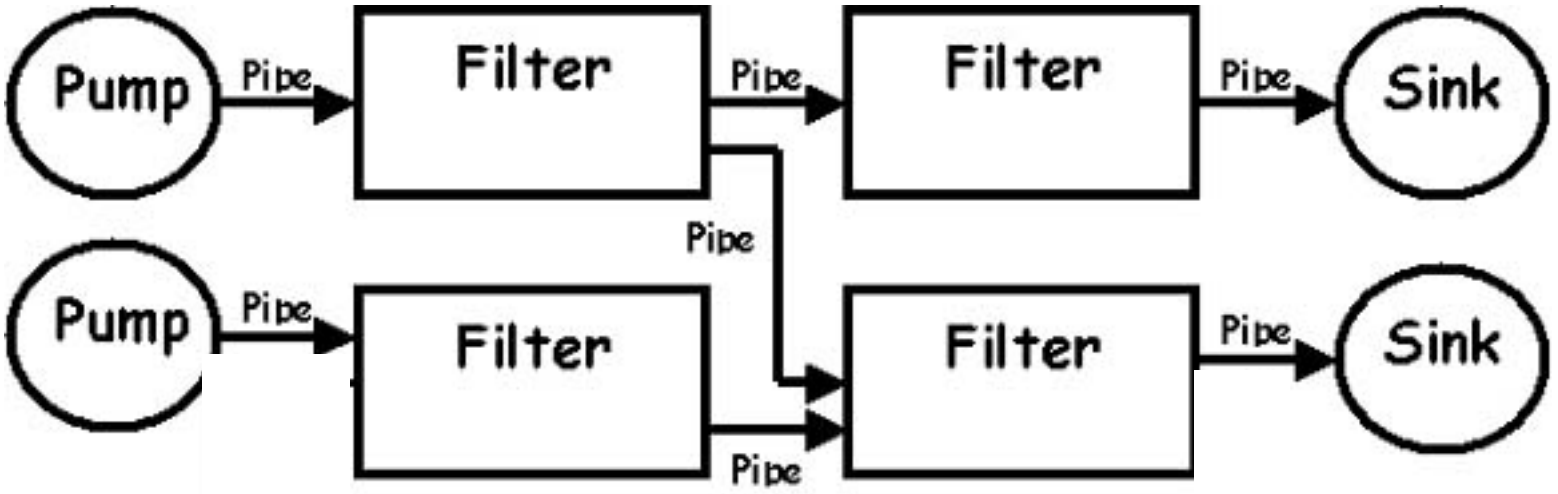
GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

PIPE-AND-FILTER STYLE

- Is a ***structural*** style
- The Pipe-and-Filter style is a dynamic model in which program components are *filters* connected by *pipes*
  - A *filter* is a program *component* that transforms an input stream to an output stream
    - Filters are isolated and communicate only through data streams
  - A *pipe* is conduit for a stream (*connector*)
- Topology: Upward filter writes to a pipe and downward filter reads from the pipe
- It requires pipes to synchronize filters
- Pipe-and-filter topologies should be **acyclic** graphs
- Avoids timing and deadlock issues

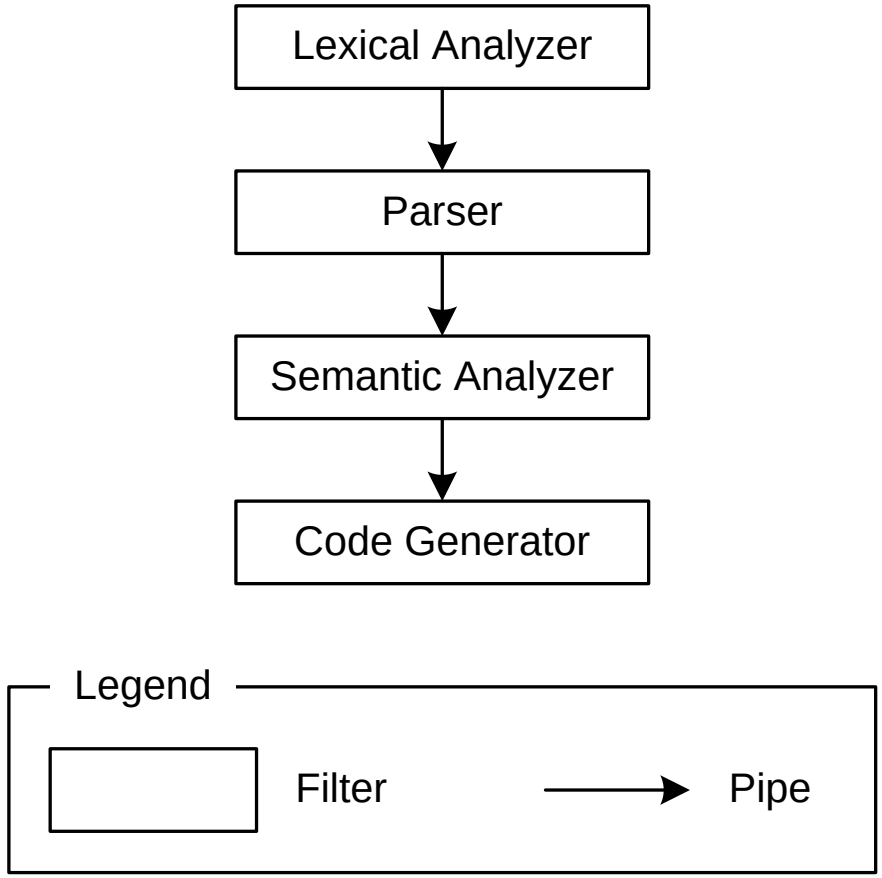
- A simple linear pipe and filter arrangement is a **pipeline**

PIPE AND FILTER GENERIC REPRESENTATION



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

PIPE-AND-FILTER STYLE EXAMPLE - COMPILER



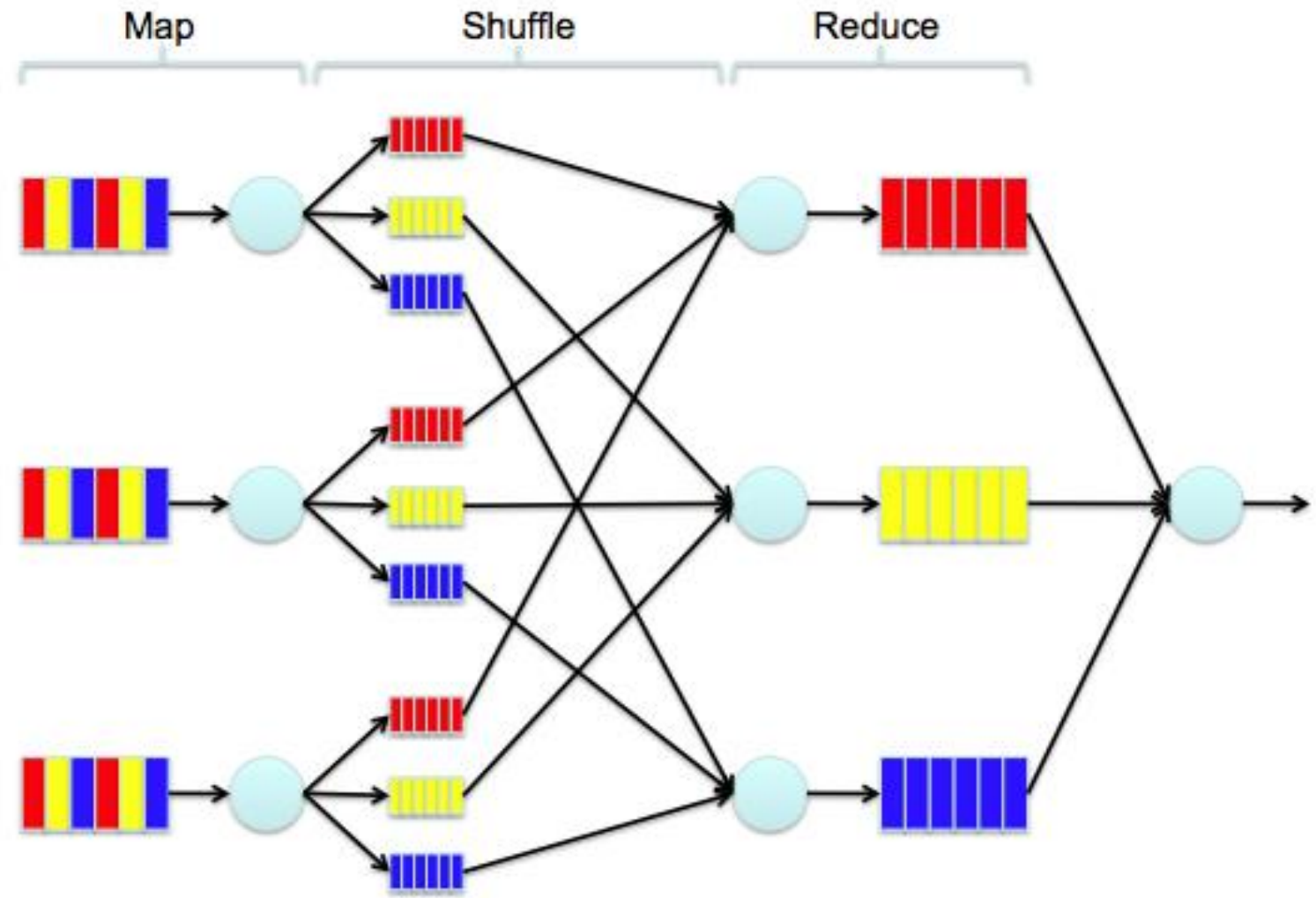
MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

PIPE-AND-FILTER STYLE EXAMPLE - UNIX

- Unix environment (pipelines)

  find . –name “*.c” | xargs cat | wc –l
- The above instructions count the total number of lines in all c files
  - find command list the files with .c extension
  - cat command concatenates all files (using xargs)
  - wc –l counts the number of lines
- Pipes (|) facilitate the co-ordination of filters (i.e., commands)

PIPE-AND-FILTER STYLE EXAMPLE – BIG DATA PROCESSING



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

PIPE-AND-FILTER STYLE ADVANTAGES

- Filters are not directly coupled to each other
- Filters can be **modified** and replaced easily
- Filters can be rearranged with little effort, making it easy to develop similar programs
  - Filters can be composed with different filters to produce new configurations
    - Helps in producing system variants
- Filters are highly **reusable**
- **Concurrency** is supported and is relatively easy to implement
  - Filters can be executed concurrently
- Filters can be tested independently

PIPE-AND-FILTER STYLE DISADVANTAGES

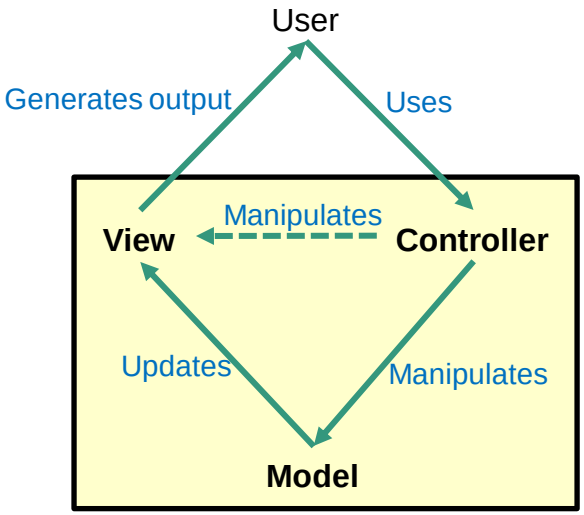
- Filters communicate only through pipes, which makes it difficult to coordinate them
- It is hard to share data between non-adjacent filters
- Filters usually work on simple data streams, which may result in wasted data conversion effort
- Error handling is difficult
  - If one intermediate filter fails then other filters might get wrong-data
  - System could be in trouble if filters disagree on the interaction protocol

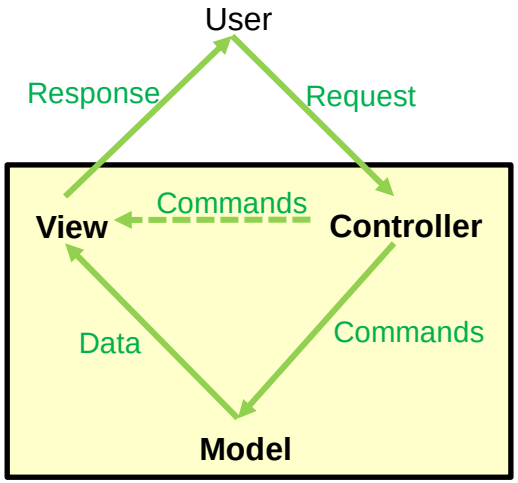
MODEL–VIEW–CONTROLLER (MVC) STYLE

- Another ***structural*** style
- Separates internal representations of information from the ways information is presented to/accepted from the user
  - Separates the *data model* with *business rules* from the *user interface*
- MVC decouples its major components:
  - ***Model***
    - The central component of the pattern
    - Expresses the *application's behavior* in terms of the *problem domain*, independent of the user interface
    - It *directly manages the data, logic and rules of the application*
  - ***View***
    - An output representation of information (e.g., chart, diagram)
    - Multiple views of the same information are possible
  - ***Controller***
    - Accepts input and converts it to commands for the model or view

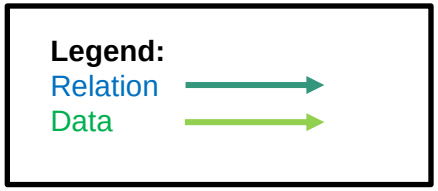
Used for implementing user interfaces, designing web/mobile applications, desktop clients

MODEL–VIEW–CONTROLLER (MVC) STYLE (CONTINUED)





- Interactions between components
  - A ***model*** stores data that is retrieved according to commands from the *controller* and displayed in the view (business logic)
  - A ***view*** generates (renders) new output to the user based on changes in the *model*
  - A ***controller*** can send commands to the *model* to update the model's state (e.g., editing a document) and generate data for the *view*. It can also send *commands* to its associated view to change the view's presentation of the model (e.g., scrolling through a document, movement of document)



MVC STYLE ADVANTAGES

- High cohesion
  - MVC enables logical grouping of related actions on a controller together
  - The views for a specific model are also grouped together
- Low coupling
  - Among models, views or controllers
- Efficient code reuse
- Simultaneous development
  - Multiple developers can work simultaneously on the model, controller and views
- Ease of modification
  - Because of the *separation of responsibilities*, future development or modification is easier
  - Models can have multiple views
- Testability
  - Components participating in MVC need to know ***only*** about the interfaces/contracts of the other components, not their implementation. This allows to unit test the *Controller* with "fake"/prototype *View* and *Model* implementations
- Separating model and view addresses a very important practical problem of how designers work with coders
- Languages like Java, C#, Ruby, PHP have MVC frameworks

MVC STYLE DISADVANTAGES

- Code navigability
  - The framework navigation can be complex
    - It introduces new layers of abstraction
    - Requires users to adapt to the decomposition criteria of MVC
- Multi-artifact consistency
  - Decomposing a feature into three artifacts might cause scattering
    - Requires developers to maintain the consistency of multiple representations at once
- Pronounced learning curve
  - Developers using MVC need to be skilled in multiple technologies

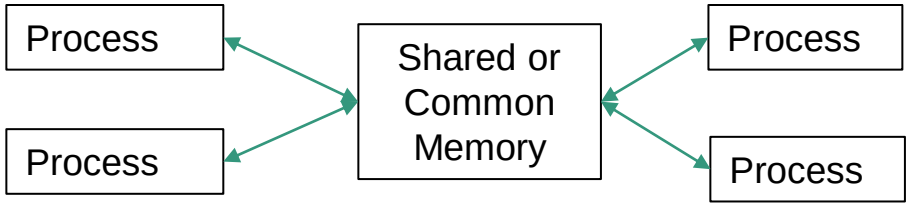
VARIATIONS OF THE MVC STYLE

- Model-View-Template
  - E.g., Django
- Model-View-View-Model (MVVM) and Model-View-Presenter (MVP)
  - See the differences between these 3 architectures

COMMUNICATION STYLES (SOLUTIONS) EVOLUTION

Timeline

Shared/Common memory

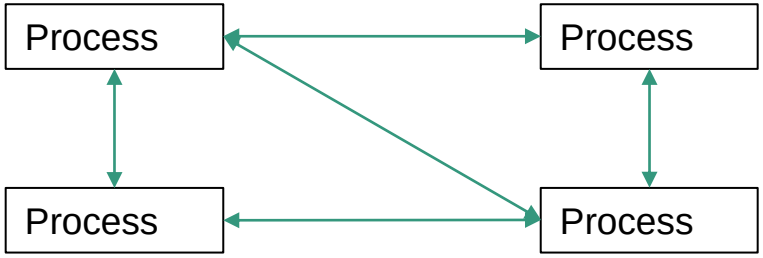


Processes write to and read data from a common or shared memory

Issues:

- Overwriting unauthorized areas
- Timing of updates
- Blocking/unblocking/deadlocking

Distributed systems with Point to point connection



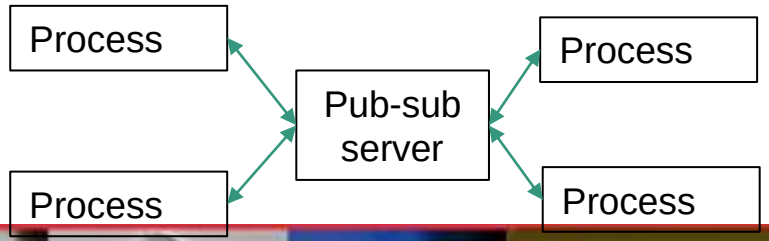
Each process manages its own data and exchanges with other processes

Advantage: data isolation and protection

Issues:

- Communication management is complex and tightly coupled
- When a link breaks, sections of the system are disabled for repairs

Publisher-subscriber



Process communicate through a server

Advantage: communication mechanism less complex and less coupled

Issues:

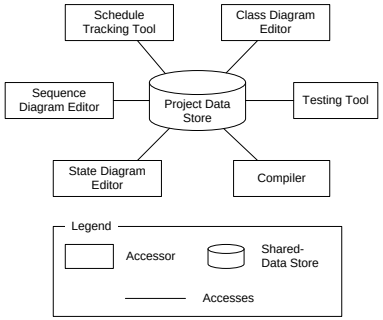
- Performance overhead
- Pub-server is a point of failure

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

SHARED-DATA STYLE

- Is a ***communication*** style
- One or more *shared-data stores* are used by one or more *shared-data accessors* that communicate **only** through the shared-data stores
- Two variants:
  - **Blackboard style**—The shared-data stores activate the accessors when the stores change
  - **Repository style**—The shared-data stores are passive and manipulated by the accessors

SHARED-DATA STYLE EXAMPLE – SOFTWARE DEVELOPMENT ENVIRONMENT



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

SHARED-DATA STYLE ADVANTAGES AND DISADVANTAGES

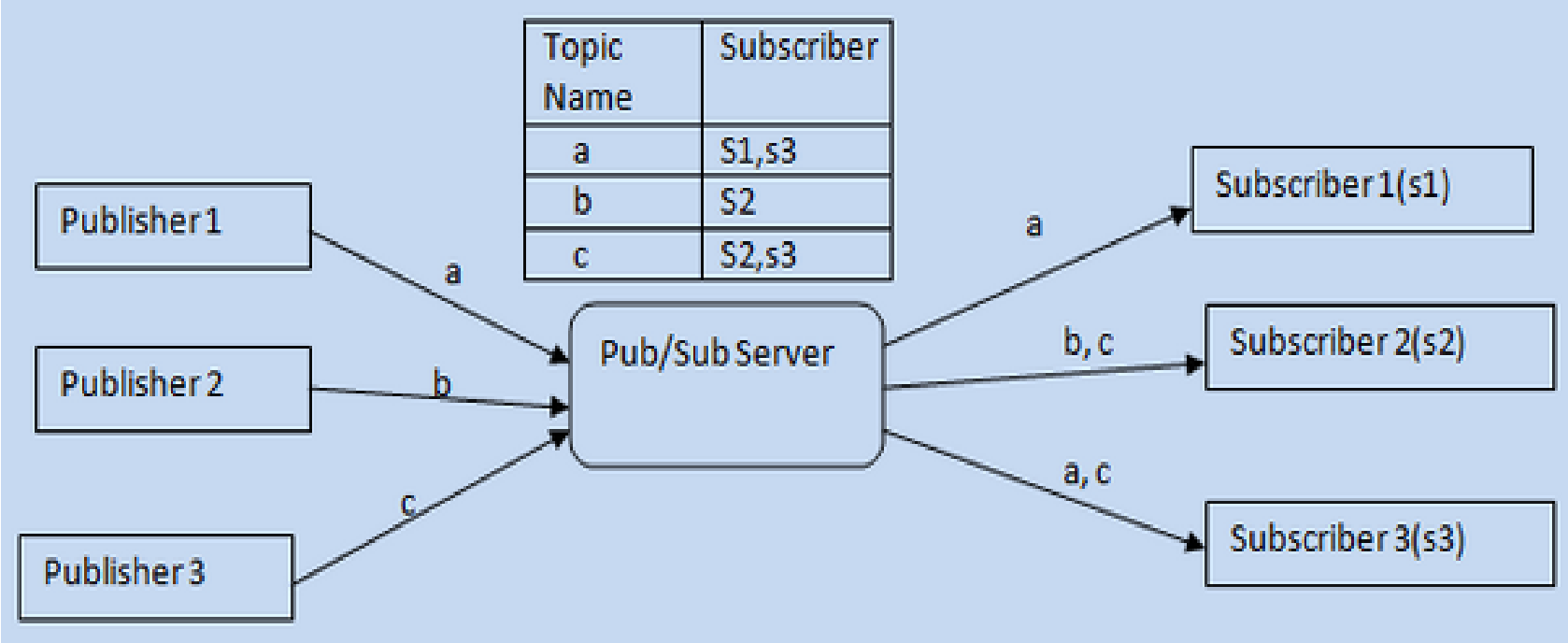
- **Advantages:**
  - **Modifiability** - Shared-data accessors communicate only through the shared-data store, so they are easy to change, replace, remove, or add to
  - Accessor independence increases **robustness** and **fault tolerance**
  - Placing all data in the shared-data store makes it easier to **secure and control**

- **Disadvantages:**
  - Forcing all data through the shared-data store may **degrade performance**
  - If the shared-data store fails, the entire program may **fail**

PUBLISH-SUBSCRIBE STYLE

- Is a ***communication*** (messaging) pattern
- Components:
  - Senders of messages (*publishers*)
  - Receivers of messages (*subscribers*)
- Rules (constrains):
  - Components do not interact directly, but through the connector
  - Publishers send (publish) message classes to a connector, agnostic of the subscribers
  - Subscribers express interest in one or more message classes (*subscribe*) and only receive messages that are of interest, without knowledge of which publishers provides it
  - The connector delivers corresponding messages from publisher to subscriber
- Components often run asynchronously

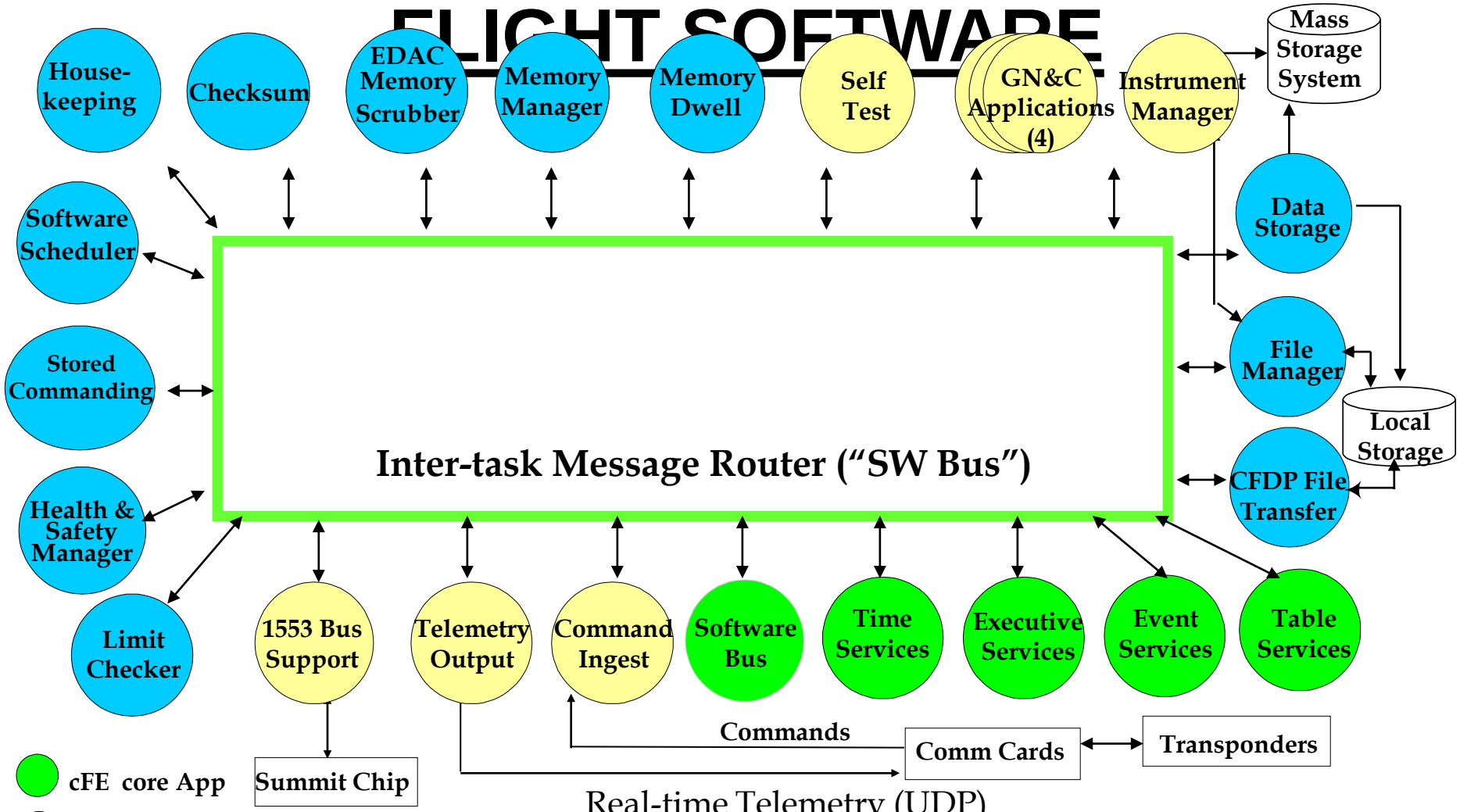
PUBLISH-SUBSCRIBE GENERIC REPRESENTATION



Components Connector Components

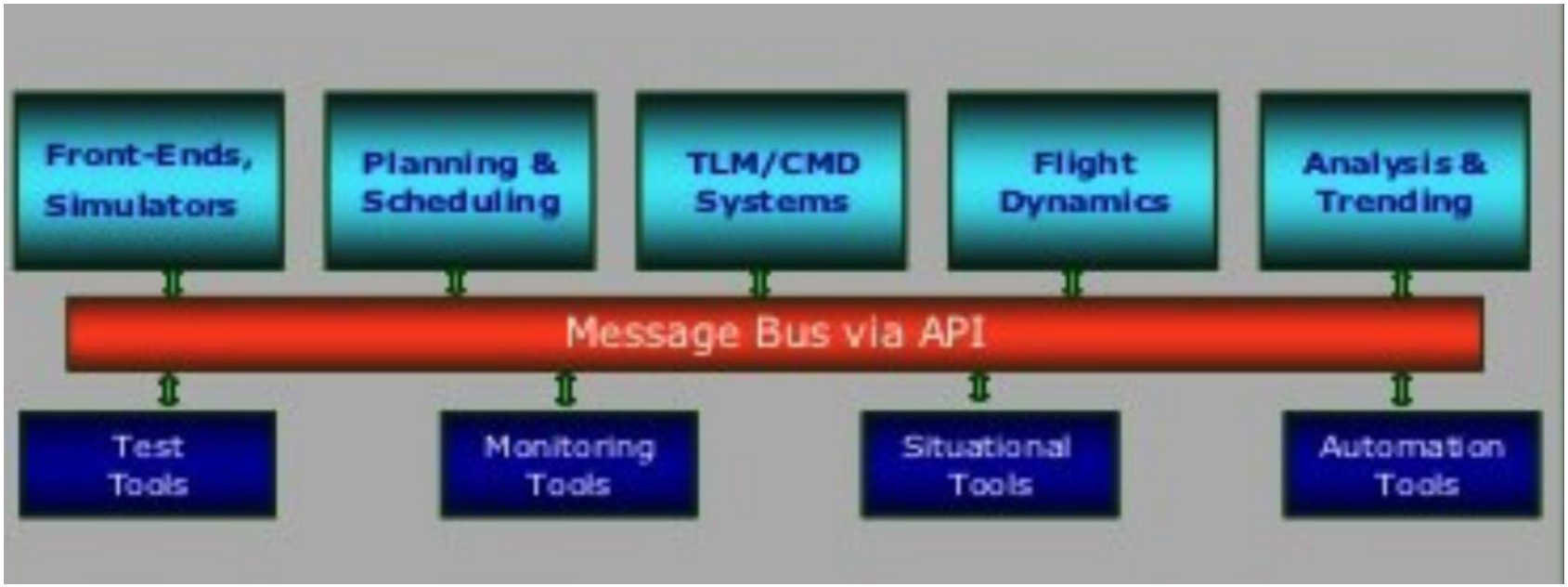
MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

PUBLISH-SUBSCRIBE EXAMPLE1: FLIGHT SOFTWARE



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

PUBLISH-SUBSCRIBE EXAMPLE2: GROUND SOFTWARE



From: Architecture Analysis of Systems based on Publish-Subscribe Systems by D. Ganesan

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

PUBLISH-SUBSCRIBE ADVANTAGES

- Loose coupling
- Scalability
  - New components can be added/removed at run-time
- Components can also dynamically unsubscribe to subscribed messages
- New variants of the system can be produced with different configurations of components
- The connector takes care of delivering messages to components

PUBLISH-SUBSCRIBE DISADVANTAGES

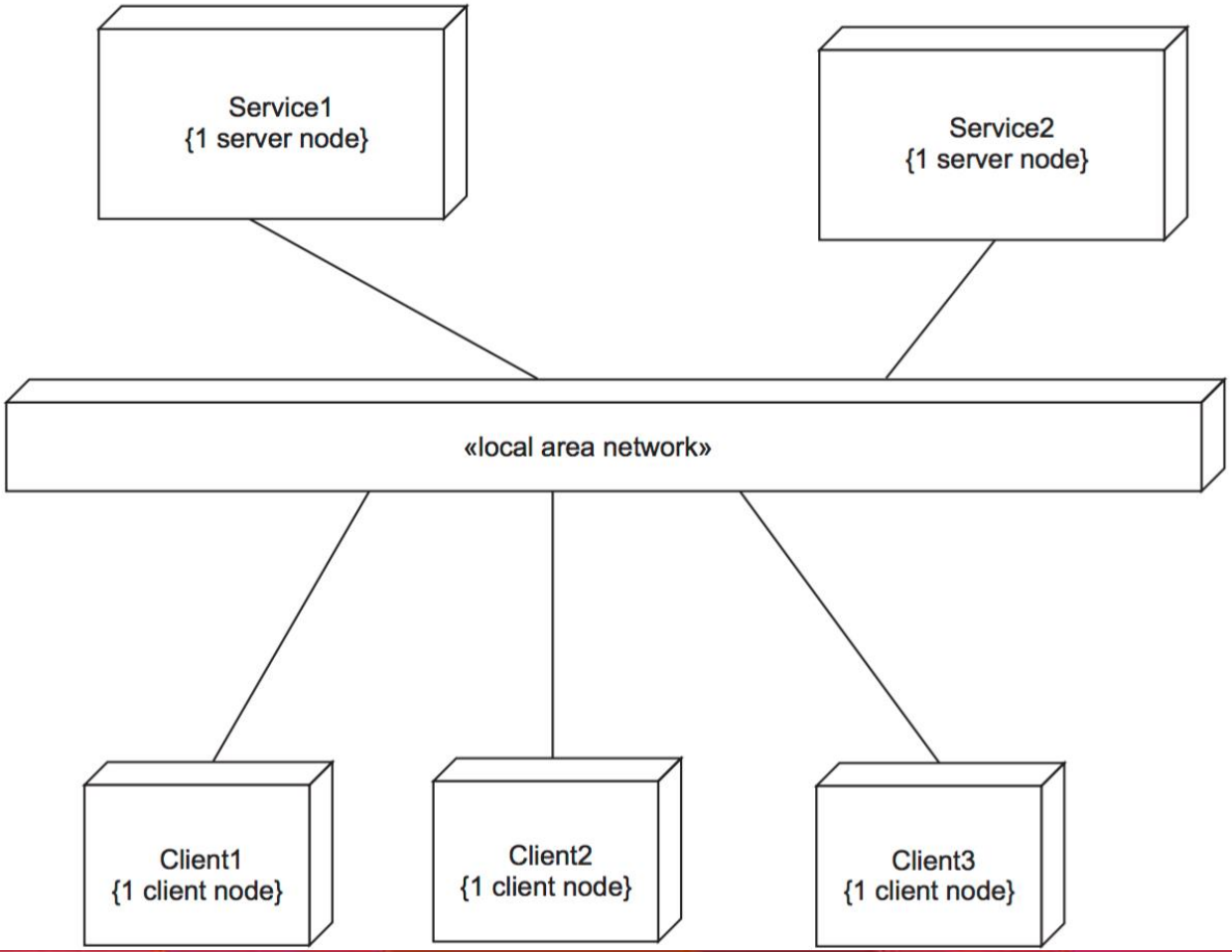
- There is overhead due to the connector's routing algorithm
  - Actual topology is known only at run-time
- Hard to analyze the implemented system because the control and data-flow information are not easy to track
- Functioning connector is crucial to all subscribers' functioning
- The correctness of the connector is crucial
  - If the message bus is not reliable then messages might be delivered to wrong subscribers

CLIENT-SERVER STYLE

- Is a ***distributed systems*** style
- A *server* is a hardware/software system that provides one or more services to multiple *clients*.
  - A *service* in a client/server system is an application software component that fulfills the needs of multiple clients
- *Client* request the server for its service
- Client and Server can run on different machines asynchronously
- Connectors help connecting clients to servers

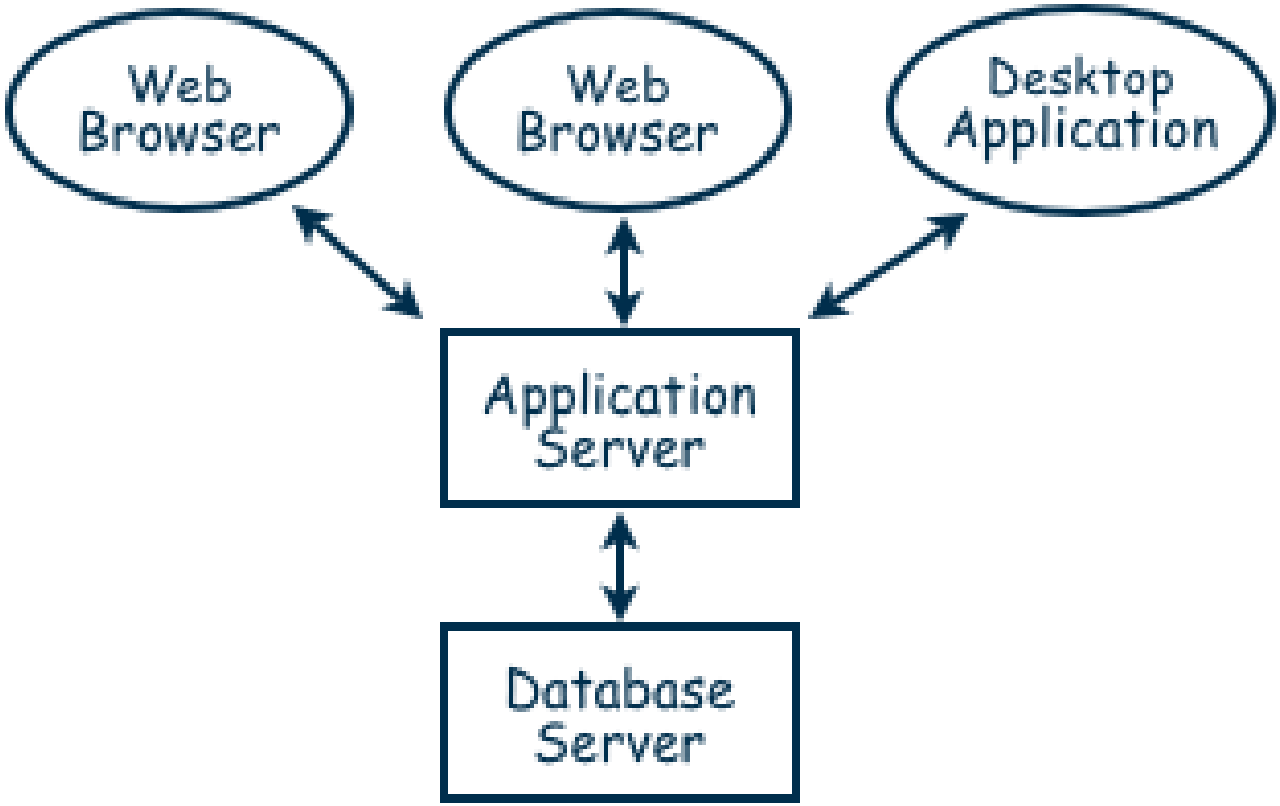
CLIENT-SERVER STYLE GENERIC REPRESENTATION

Multiple Clients, multiple servers



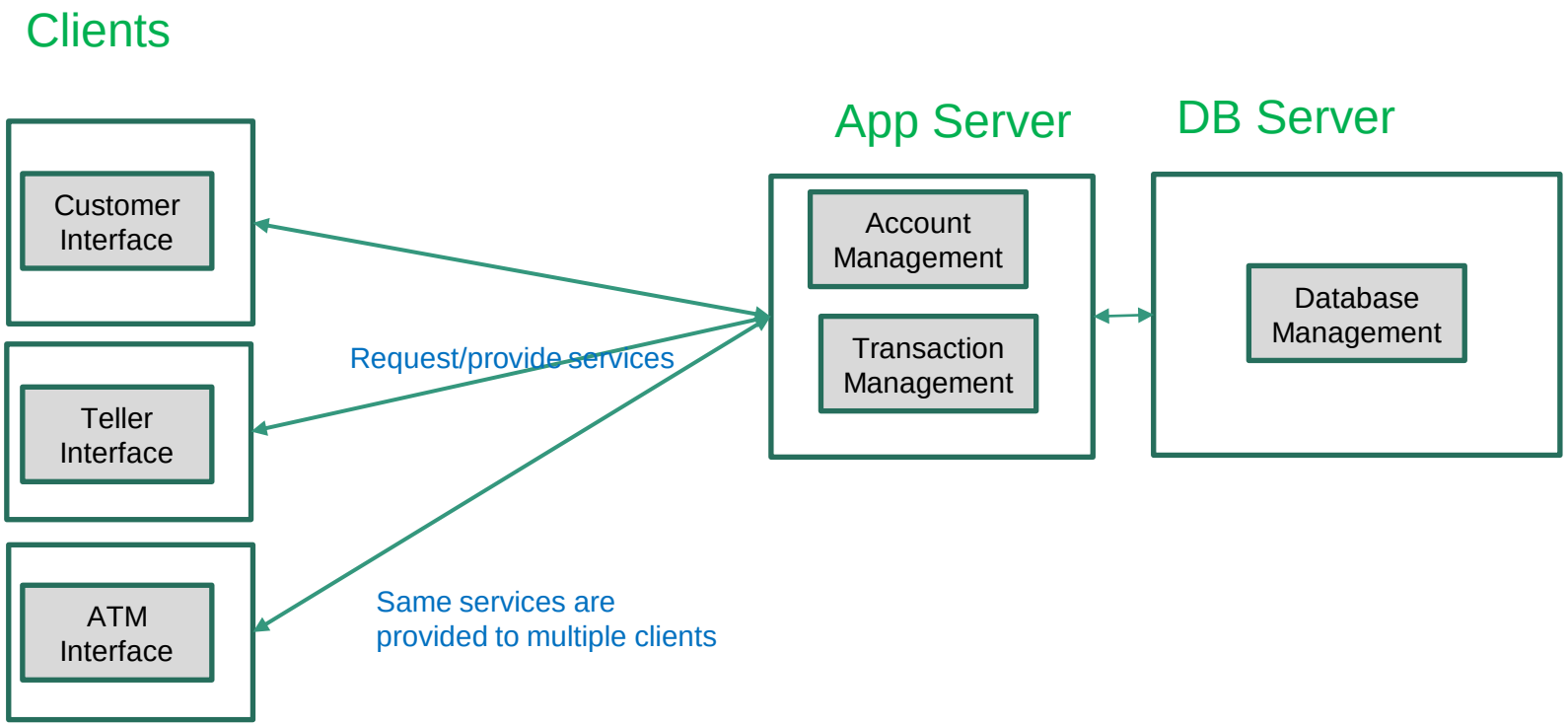
MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

CLIENT-SERVER STYLE EXAMPLE – WEB BASED APPLICATIONS



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

CLIENT-SERVER STYLE EXAMPLE – BANKING SYSTEM



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

CLIENT-SERVER STYLE ADVANTAGES

- Intuitive structure based on request/response paradigm
- Clear separation of responsibilities among the client and server
- Many clients can use a server
- Client and server can run concurrently
- Higher security and ease of maintenance

CLIENT-SERVER STYLE DISADVANTAGES

- Server must handle heavy traffic (if applicable)
  - Also load balancing is needed
- Additional complexity due to:
  - Thread creation/termination per connection
  - Thread pool management
  - Managing queue of requests from clients
- Must be prepared to handle network failures

SOFTWARE SERVICES

- A *software service* is a software component that can be accessed from remote computers over the Internet. Given an input, a service produces a corresponding output, without side effects.
- When a service request is made, the state information may be included as part of the request and the updated state information is returned as part of the service result.
- As there is no local state, services can be dynamically reallocated from one virtual server to another and replicated across several servers

MICROSERVICES STYLE

Material for this style from: Ian Sommerville, *Engineering Software Products*, Chapter 6

- Is a ***services*** style in which the system is constructed from communicating microservices
- Microservices are
  - Small-scale, stateless, services that have a single responsibility
  - Completely independent with their own database and UI management code

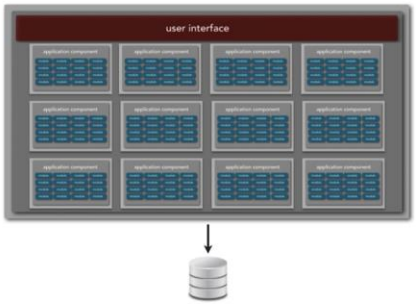
- Characteristics of microservices
  - Self-contained
  - Lightweight
  - Implementation-independent
    - May be implemented in different programming languages and use different technologies (e.g., databases)
  - Independently deployable
  - Business-oriented

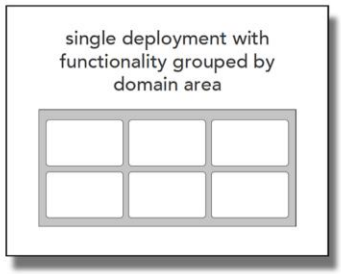
- A microservice architecture is appropriate, for example, to create cloud-based software products that are adaptable, scaleable and resilient.

MICROSERVICES STYLE (CONTINUED)

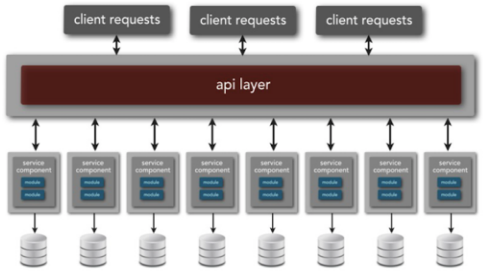
- Microservices are the alternative to a *monolithic* architecture
- A microservices architecture addresses two *problems with monolithic applications:*
  - The whole system has to be rebuilt, re-tested and re-deployed when any change is made. This can be a slow process as changes to one part of the system can adversely affect other components.
  - As the demand on the system increases, the whole system has to be scaled, even if the demand is localized to a small number of system components that implement the most popular system functions

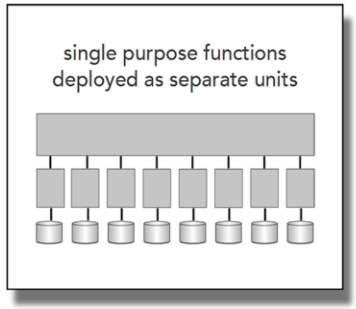
modular monolith





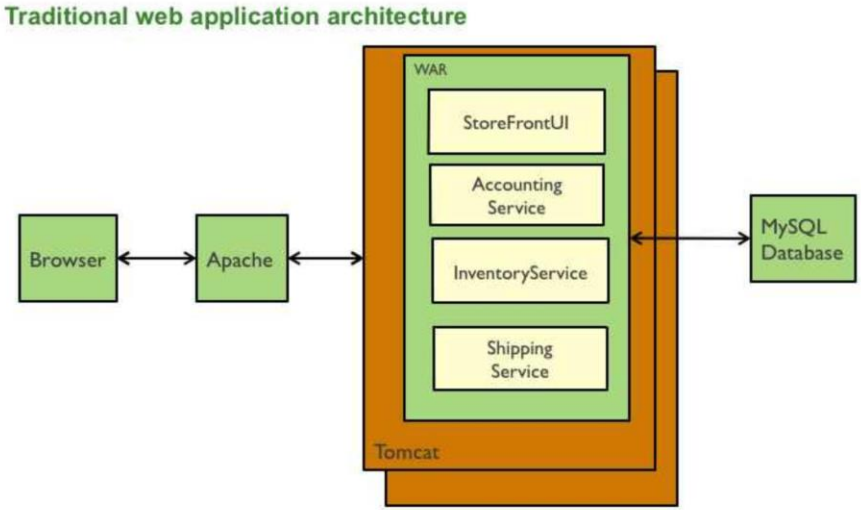
microservices architecture

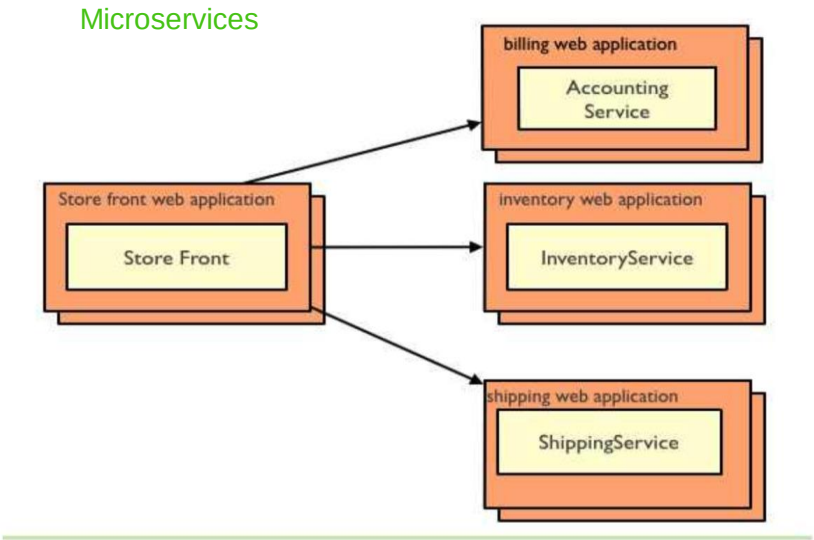




MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

MONOLITH AND MICROSERVICES EXAMPLE





MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

KEY DESIGN CONSIDERATIONS FOR MICROSERVICES ARCHITECTURE

- Identify the microservices
- Microservices communication and coordination
  - Microservice communication protocols
  - Data sharing
  - Whether services should be centrally coordinated
  - Failure management

- The RESTful architectural style is widely used in microservice-based systems
  - Services are designed so that the HTTP verbs, GET, POST, PUT and DELETE, map onto the service operations
- The RESTful style is based on digital resources that, in a microservices architecture, may be represented using XML or, more commonly, JSON

MICROSERVICES ADVANTAGES

- **Scalability -** If the demand on a service increases, service replicas can be quickly created and deployed
  - These do not require a more powerful server so ‘scaling-out’ is, typically, much cheaper than ‘scaling up’
- Each microservice can be independently developed, maintained, and deployed
  - Smaller and faster deployments
  - In cloud-based systems, each microservice may be deployed in its own container. A microservice can be stopped and restarted without affecting other parts of the system
- Isolated services have a better **failure tolerance**
- Microservices **reduce the time to market** and speed up the CI/CD pipeline
- **Understandability and maintainability** - easier to maintain and debug a lightweight microservice than a complex application

MICROSERVICES DISADVANTAGES

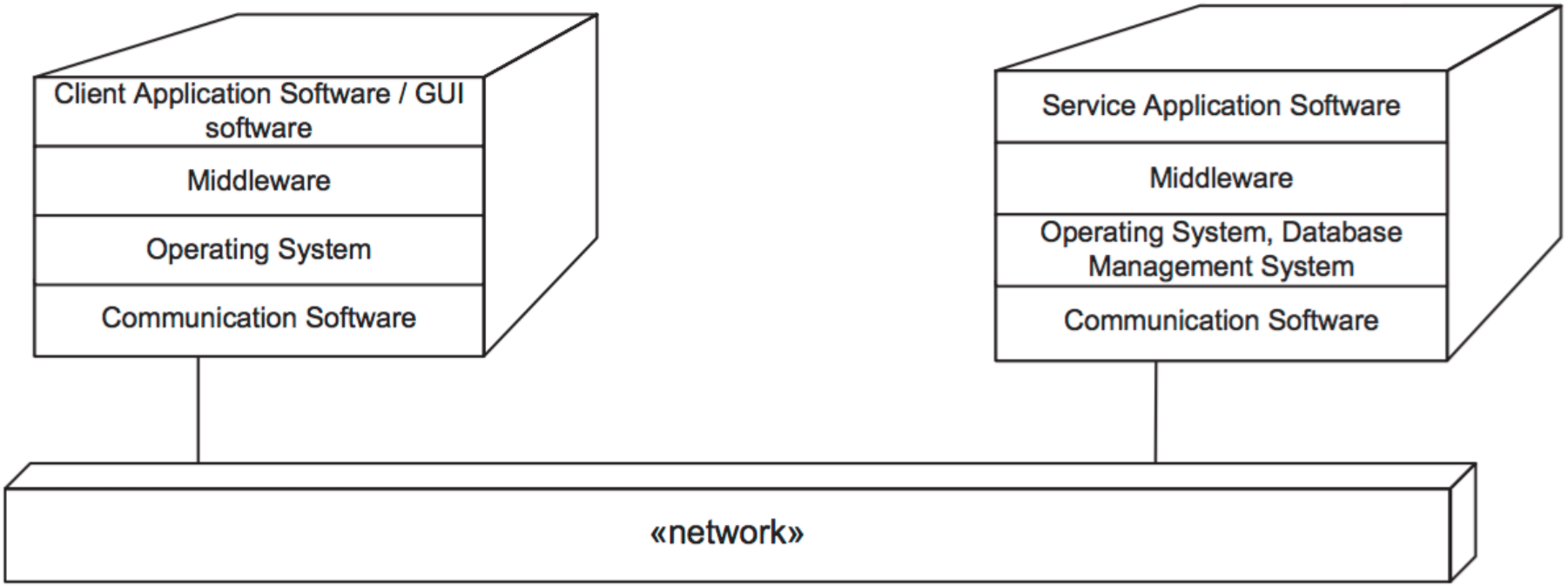
- Increased complexity of communication between services
- More services equals more resources - Multiple databases and transaction management can be painful
- When a system is composed of tens or even hundreds of microservices, **deployment** of the system is more complex than for monolithic systems
- Needs more collaboration (each team has to cover the whole microservice lifecycle)
- Harder to test, debug, and monitor
  - Because of the complexity of the architecture
- Poorer performance
  - Microservices need to communicate -> network latency, message processing, etc.
- Harder to maintain the network
  - Has less fault tolerance, needs more load balancing, etc.
- Security issues
  - Harder to maintain transaction safety, distributed communication goes wrong more likely, etc.
- Large vs small product companies - Microservices are great for large companies, but can be slower to implement and too complicated for small companies who need to create and iterate quickly, and don't want to get bogged down in complex orchestration.

COMPOSING STYLES

- Large systems can consist of different styles, often at different levels of abstraction
- Examples:
  1. The high-level structure could be a layer, and
     Components within a layer could follow publish/subscribe architecture style
     Two adjacent layers could interact through a pipe
  2. An overall a system may have a Layered style
     One layer may use the Event-Driven style, and another the Shared-Data style
  3. An overall system may have a Pipe-and-Filter style, but the individual filters may have Layered styles
- Styles can be composed as long as they do not violate style constraints
- Styles are selected and composed based on software's quality properties and their tradeoffs

EXAMPLE 1 OF USING MULTIPLE STYLES

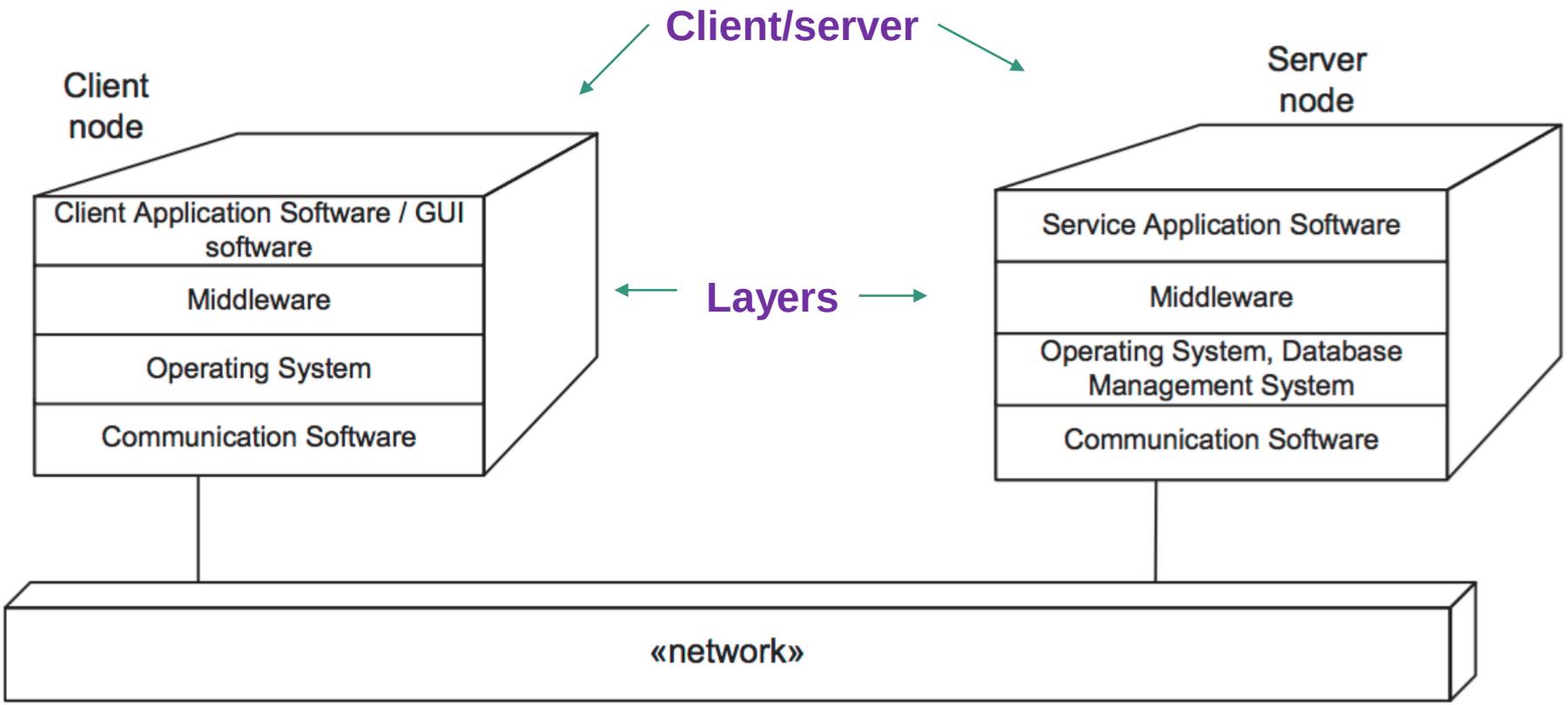
What styles are combined in this diagram?



[From : *Software Modeling and Design*, By Hassan Gomaa]

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

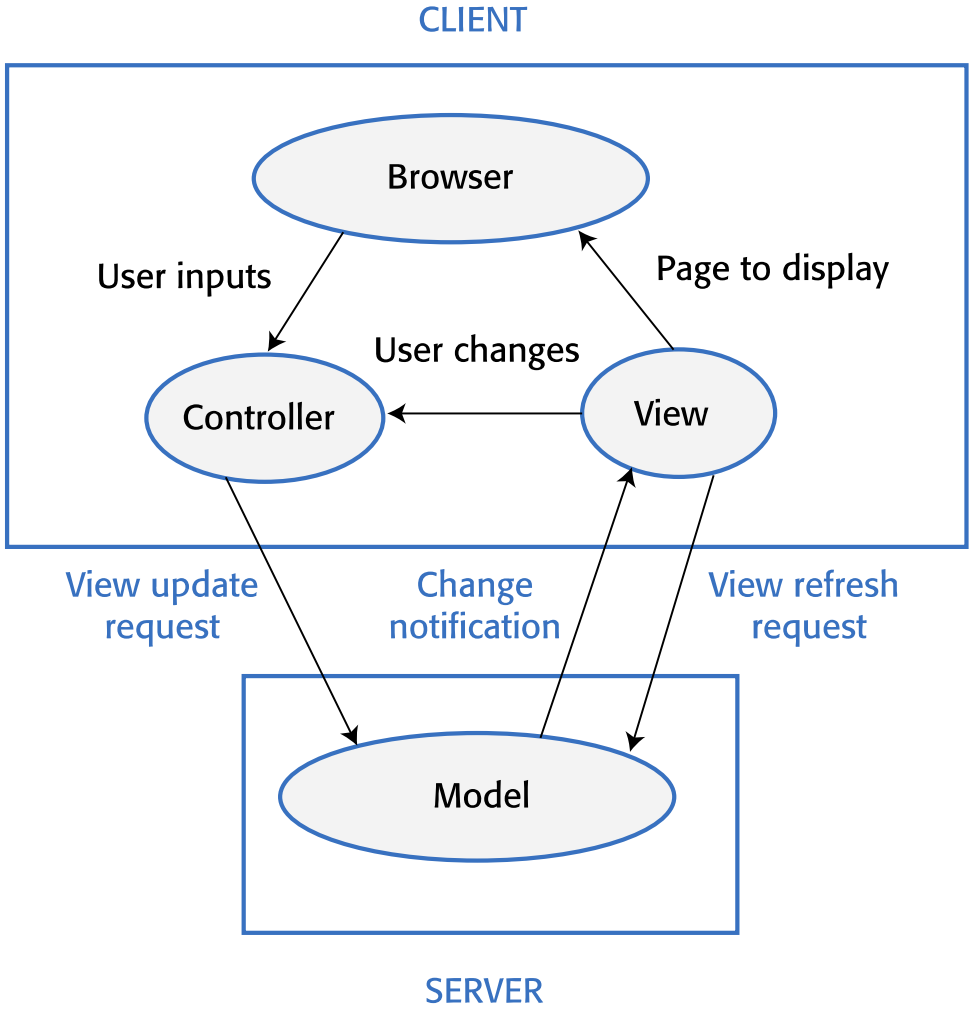
EXAMPLE 1 OF USING MULTIPLE STYLES



[From : *Software Modeling and Design*, By Hassan Gomaa]

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

EXAMPLE 2 OF USING MULTIPLE STYLES: CLIENT-SERVER AND MVC



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

CLASS EXERCISE

- What architectural style(s) would you consider for your class LMS software development?
- What is your rationale?

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

OUTLINE

- Lecture
  - Architecture design concerns
  - Architecture generation approaches and strategies
  - Design patterns / architecture styles
  - Quality driven design mechanisms/tactics
    - Application software security design
  - Reuse: designing with components and frameworks
  - Anti-patterns
- Class exercises: styles and tactics
- Assignments

We are here

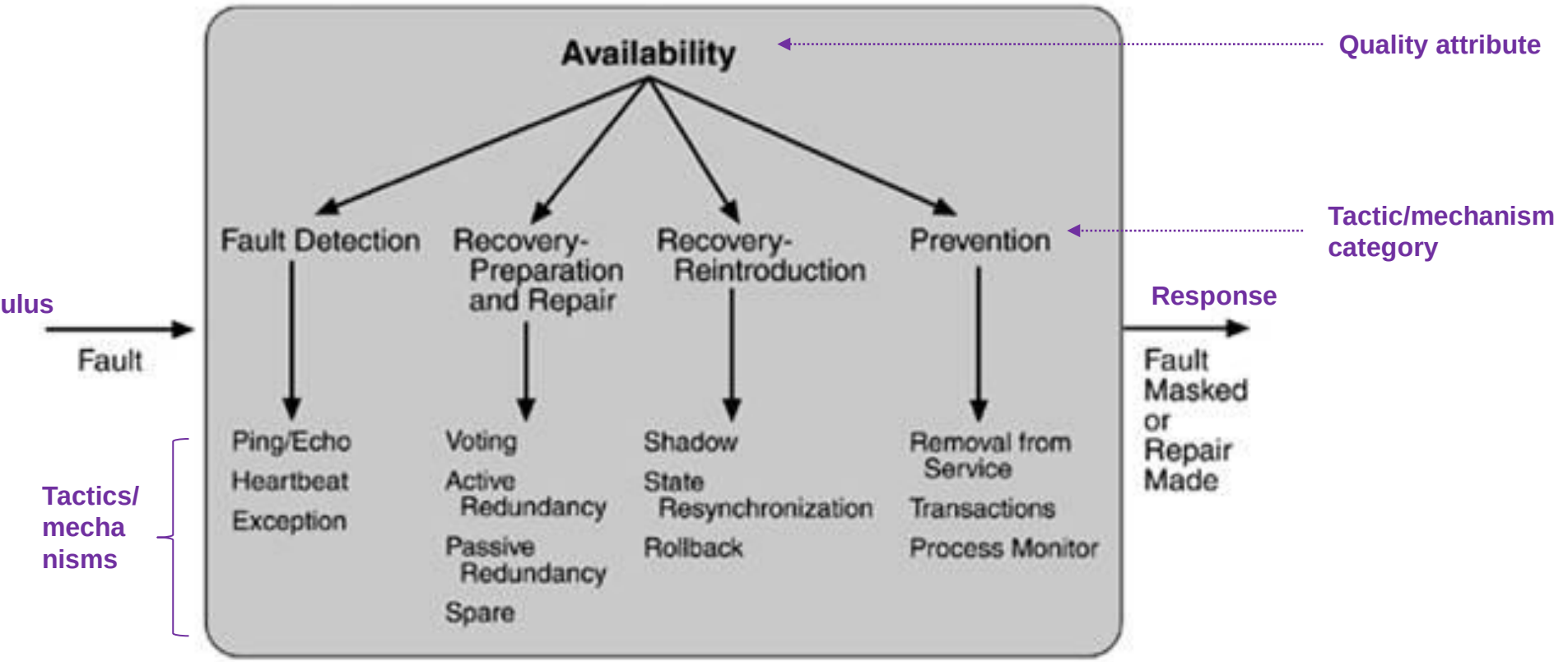
ARCHITECTURE TACTICS

- *Tactics* are fundamental elements of software architecture that an architect employs to meet a system's quality requirements
- A tactic (aka “mechanism”) is a design option/decision
- Quality-driven tactics – control responses to stimuli

[From: *Software Architecture in Practice* by Len Bass and Paul Clements]

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

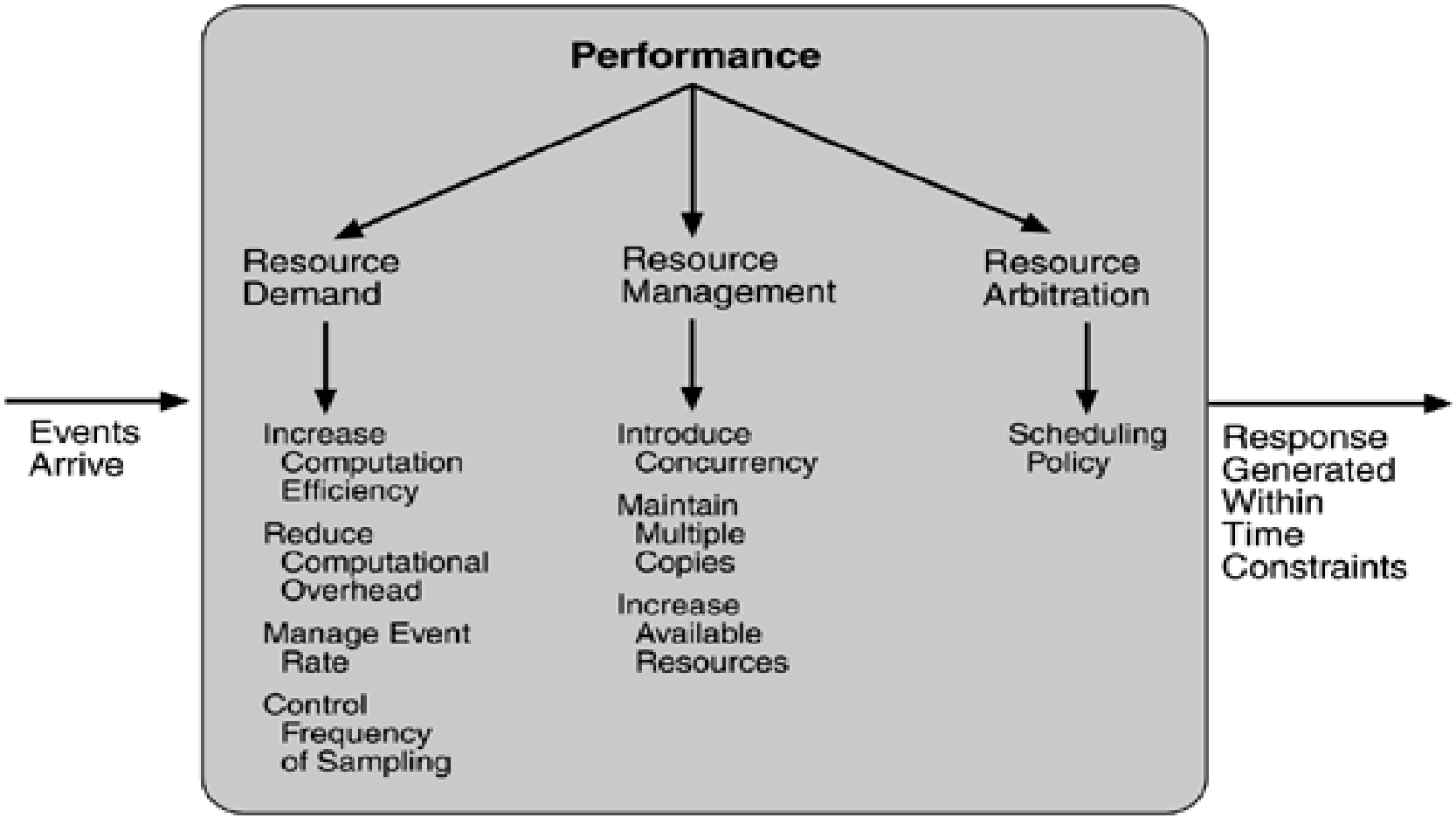
AVAILABILITY TACTICS



[From: *Software Architecture in Practice*, by Len Bass and Paul Clements]

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

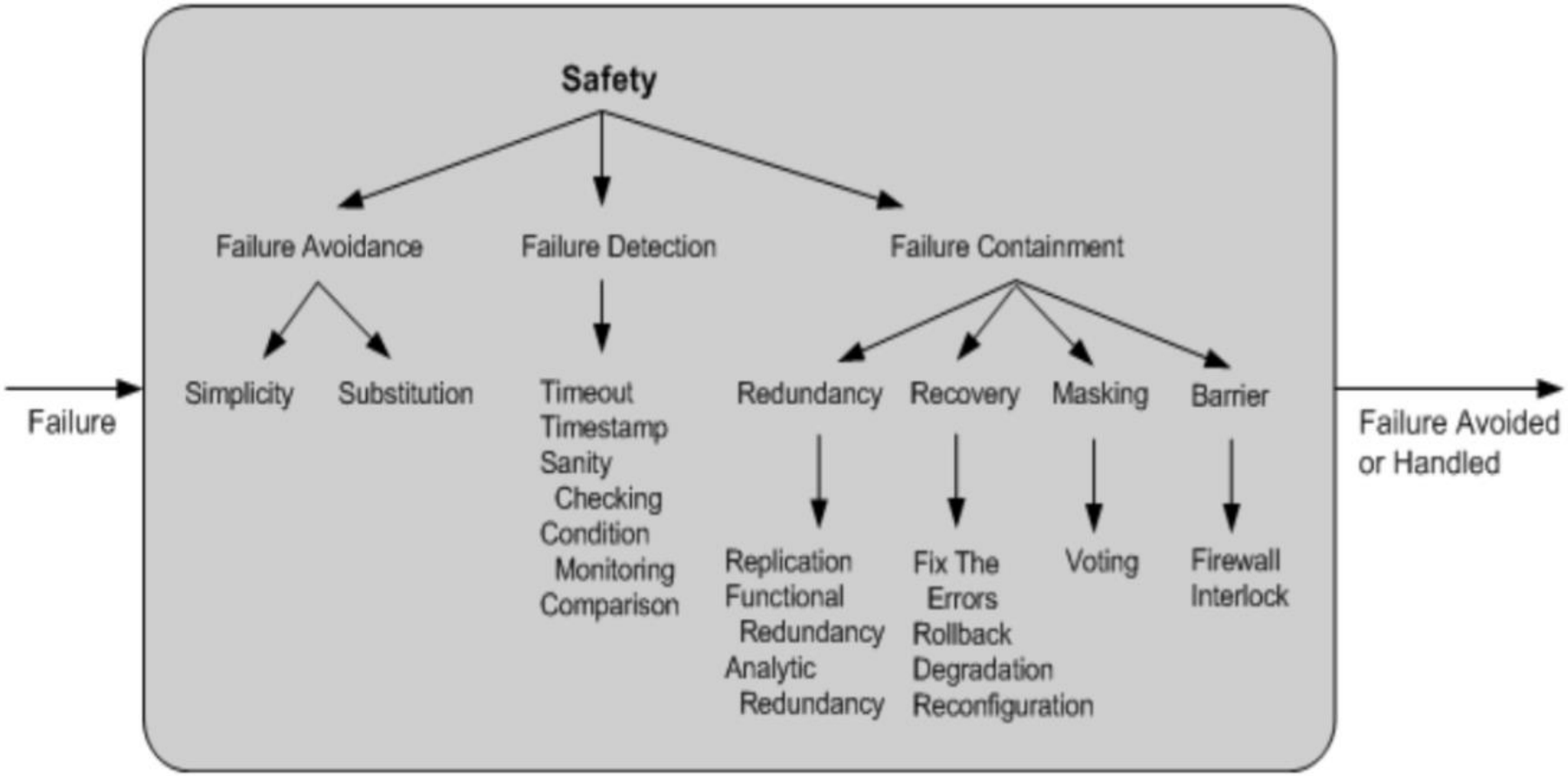
PERFORMANCE TACTICS



[From: *Software Architecture in Practice* by Len Bass and Paul Clements]

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

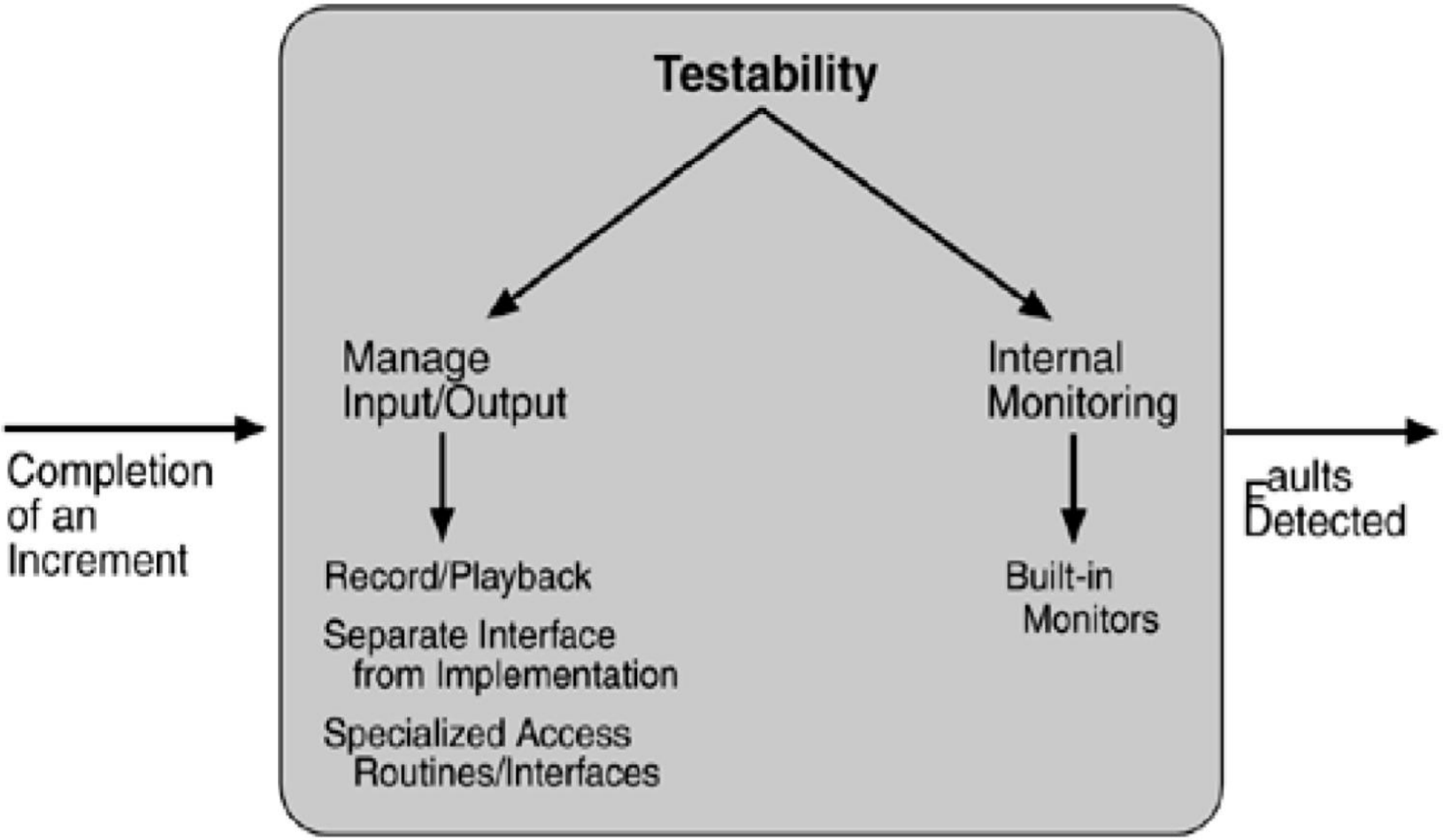
SAFETY TACTICS



[From: *Software Architecture in Practice*, by Len Bass and Paul Clements]

Example: Airbus FCS - software and hardware redundancy (watch Video by Ian Sommerville)

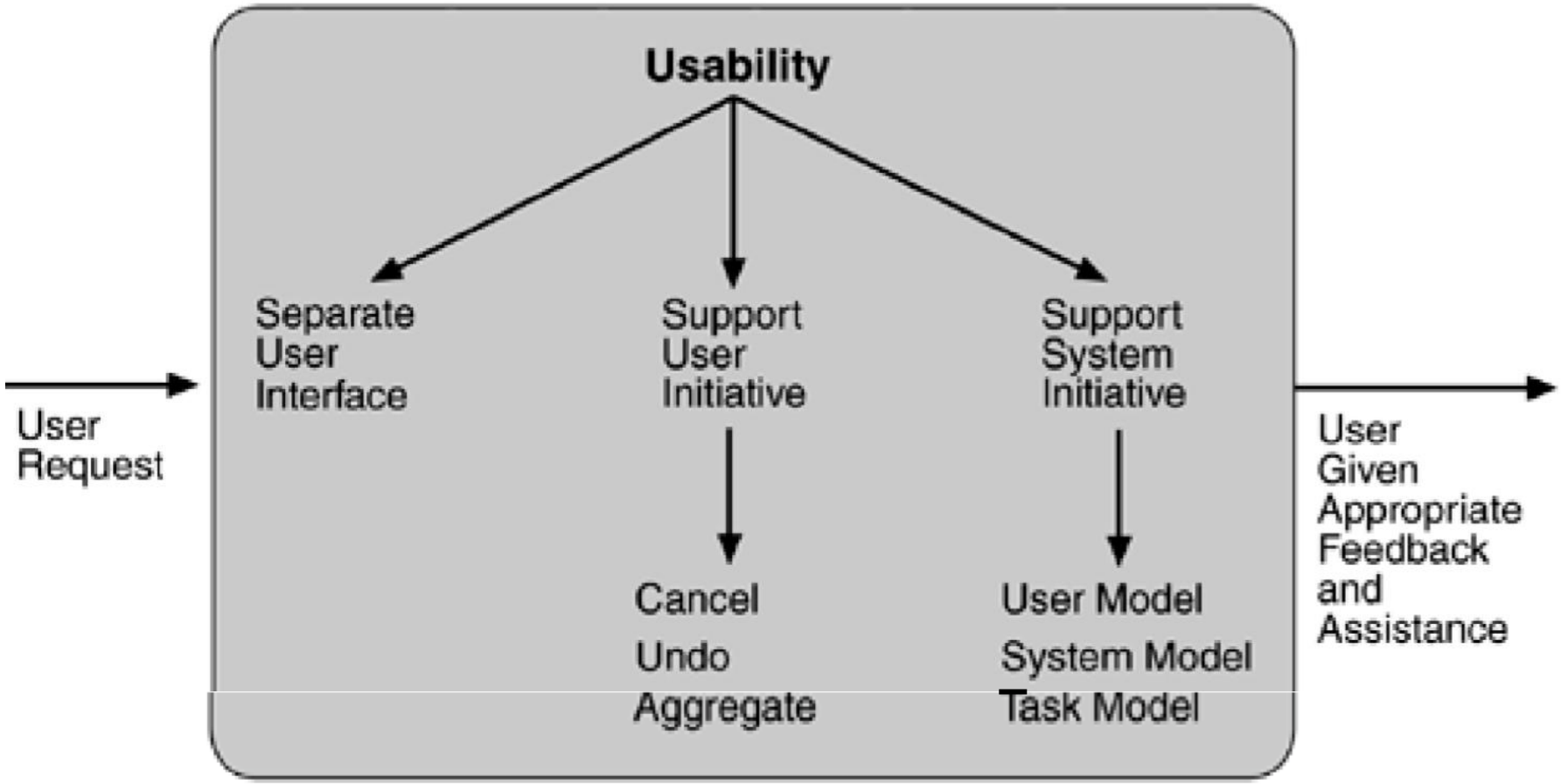
TESTABILITY TACTICS



[From: *Software Architecture in Practice*, by Len Bass and Paul Clements]

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

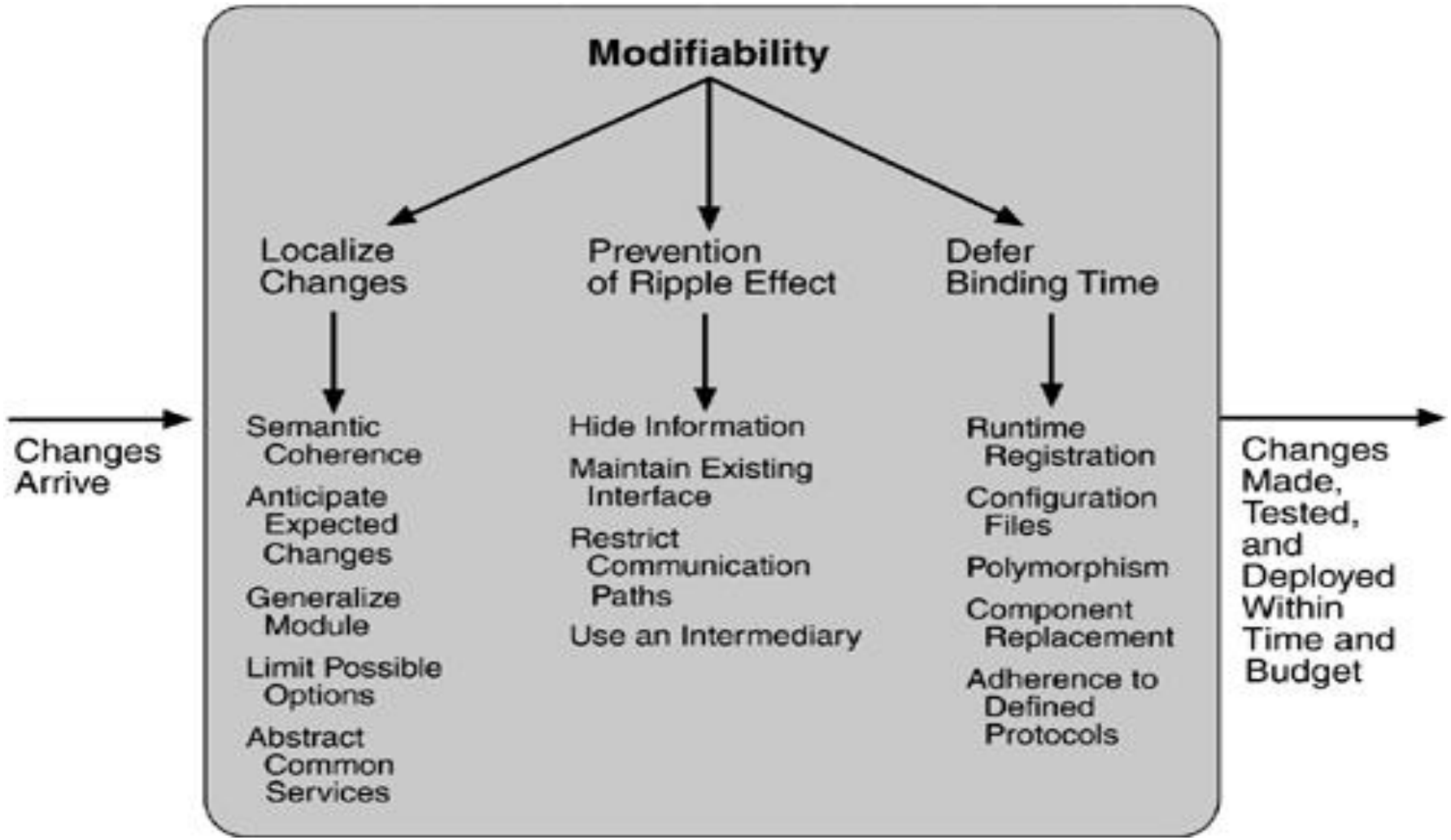
USABILITY TACTICS



[From: *Software Architecture in Practice*, by Len Bass and Paul Clements]

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

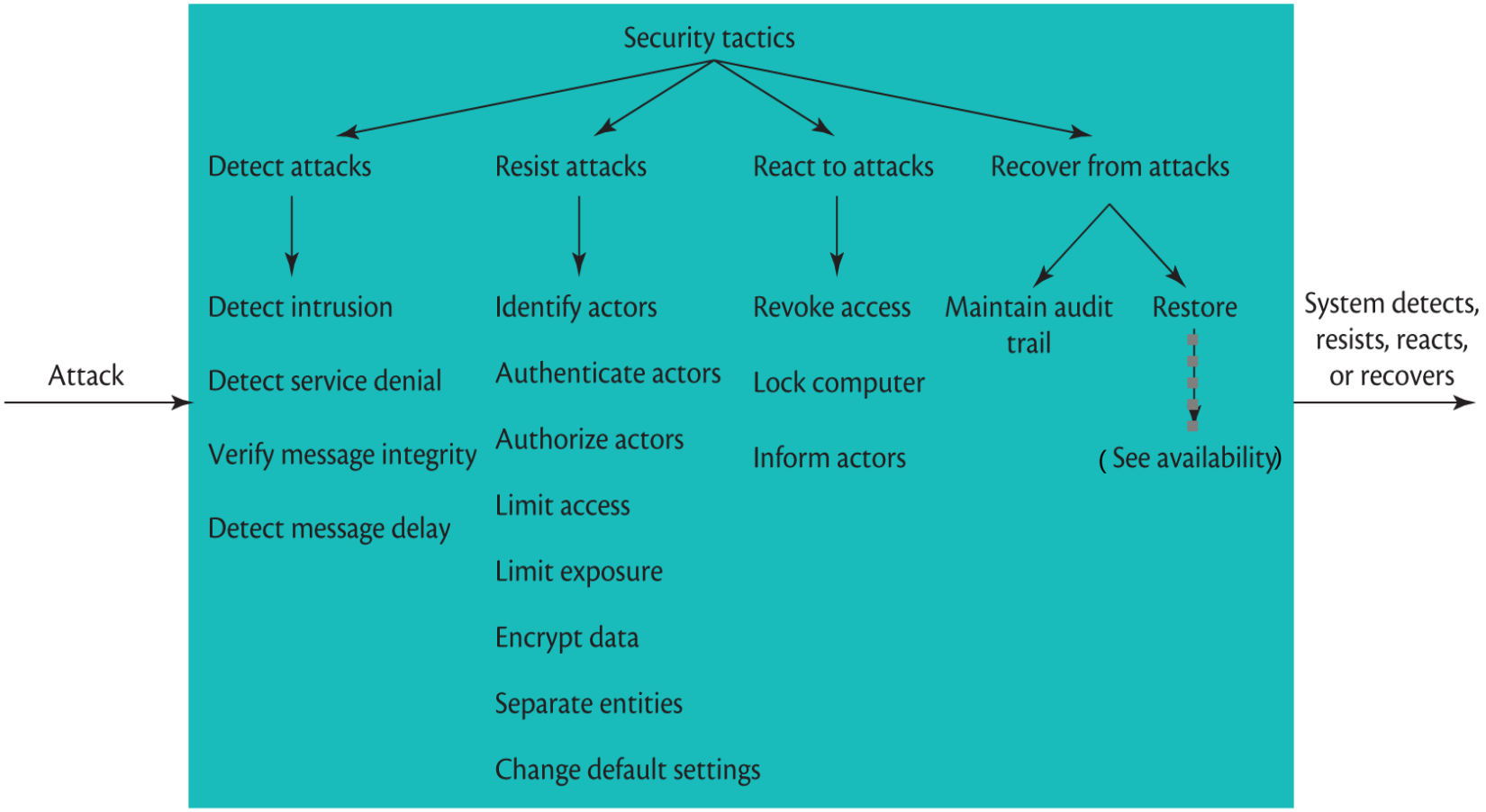
MODIFIABILITY TACTICS



[From: *Software Architecture in Practice*, by Len Bass and Paul Clements]

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

SECURITY TACTICS

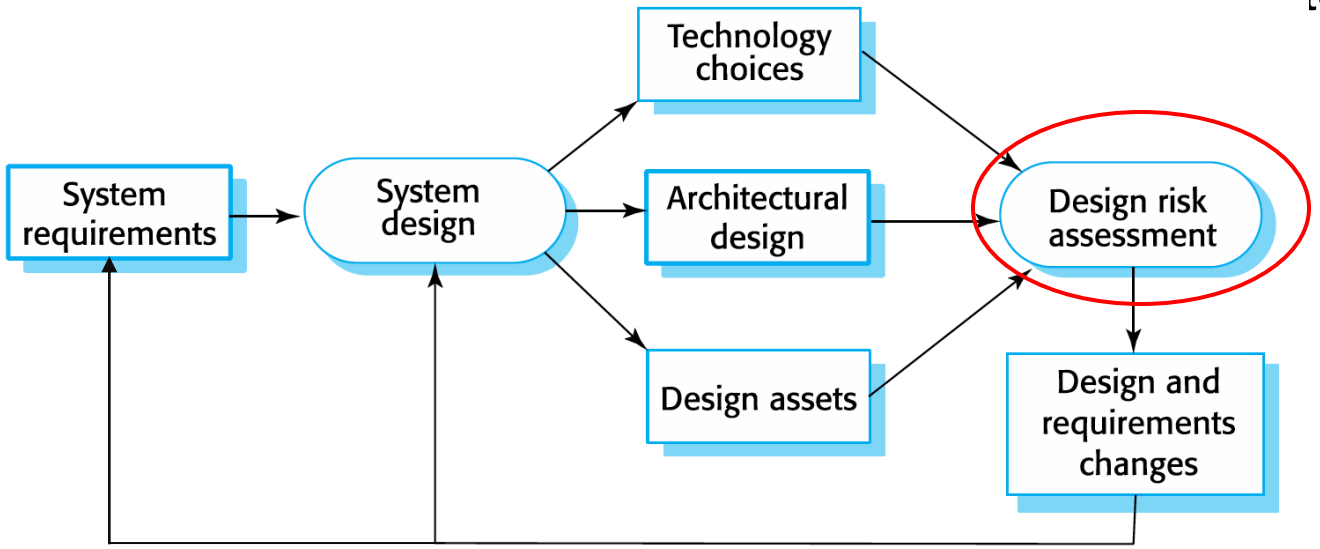


[From: *Software Architecture in Practice*, by Len Bass and Paul Clements]

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

DESIGN SECURITY RISKS

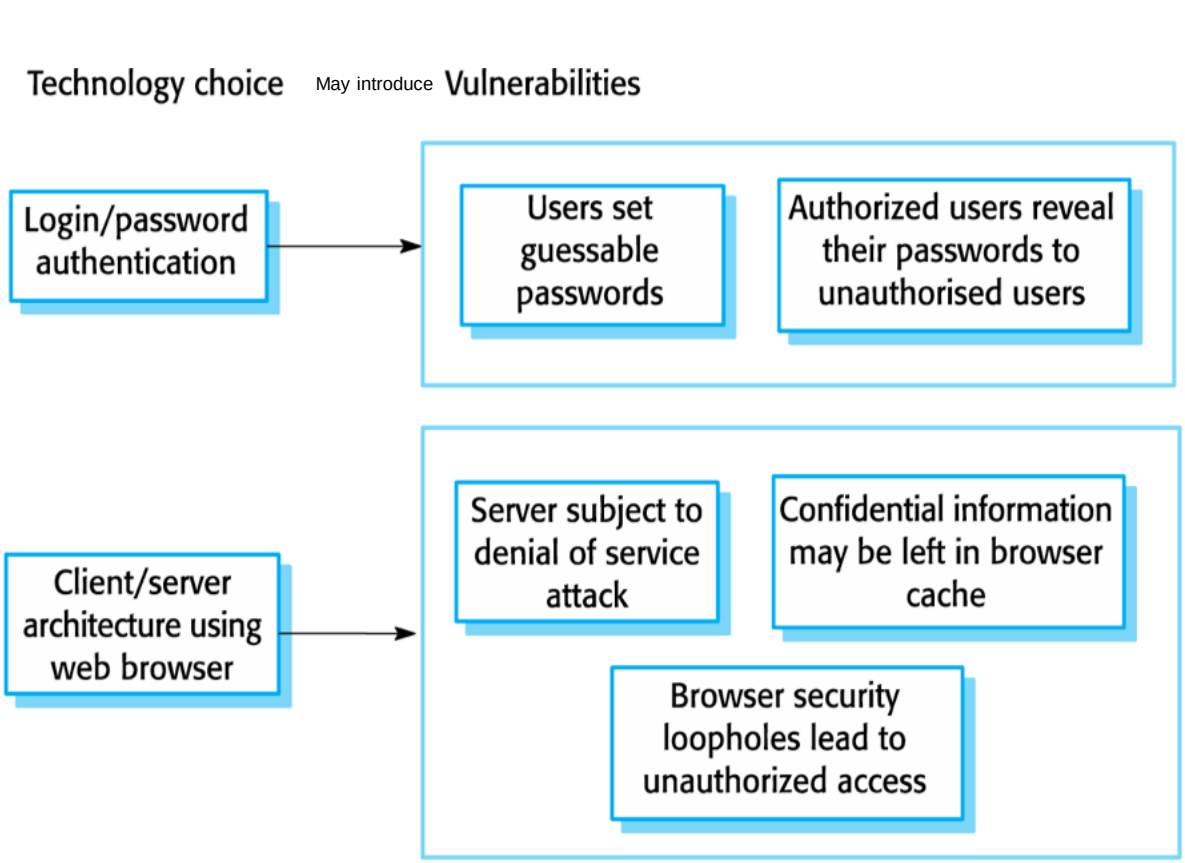
- *Security tactics* (mechanisms) **build-in security** during software design to protect (detect, resist, react, and recover) from attacks
- On the other hand, some design choices might **introduce** security vulnerabilities
  - [illegible] ted



[From: Ian Sommerville, *Software Engineering*, Ch 13]

GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

EXAMPLE OF VULNERABILITIES INTRODUCED BY TECHNOLOGY CHOICES AND DESIGN COUNTERMEASURES



SW control mechanisms

Countermeasures -> software and system des (and possibly requirements) changes

A password checker shall be made available and shall be run daily.

Weak passwords shall be reported to system administrators.

Access to the system shall only be allowed by approved client computers.

All client computers shall have a single, approved web browser installed by system administrators

Admin/operational (not SW) control mechanisms

[From: Ian Sommerville, *Software Engineering*, Ch 13]

GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

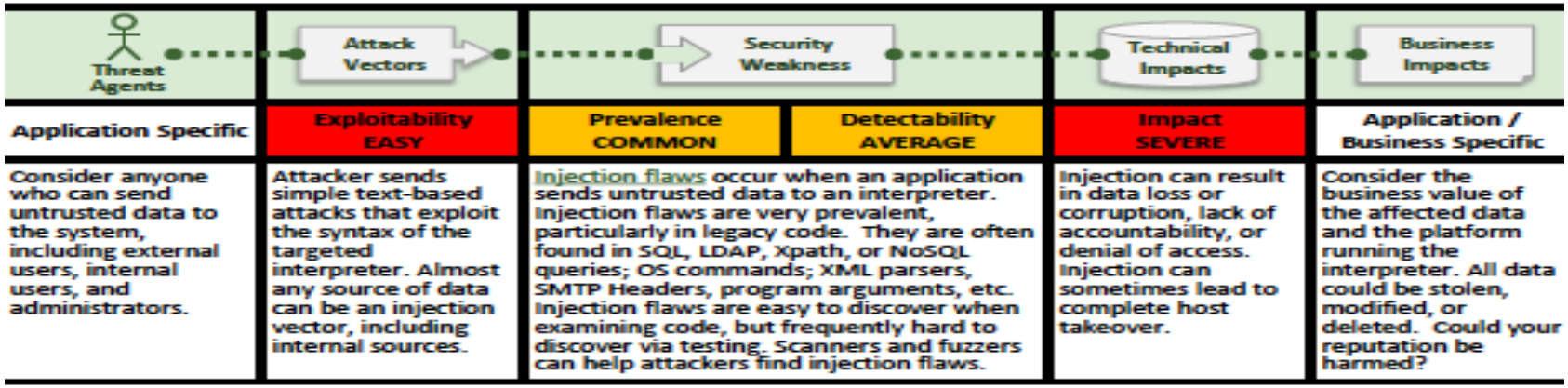
OWASP TOP 10 MOST CRITICAL WEB APPLICATIONS SECURITY RISKS

- The Open Web Application Security Project https://www.owasp.org
- OWASP Top 10 Most Critical Web Application Security Risks:
  - A1 Injection
  - A2 Broken Authentication
  - A3 Sensitive Data Exposure
  - A4 XML External Entities (XXE)
  - A5 Broken Access Control
  - A6 Security Misconfiguration
  - A7 Cross-Site Scripting (XSS)
  - A8 Insecure Deserialization
  - A9 Using Components with Known Vulnerabilities
  - A10 Insufficient Logging & Monitoring

OWASP TOP 10 MOST CRITICAL WEB APPLICATION SECURITY RISKS (CONTINUED)

- For each OWASP Top 10:
  - A description
  - Example vulnerabilities
  - Example attacks
  - **Guidance on how to avoid**
  - References to OWASP and other related resources
- Example – next slide

A1 Injection



Am I Vulnerable To Injection?

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures, and avoiding dynamic queries.

Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help a security analyst find the use of interpreters and trace the data flow through the application. Penetration testers can validate these issues by crafting exploits that confirm the vulnerability.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection flaws exist. Scanners cannot always reach interpreters and have difficulty detecting whether an attack was successful. Poor error handling makes injection flaws easier to discover.

How Do I Prevent Injection?

Preventing injection requires keeping untrusted data separate from commands and queries.

1. The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Be careful with APIs, such as stored procedures, that are parameterized, but can still introduce injection under the hood.
2. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. OWASP's ESAPI provides many of these escaping routines.
3. Positive or "white list" input validation is also recommended, but is not a complete defense as many applications require special characters in their input. If special characters are required, only approaches 1. and 2. above will make their use safe. OWASP's ESAPI has an extensible library of white list input validation routines.

Example Attack Scenarios

Scenario #1: The application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE
custID='" + request.getParameter("id") + "'";
```

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g., Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts
WHERE custID='" + request.getParameter("id") + "'");
```

In both cases, the attacker modifies the 'id' parameter value in her browser to send: ' or '1'='1. For example:

```
http://example.com/app/accountView?id=' or '1'='1
```

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify data or even invoke stored procedures.

References

OWASP

- OWASP SQL Injection Prevention Cheat Sheet
- OWASP Query Parameterization Cheat Sheet
- OWASP Command Injection Article
- OWASP XML eXternal Entity (XXE) Reference Article
- ASVS: Output Encoding/Escaping Requirements (V6)
- OWASP Testing Guide: Chapter on SQL Injection Testing

External

- CWE Entry 77 on Command Injection
- CWE Entry 89 on SQL Injection
- CWE Entry 564 on Hibernate Injection

TOP 10 MOST IMPORTANT SECURITY CONTROLS

C1: Define Security Requirements
C2: Leverage Security Frameworks and Libraries C3: Secure Database Access
C4: Encode and Escape Data
C5: Validate All Inputs
C6: Implement Digital Identity
C7: Enforce Access Controls
C8: Protect Data Everywhere
C9: Implement Security Logging and Monitoring
C10: Handle All Errors and Exceptions

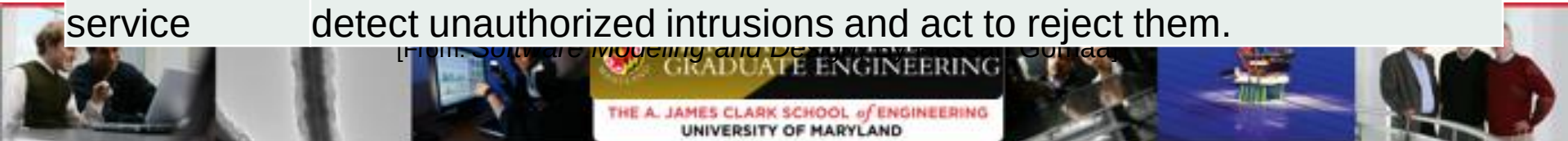
MORE SECURITY DESIGN GUIDANCE

- IEEE Center for Secure Design guidance for "Avoiding the Top 10 Software Security Design Flaws"
  - Earn or give, but never assume, trust
  - Use an authentication mechanism that cannot be bypassed or tampered with
  - Authorize after you authenticate
  - Strictly separate data and control instructions, and never process control instructions received from untrusted sources
  - Define an approach that ensures all data are explicitly validated
  - Use cryptography correctly
  - Identify sensitive data and how they should be handled
  - Always consider the users
  - Understand how integrating external components changes your attack surface
  - Be flexible when considering future changes to objects and actors

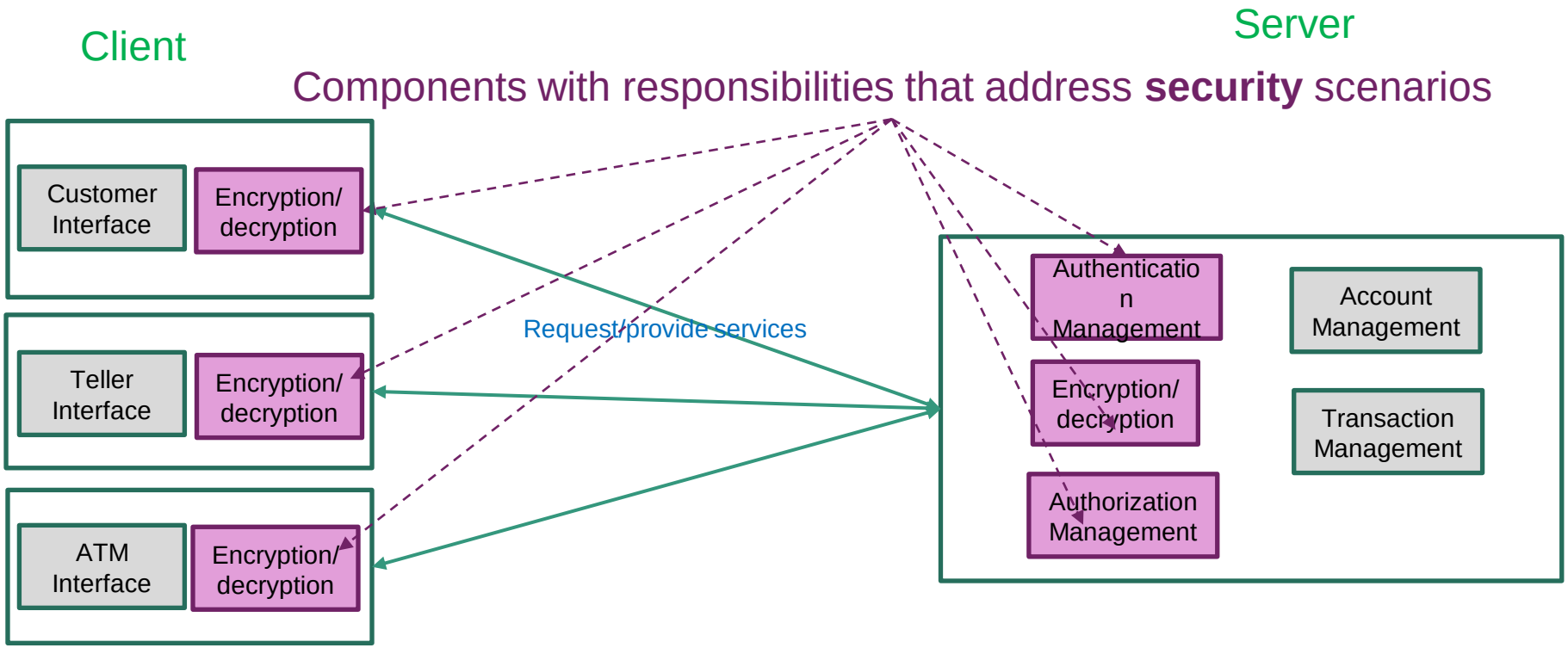
EXAMPLE SECURITY DESIGN MECHANISMS – FOR A BANKING SYSTEM

Potential threat	Design mechanism
System penetration	Encrypt the messages at source, particularly transactions originating at the ATM Client and the responses sent by the Banking Service, and then to decrypt the messages at the destination.
Authorization violation	A log of all access to the system needs to be maintained, so that cases of misuse or abuse can be tracked down, so that any abuse can be corrected.
Confidentiality disclosure	Secret information, such as card numbers and bank accounts, needs to be protected by an access control method that only allows users with the appropriate privileges to access the data.
Integrity compromise	An access control method needs to be enforced to ensure that unauthorized persons are prevented from making changes to application data in the database or communication data.
Repudiation	A log needs to be maintained of all transactions so that a claim that the transaction or activity did not occur can be verified by analyzing the log.
Denial of service	An intrusion detection capability is required so that the system can detect unauthorized intrusions and act to reject them.

[From: *Software Modeling and Design* – Gomaa]



EXAMPLE OF USING MECHANISMS – FOR THE BANKING SYSTEM



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

RELATION PATTERNS-TACTICS

- Typically, a pattern incorporates several tactics
- ***Patterns*** package ***tactics***, *e.g.*,
  - A pattern that supports availability will likely use both a *redundancy tactic* and a *synchronization tactic*
  - *Active Object* design pattern addresses “introduce concurrency" performance tactic, as well as:
    - Information hiding (modifiability)
    - Intermediary (modifiability)
    - Binding time (modifiability)
    - Scheduling policy (performance)

[From: *Software Architecture in Practice* (3rd Edition) 2012, by Len Bass and Paul Clements - Section 5.8]

PATTERNS AND TACTICS USAGE

- The *design process* involves making a judicious **choice** of what patterns and combination of tactics will achieve the system's desired goals
- The *architecture analysis* process involves **understanding** all of the patterns and tactics embedded in a design or implementation

CLASS EXERCISE

- Identify a quality scenario for your project LMS software development
- What tactic(s) should you use in your architecture design such that the software response satisfies this scenario?

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

OUTLINE

- Lecture
  - Architecture design concerns
  - Architecture generation approaches and strategies
  - Design patterns / architecture styles
  - Quality driven design mechanisms/tactics
    - Application software security design
  - Reuse: components, frameworks
  - Anti-patterns
- Class exercises: styles and tactics
- Assignments

We are here

COMPONENT-BASED DESIGN

- A *reuse-based approach* to defining, implementing and composing loosely coupled independent components into systems
  - Design with reuse
  - Modular and cohesive design
  - A component can be: software package, a web service, a web resource, or a module that encapsulates a set of related functions and/or data
- *Composition* instead of *decomposition*
- Short-term and the long-term benefits for the software itself and for the organizations
- Components are typically seen as “black boxes”
- Origin of components can be:
  - Developed in house (designed for reuse)
  - Off the shelf: COTS, GOTS, Open source
  - Custom developed (acquisition)

COMPONENT MODEL

- Is a definition of *properties* that components must satisfy, methods and mechanisms for the *composition* of components
- Examples:
  - Enterprise JavaBeans (EJB) model
  - Component Object Model (COM) model
  - .NET model
  - Common Object Request Broker Architecture (CORBA) component Model

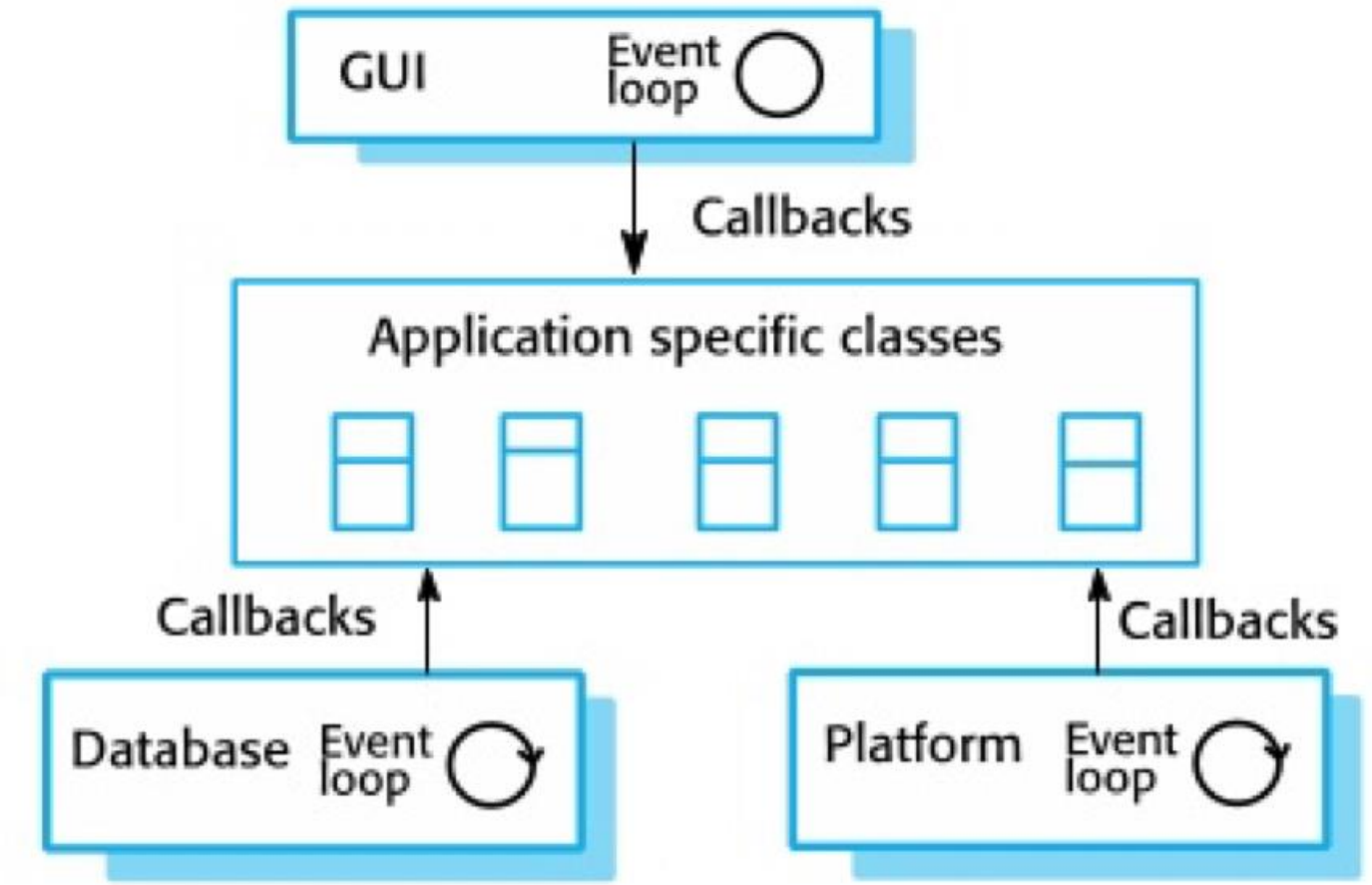
ARCHITECTURE *FRAMEWORKS*

- Abstraction in which
  - Software providing generic functionality can be selectively extended by **additional** user-written code -> application-specific software
- Provides a standard way to build and deploy applications
- A universal, *reusable* software environment
- Provides particular functionality as part of a larger software platform to facilitate development of software applications, products and solutions
- Bring together different components to enable development of a project or system
- May include:
  - Support programs
  - Code libraries
  - Tool sets
  - Application programming interfaces (APIs)

FRAMEWORK FEATURES

- Inversion of control
  - The overall program's flow of control is **not** dictated by the caller, but by **the framework** (unlike in libraries or standard user applications)
    - Framework code calls the application code, as needed
- Extensibility
  - A user can extend the framework - usually by selective overriding
  - Programmers can add specialized user code to provide specific functionality
- Non-modifiable framework code
  - Users can *extend* the framework, but should *not modify* its code

INVERSION OF CONTROL EXAMPLE



From: *Software Engineering*, Ian Sommerville

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

EXAMPLE FRAMEWORKS DOMAINS

- Web framework
  - ASP.NET, WebObjects, web2py, OpenACS, Catalyst, Mojolicious, Ruby on Rails, Laravel, Grails, Django, ReactJS, Vue.js, Spring, etc.
  - MVC pattern largely used
- Ajax framework / JavaScript framework
- Application framework - General GUI applications
- Oracle Application Development Framework
- Financial modeling applications
- Artistic drawing, music composition, and mechanical CAD
- Earth system modeling applications
- Decision support systems
- Media playback and authoring
- Middleware
- Cactus Framework - High performance scientific computing
- Enterprise Architecture framework

CHALLENGES IN DESIGNING WITH FRAMEWORKS

- Integration with legacy systems
- Framework gap
- Architectural mismatch
- If multiple frameworks are used in a system-> challenges:
  - Inversion of control
    - If there are more frameworks that call the same application simultaneously
  - Integrating functionality from different frameworks
  - Overlapping of frameworks components

[From: *Software Design,* by David Budgen, chapters 16-17]

OUTLINE

- Lecture
  - Architecture design concerns
  - Architecture generation approaches and strategies
  - Design patterns / architecture styles
  - Quality driven design mechanisms/tactics
    - Application software security design
  - Reuse: components, frameworks
  - Anti-patterns
- Class exercises: styles and tactics
- Assignments

We are here

ARCHITECTURE ANTI-PATTERNS

- Anti-patterns are *wrong solutions* to occurring problems
- Styles (patterns) and tactics = the *do*s (what to use/do)
- Anti-patterns = the *don't*s (what to avoid)

ARCHITECTURE ANTI-PATTERNS (CONTINUED)

Anti Pattern	Synopsis	Refactored Solution
Architecture by Implication	**System that is developed without a documented architecture; often due to overconfidence based on recent success.**	**Define architecture in terms of multiple viewpoints corresponding to system stakeholders.**
Autogenerated Stovepipe	Automatic generation of interfaces for distributed, large-scale systems from fine grain header files.	Separation of the architecture-level framework design from the subsystem-specific design is essential to manage complexity.
Cover Your Assets	Document driven software processes often employ authors who list alternatives instead of making decisions.	Establish clear purposes and guidelines for documentation tasks; inspect the results for the value of documented decisions.
Design by Committee	Committee designs are overly complex and lack a common architectural vision.	Proper facilitation and software development roles can lead to much more effective committee-based processes.
Intellectual Violence	People sometimes use obscure references to esoteric papers, theories, and standards for intimidation or short-term gain.	Individuals and the organization should encourage and practice mutual education and mentoring.
Jumble	Interface designs are an un-factored mixture of horizontal and vertical elements, leads to frequent interface changes, lack of reusability.	Partition architectural designs with respect to horizontal, vertical, and metadata elements.
Reinvent the Wheel	**Legacy systems with overlapping functionality that don't interoperate. Every system built in isolation.**	**Use architecture mining and "best of breed" generalization to define a common interface, then object wrapping to integrate.**

ARCHITECTURE ANTI-PATTERNS (CONTINUED)

Anti Pattern	Synopsis	Refactored Solution
Spaghetti Code	**An ad hoc software structure makes it difficult to extend and optimize code.**	**Frequent code refactoring can improve software structure, support software maintenance, and iterative development.**
Stovepipe Enterprise	**Uncoordinated software architectures lead to lack of adaptability, reuse, and interoperability.**	**Use enterprise architecture planning to coordinate system conventions, reuse, and interoperability.**
Stovepipe System	**Ad hoc integration solutions and lack of abstraction lead to brittle, un-maintainable architectures**	**Proper use of abstraction, subsystem facades, and metadata leads to adaptable systems.**
Swiss Army Knife	Over-design of interfaces leads to objects with numerous methods that try to anticipate every possible need -- leads to designs that are difficult to comprehend, utilize, and debug, as well as implementation dependencies.	Defining a clear purpose for the component and properly abstracting the interface is essential to managing complexity.
The Grand Old Duke of York	**Four out of Five developers cannot define good abstractions; this leads to excess complexity.**	**Project teams should have designated architects who are abstractionists, i.e. possess the architecture instinct.**

ARCHITECTURE ANTI-PATTERNS (CONTINUED)

Anti Pattern	Synopsis	Refactored Solution
Vendor Lock-In	**Proprietary, product-dependent architectures do not manage complexity and lead to a loss of control of the architecture and maintenance costs.**	**Providing an isolation layer between product-dependent interfaces and the majority of application software enables management of complexity and architecture.**
Warm Bodies	Large software project teams make for ineffective organizations and overruns. Heroic programmers are essential.	Small projects (4 people in 4 months) are much more likely to produce software success.
Wolf Ticket	**A technology is assumed to have positive qualities due to its open systems packaging or claimed standards compliance. Few standards have test suites (< 6%) and few products are actually tested for conformance.**	**Discover the real truth behind the claims. Question authority. Assume nothing. Shift the burden of proof to the marketing organization. Talk directly to the technical product experts and developers.**

SUMMARY

- Architecture generation approaches and strategies
- Design patterns and architecture styles
- Quality driven design tactics (mechanisms)
- Reuse: components, frameworks
- Anti-patterns

YOUR ARCHITECTURE DESIGN "TOOLBOX"

Which one(s) to choose????

https://twitter.com/KentBeck/status/596007846887628801

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

OUTLINE

- Lecture
  - Architecture design concerns
  - Architecture generation approaches and strategies
  - Design patterns / architecture styles
  - Quality driven design mechanisms/tactics
    - Application software security design
  - Reuse: components, frameworks
  - Anti-patterns
- Class exercises: styles and tactics
- Assignments

We are here

ARCHITECTURE DESIGN QUIZ

- Demonstrate knowledge and understanding of architecture design drivers, methods, tactics, styles
- Individual assignment, online

SOFTWARE DEVELOPMENT PROJECT ASSIGNMENT

- Tasks:
  - Select architecture style(s) for LMS
    - explain rationale, list advantages and potential drawbacks
  - Revisit and update (as needed) the quality scenarios (utility tree)
  - (Re)prioritize scenarios (as needed)
  - Select and document tactics and mechanisms for the top priority quality scenarios
  - Revise preliminary software architecture
    - Represent the selected ***styles*** and ***tactics*** (*mechanisms)* for **your** software (not generic)
    - Represent the components of **your** LMS, as they fit into the selected style(s) and tactics
- Team assignment – only one team member needs to submit
- Online submission – submit to the *Architecture styles and tactics* assignment (MS Word or pdf file)
  - Submission is not graded, but is **mandatory**

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

BONUS MATERIAL
SYSTEMS ENGINEERING/DESIGN

RESOURCES

- The following slides are from Ian Sommerville – Software Engineering 10th edition, Chapters 19, 20, 21
  http://iansommerville.com/software-engineering-book/

- Systems engineering videos:
  - John McCarthy (UMD)
    https://www.youtube.com/watch?v=gQfdVMxFKPI
  - https://www.youtube.com/watch?v=7rCp9yAPm2A
  - Models in systems engineering:
    https://www.youtube.com/watch?v=qjL7FeFN59w

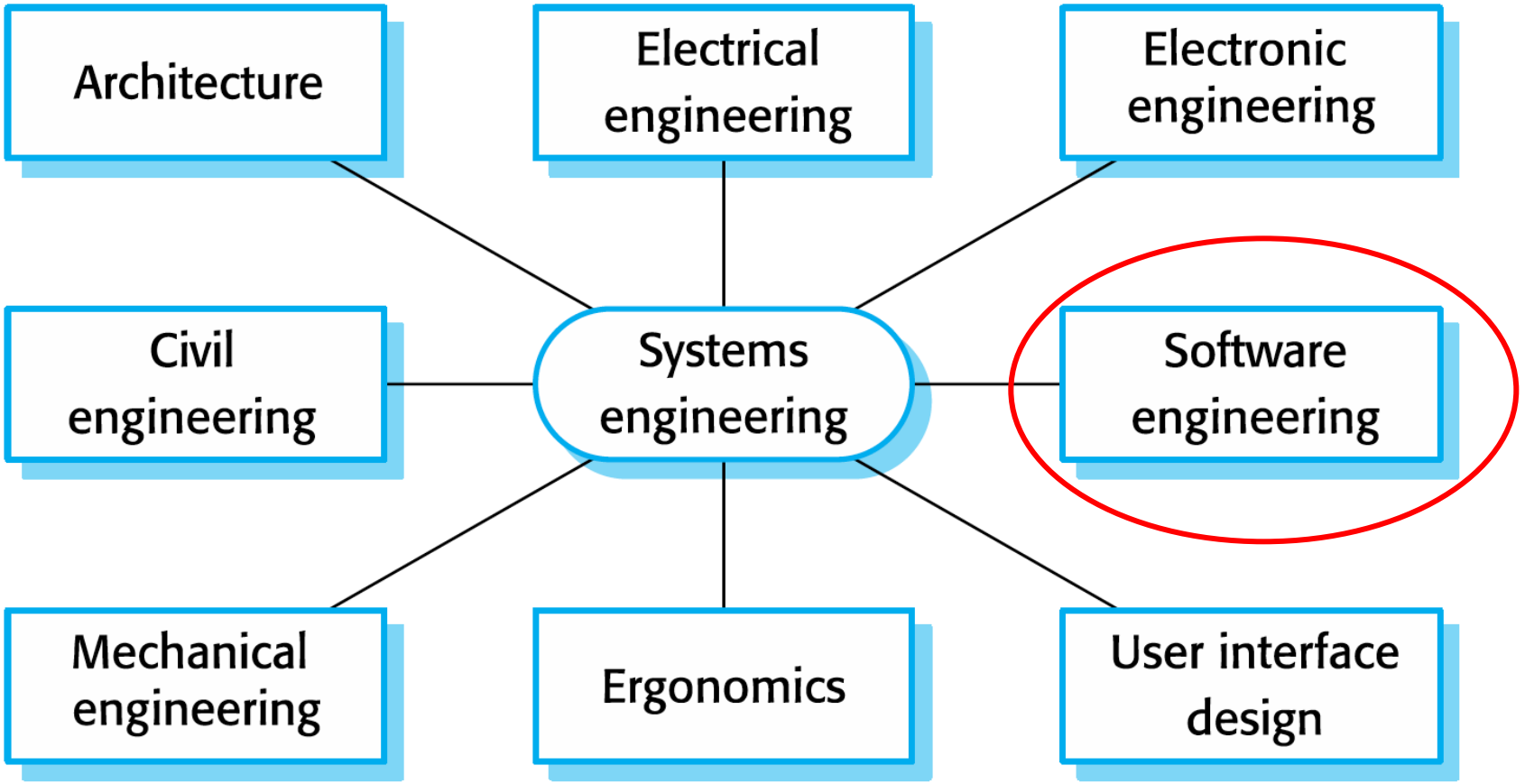
SOFTWARE SYSTEMS

- Software engineering is not an isolated activity but is part of a broader systems engineering process.
- Software systems are therefore not isolated systems but are essential components of broader systems that have a human, social or organizational purpose

TYPES OF SYSTEMS

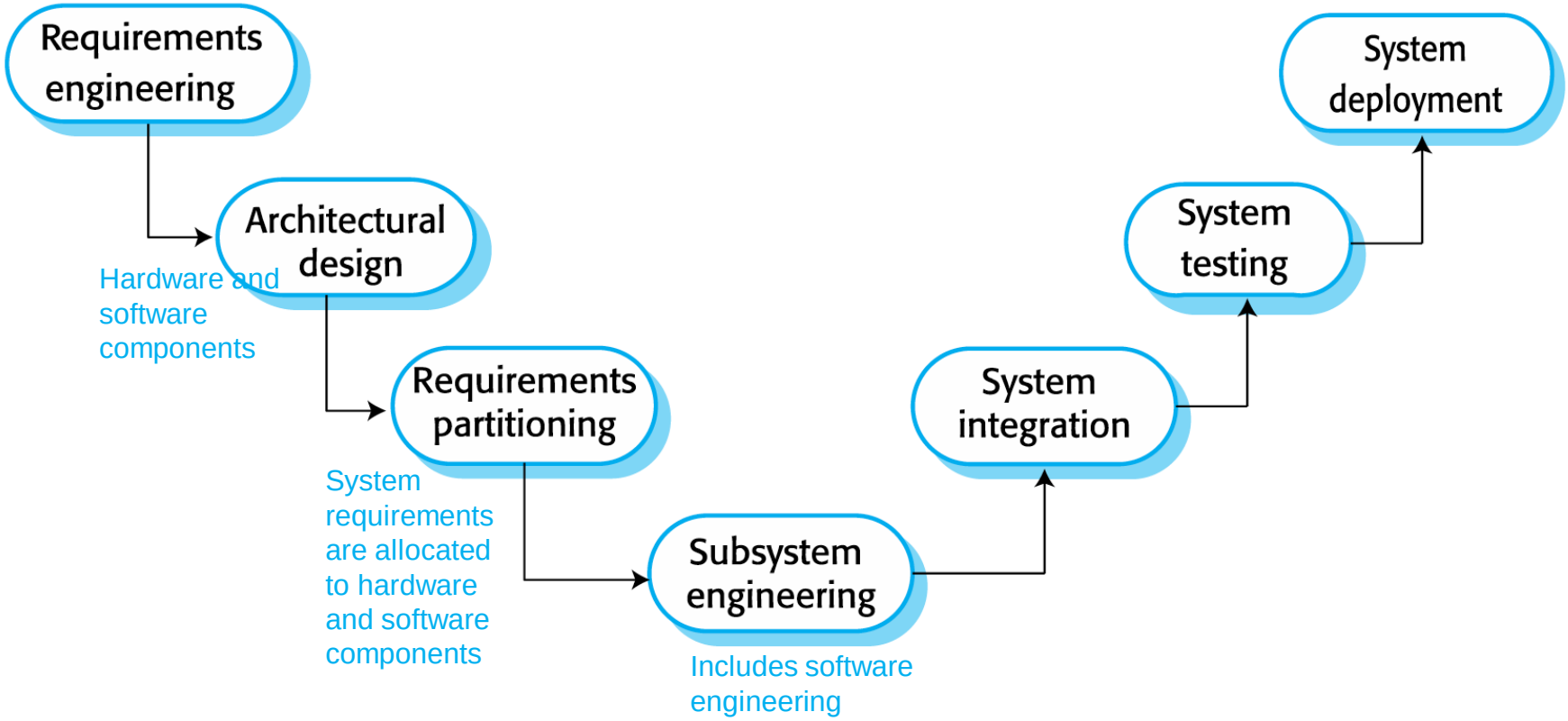
- Technical computer-based systems
  - Include hardware and software but not humans or organizational processes.
  - Off the shelf applications, control systems, etc.
- Sociotechnical systems
  - Include technical systems plus people who use and manage these systems and the organizations that own the systems and set policies for their use.
  - Business systems, command and control systems, etc.

PROFESSIONAL DISCIPLINES INVOLVED IN SYSTEMS ENGINEERING



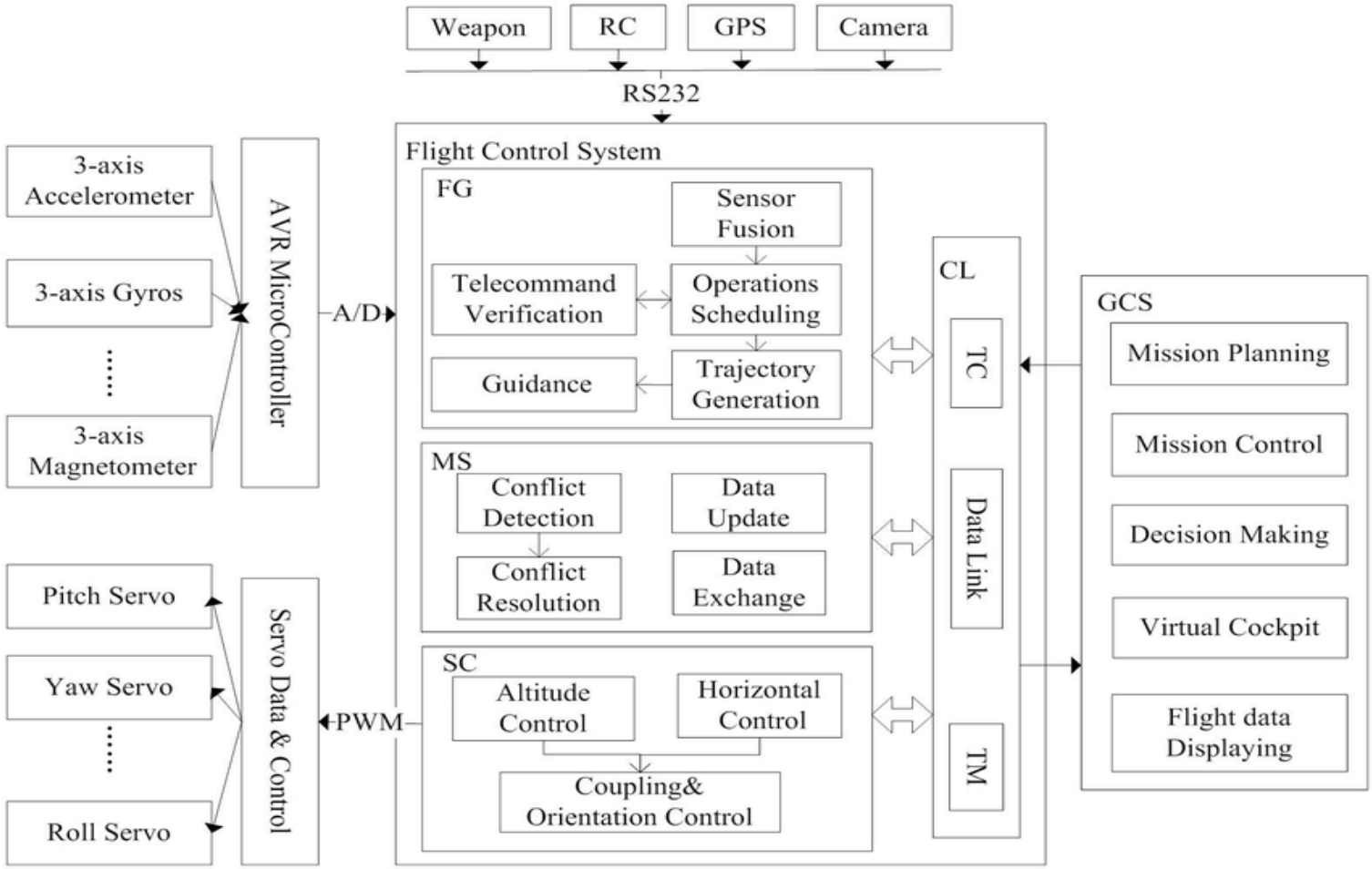
MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

SYSTEMS DEVELOPMENT – PROCESS "V MODEL"



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

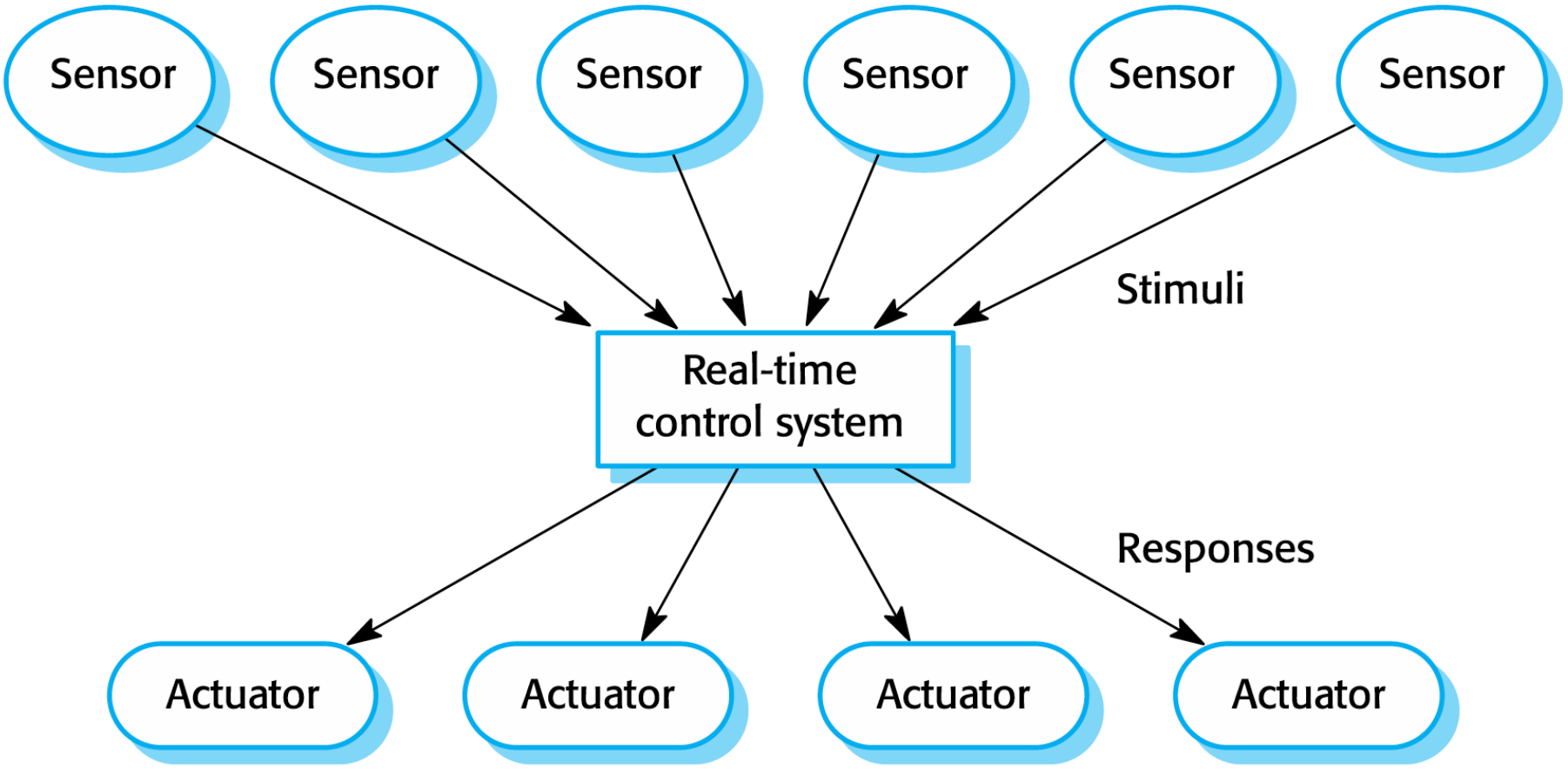
EXAMPLE – (PART OF) HARDWARE AND SOFTWARE ARCHITECTURE OF THE FLIGHT CONTROL SYSTEM



From: https://www.researchgate.net/figure/311449631_fig1_Fig-1-Part-of-hardware-and-software-architecture-of-the-flight-control-system-The

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

A GENERIC REAL TIME CONTROL SYSTEM



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

ARCHITECTURAL PATTERNS FOR EMBEDDED SYSTEMS

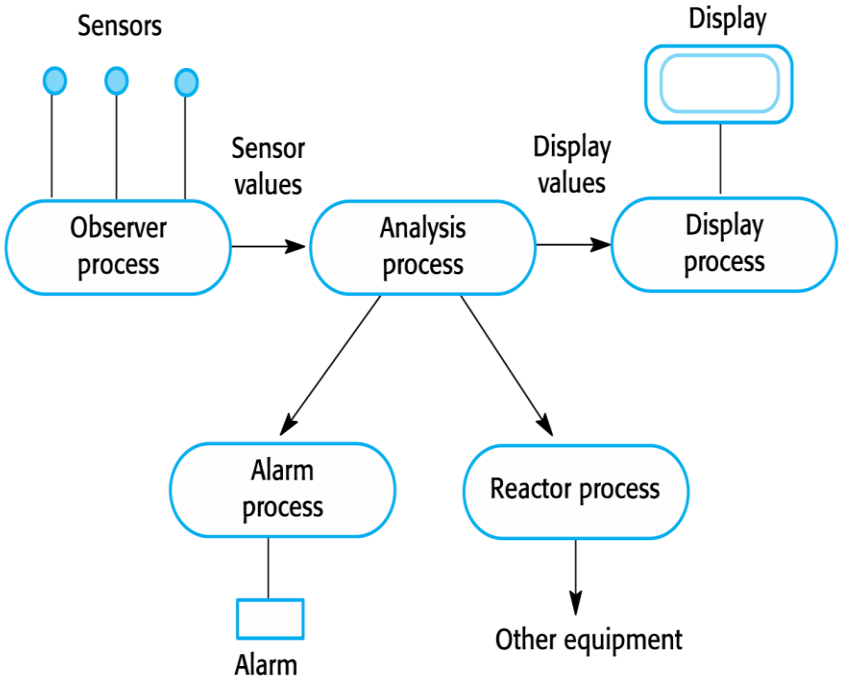
- Observe and React
  - This pattern is used when a set of sensors are routinely monitored and displayed.
- Environmental Control
  - This pattern is used when a system includes sensors, which provide information about the environment and actuators that can change the environment
- Process Pipeline
  - This pattern is used when data has to be transformed from one representation to another before it can be processed

OBSERVE AND REACT DESCRIPTION

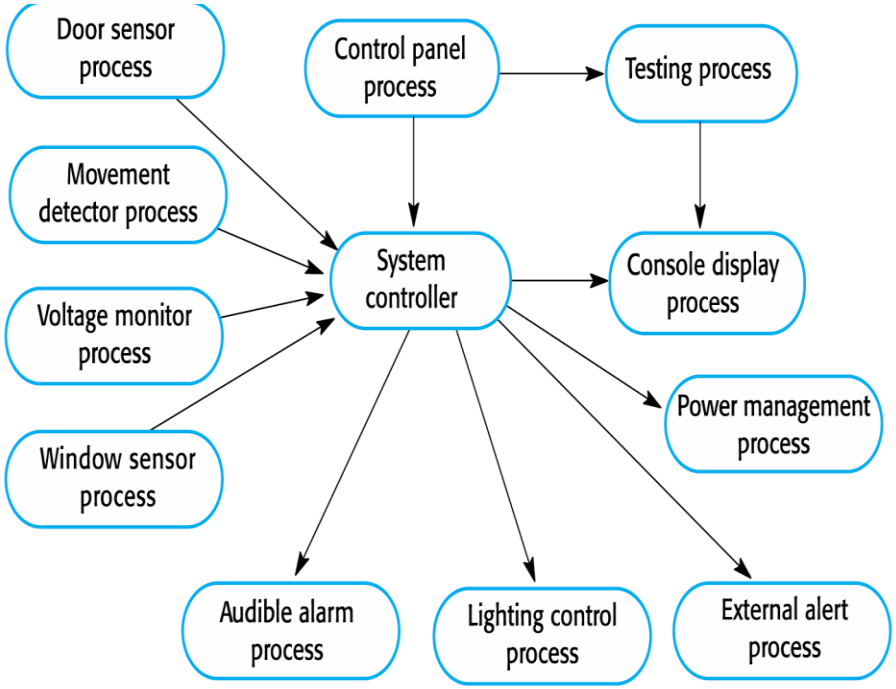
- Description
  - The input values of a set of sensors of the same types are collected and analyzed.
  - These values are displayed in some way.
  - If the sensor values indicate that some exceptional condition has arisen, then actions are initiated to draw the operator's attention to that value and, in certain cases, to take actions in response to the exceptional value
- Stimuli
  - Values from sensors attached to the system
- Responses
  - Outputs to display, alarm triggers, signals to reacting systems
- Processes
  - Observer, Analysis, Display, Alarm, Reactor
- Used in
  - Monitoring systems, alarm systems

OBSERVE AND REACT ARCHITECTURE

Generic pattern



Pattern instance - burglar alarm system

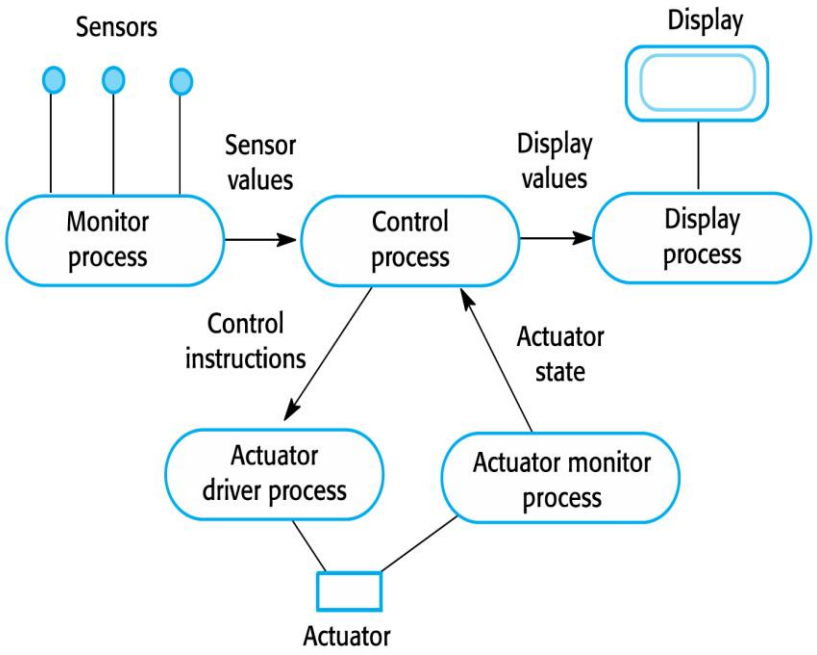


ENVIRONMENTAL CONTROL DESCRIPTION

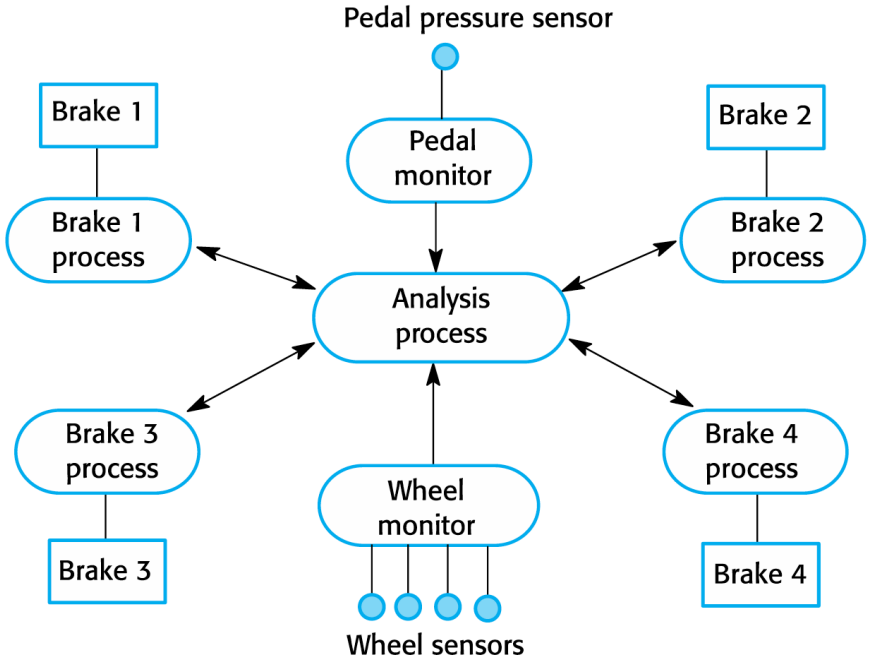
- Description
  - The system analyzes information from a set of sensors that collect data from the system's environment.
  - Further information may also be collected on the state of the actuators that are connected to the system.
  - Based on the data from the sensors and actuators, control signals are sent to the actuators that then cause changes to the system's environment.
  - Information about the sensor values and the state of the actuators may be displayed
- Stimuli
  - Values from sensors attached to the system and the state of the system actuators
- Responses
  - Control signals to actuators, display information
- Processes
  - Monitor, Control, Display, Actuator Driver, Actuator monitor
- Used in
  - Control systems

ENVIRONMENTAL CONTROL ARCHITECTURE

Generic pattern



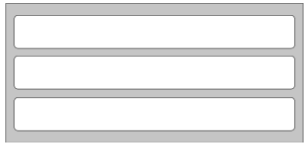
Pattern instance - anti-skid braking system



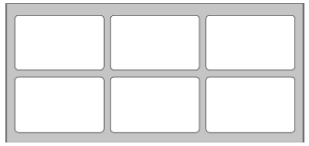
MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

From Mark Richards and Neal Ford

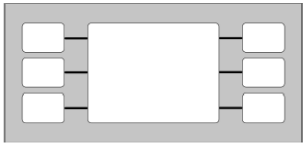
A “VISUAL” SUMMARY OF ARCHITECTURE STYLES



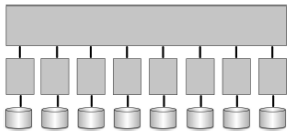
layered architecture



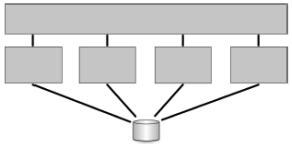
modular monolith



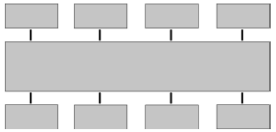
microkernel architecture



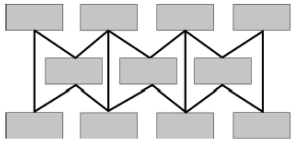
microservices architecture



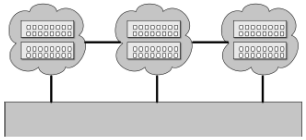
service-based architecture



service-oriented architecture



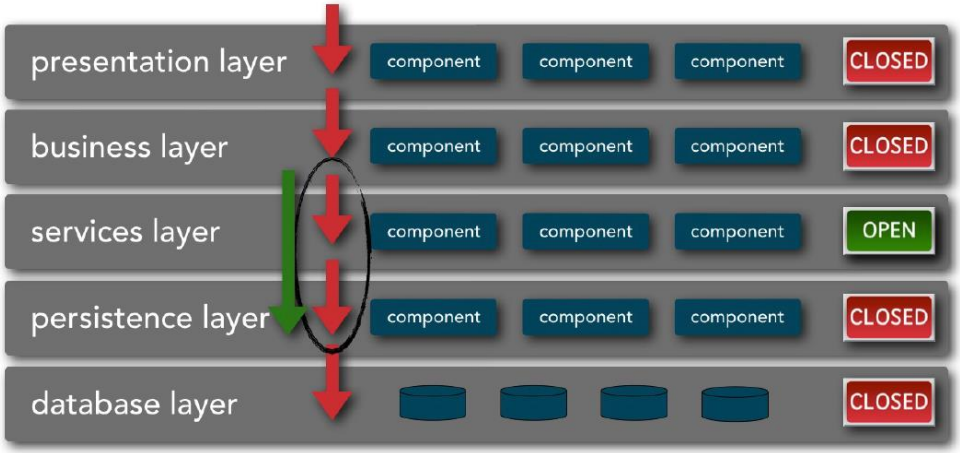
event-driven architecture



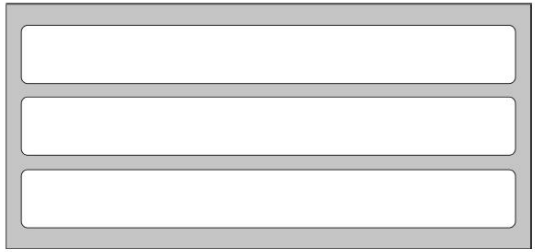
space-based architecture

MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

layered architecture

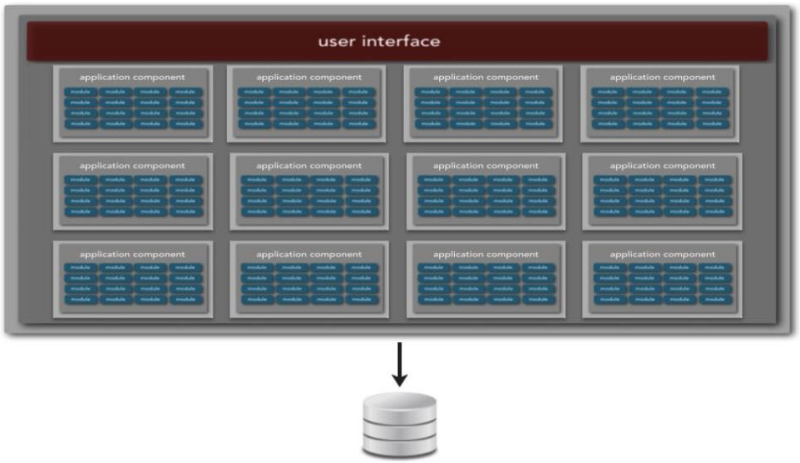


single deployment with functionality grouped by technical categories

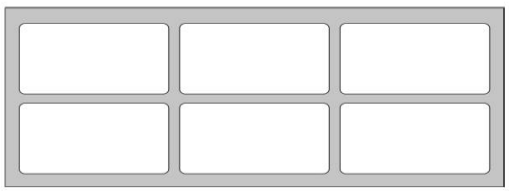


MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

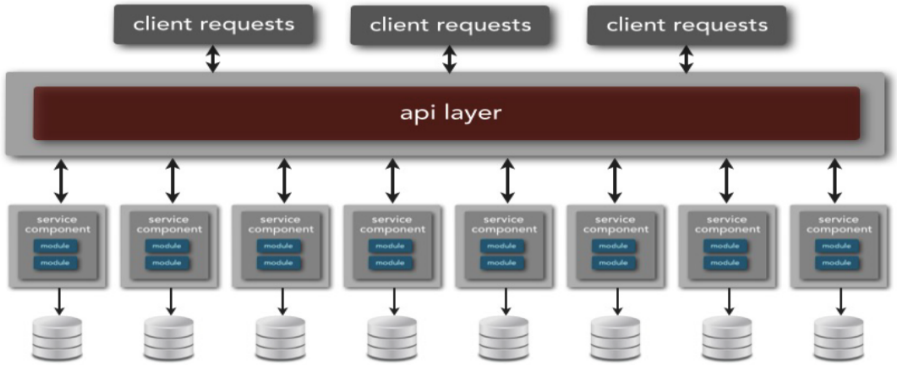
modular monolith



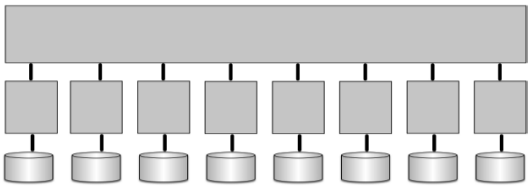
single deployment with functionality grouped by domain area



microservices architecture

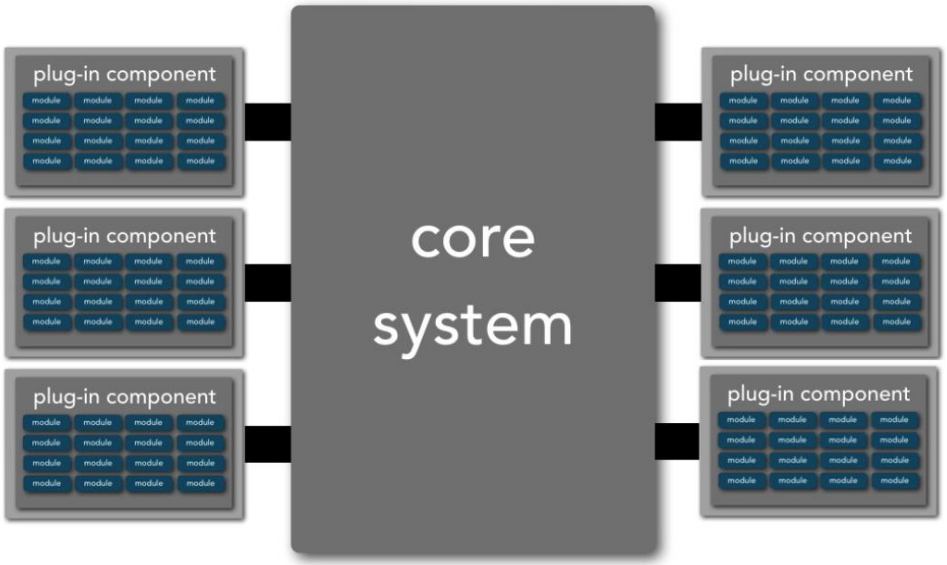


single purpose functions deployed as separate units

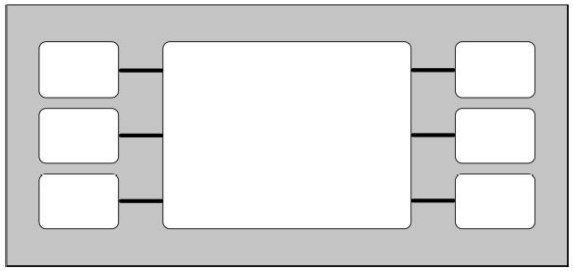


MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

microkernel architecture

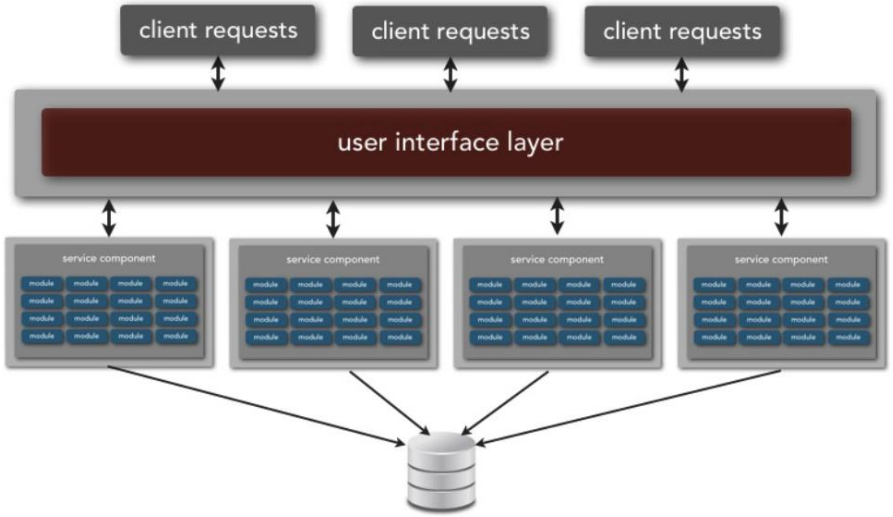


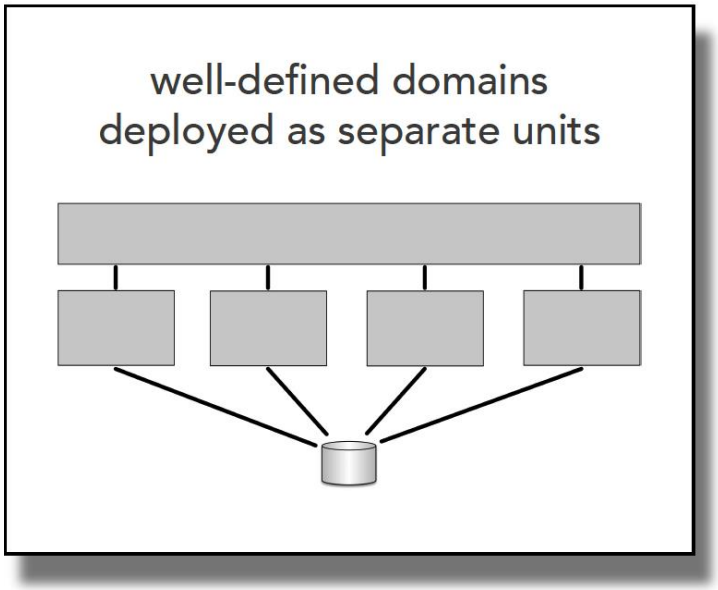
single deployment with
modular, independent add-on
functionality



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

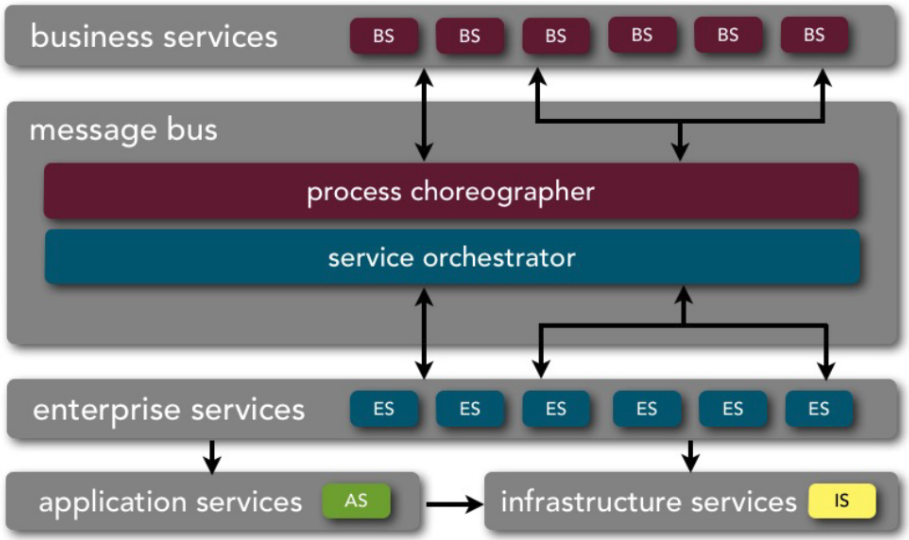
service-based architecture



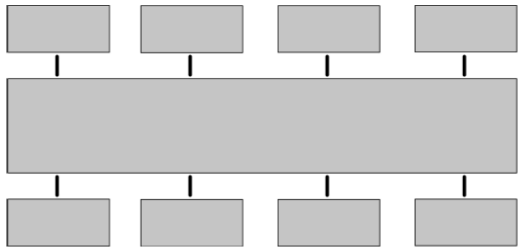


MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

service-oriented architecture

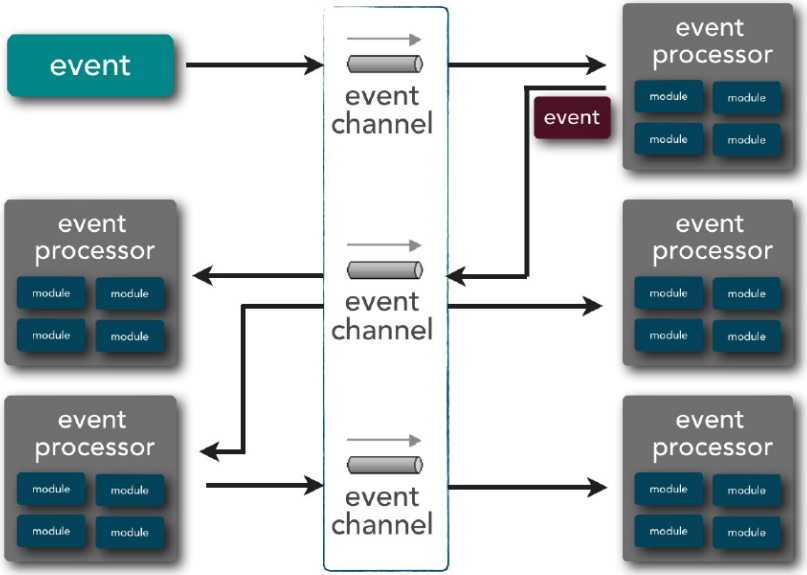


integrated communication between heterogeneous enterprise systems

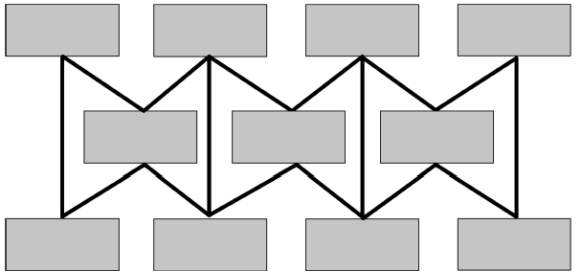


MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

event-driven architecture

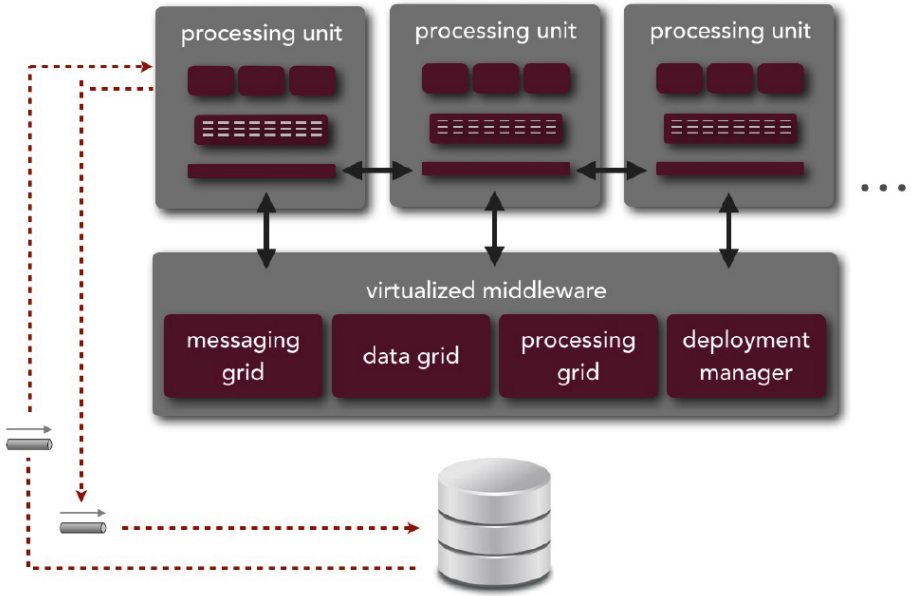


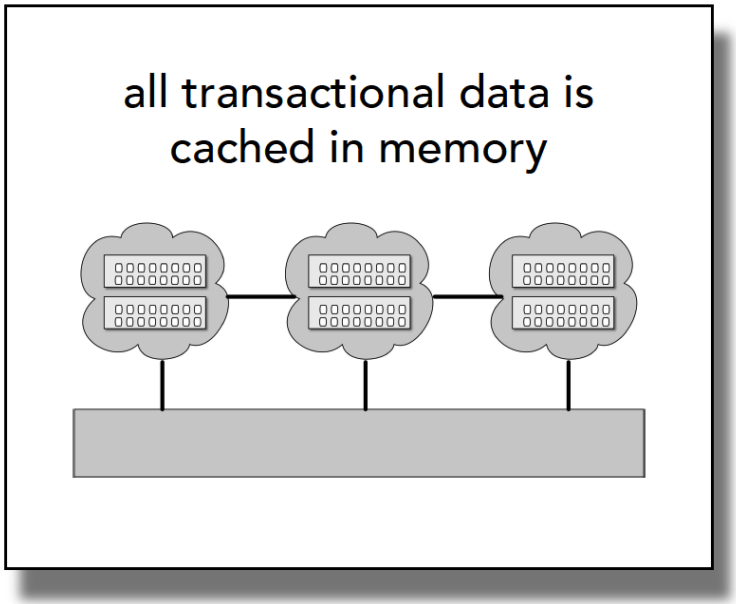
reacts to events that happen in the system



MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND

space-based architecture





MARYLAND APPLIED GRADUATE ENGINEERING
THE A. JAMES CLARK SCHOOL of ENGINEERING
UNIVERSITY OF MARYLAND