Software Engineering

Software Architecting

Elegant tools for a more civilized age.

Dr. Hale
University of Nebraska at Omaha
Secure Web Application Development – Lecture 4

Continued from last time.

Part 3: Software Architectures

- Architectures 101
- Architectural patterns
- Benefits
- Examples
- Adding Security Components
- Asking the right questions

Hands-on

- Build an architecture diagram of your app idea
- What security components should be added?

Part 3: Software architecture

Software Architecture: Origins

Software Engineers have always employed software architectures
Very often without realizing it!

A "field" was needed to address issues identified by researchers and practitioners

- Essential software engineering difficulties
- Need for software reuse

Christopher Alexander – Architect and professor

Notes on the Synthesis of Form

- required reading for researchers in computer science throughout the 1960s.
- Influenced programming language design, modular programming, object-oriented programming, software engineering and other design methodologies

Edsger Dijkstra

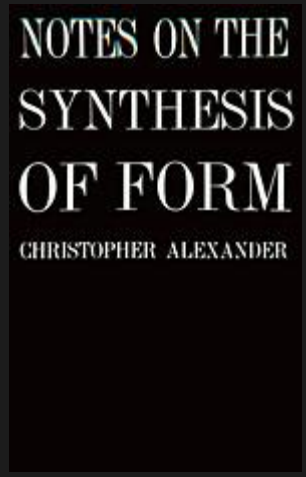
A Discipline of Programming

- Emphasized mathematical concepts, orientation, and system structure

Mary Shaw and David Garlan

Software Architecture: Perspectives on an Emerging Discipline

- Codified standard terms like component, connector, and design patterns

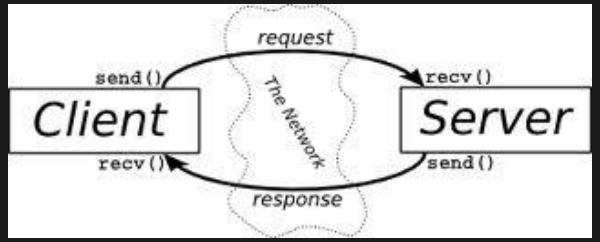


Software Architecting

- An architecture-centric approach to software engineering places an emphasis on design
- Design pervades the engineering activity from the very beginning

- This occurs in three phases (usually iterative)
  - Analysis
    - Determining Requirements from the customer
    - Analyzing requirements to determine what components are needed (what tools?).
  - Design (modeling)
    - Designing and diagraming components and there interactions.
    - Refining the design to improve efficiency, usefulness, security, etc.
  - Implementation
    - Coding components and integrating them together
    - Testing to make sure the implementation meets the design

What is an Architecture?



A software architecture defines four Cs

- Components: computational elements
  - clients, servers, databases, filters, layers
- Connections: interactions among components
  - procedure call, shared variable, protocols, events
- Configuration: patterns that guide composition of system elements
  - throughput, capacity, performance, scalability
- Control Flow: how information passes through the elements
  - component capability, consistency with requirements

Ex. Client-server architecture

- Should be familiar, server provides a service, client requests and uses the service
- Client browser (at least) provides the UI
- Server provides the application and database

Components

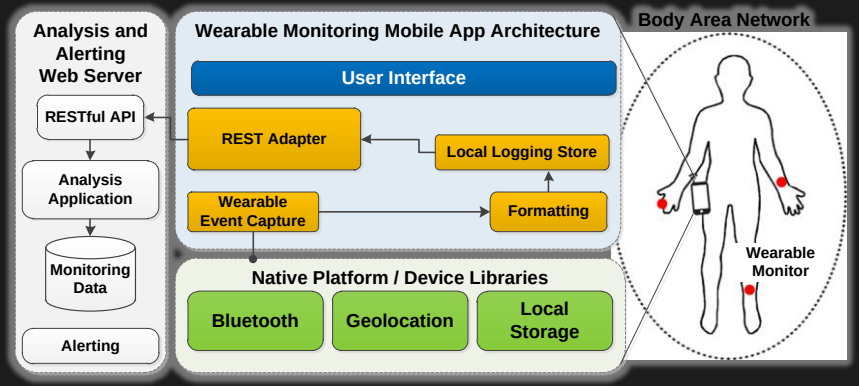
Elements that encapsulate processing and data in a system's architecture are referred to as software components

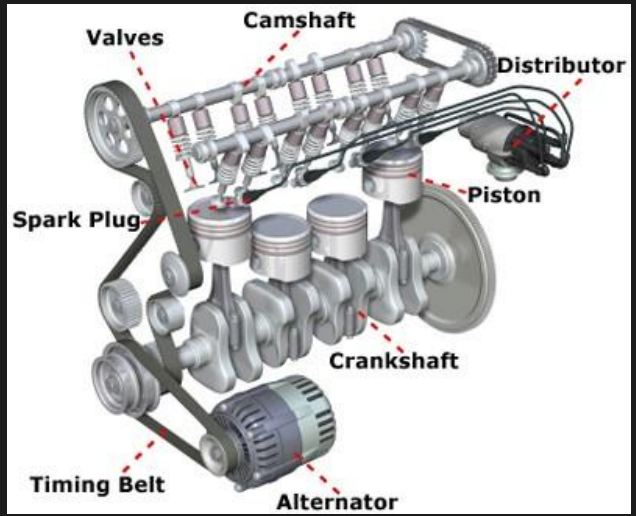
Definition

A *software component* is an architectural entity that

- encapsulates a subset of the system's functionality and/or data
- restricts access to that subset via an explicitly defined interface
- has explicitly defined dependencies on its required execution context

Components are separable chunks much like car parts





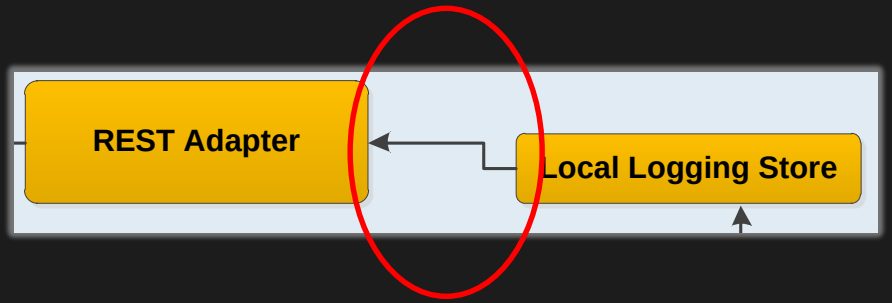
Connectors

In complex systems interaction may become more important and challenging than the functionality of the individual components

Definition

A *software connector* is an architectural building block tasked with effecting and regulating interactions among components. They define *interfaces*.

Connectors allow concerns to be separated into chunks

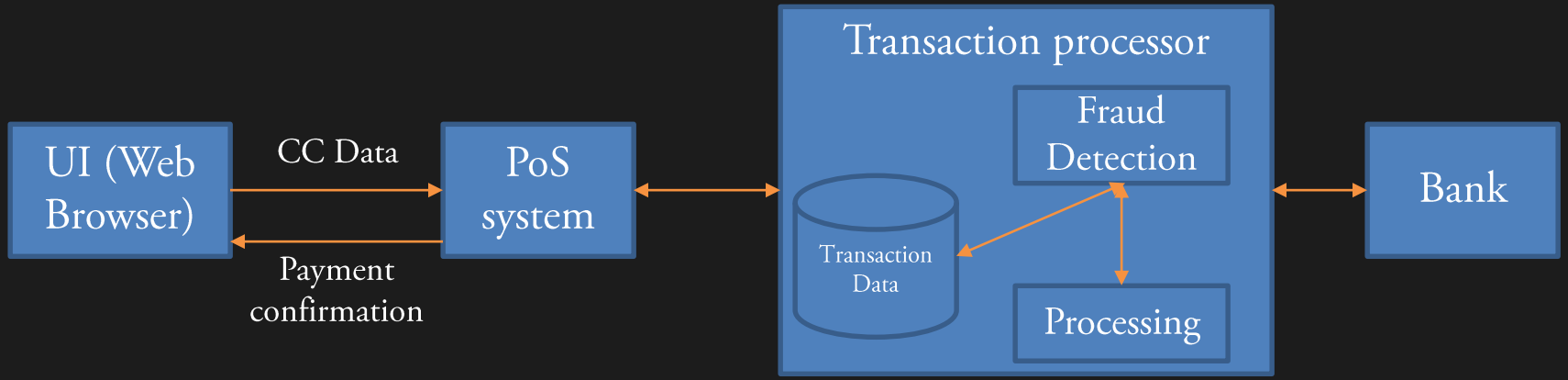


Configuration: Architectural models

Components and connectors are composed in a specific way in a given system's architecture to accomplish that system's objective

Definition

An *architectural configuration*, or topology, is a set of specific associations between the components and connectors of a software system's architecture. It can be high or low level.



a high level architecture

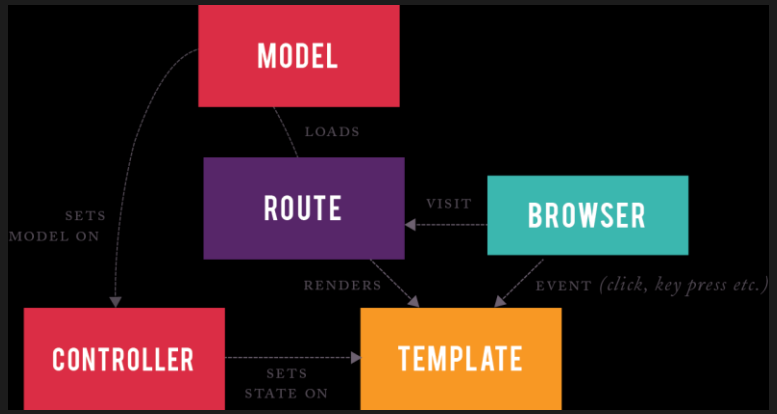
Control Flow

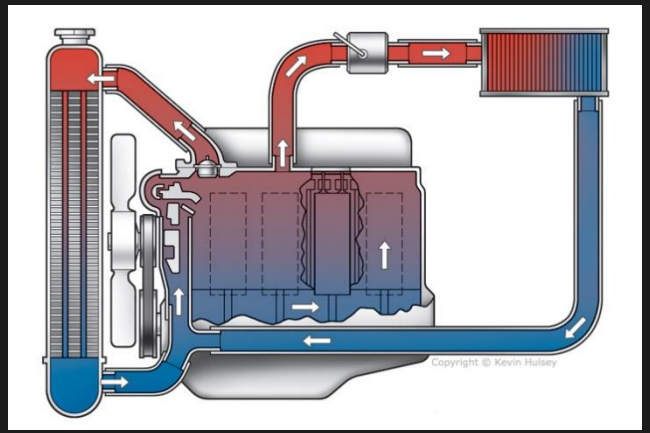
Control flow dictates how execution proceeds logically

Definition

A *control flow*, is a logical progression between components and across connectors that characterizes how an application works.

Although not always present on an architecture, control flows can be very beneficial for descriptive purposes, especially with abstract architectures





Another great tool for thinking about what a product should do

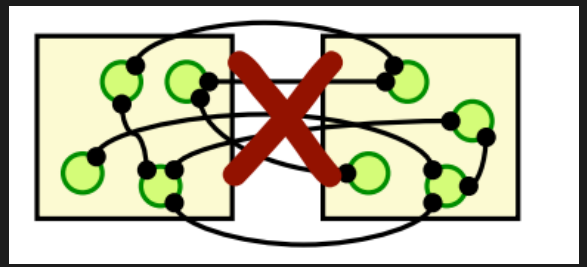
You should ALWAYS have a high level architecture diagram before you create anything.
It represents your single or collective vision.

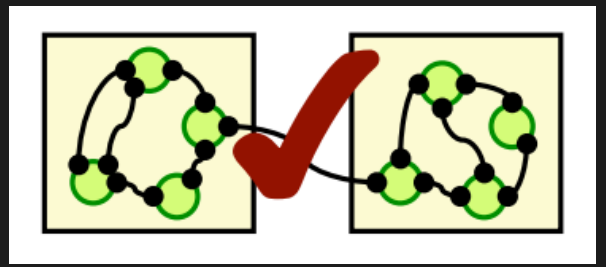
Coupling vs Cohesion

Building blocks (component AKA module)

Good architectures:

- Minimize coupling
  - Goal: Modules don't need to know much about each other to interact
  - Improves: resiliency and extensibility
- Maximize cohesion
  - Goal: the contents of a single module are strongly inter-related
  - Improves: separation of concerns, extensibility, and reuse

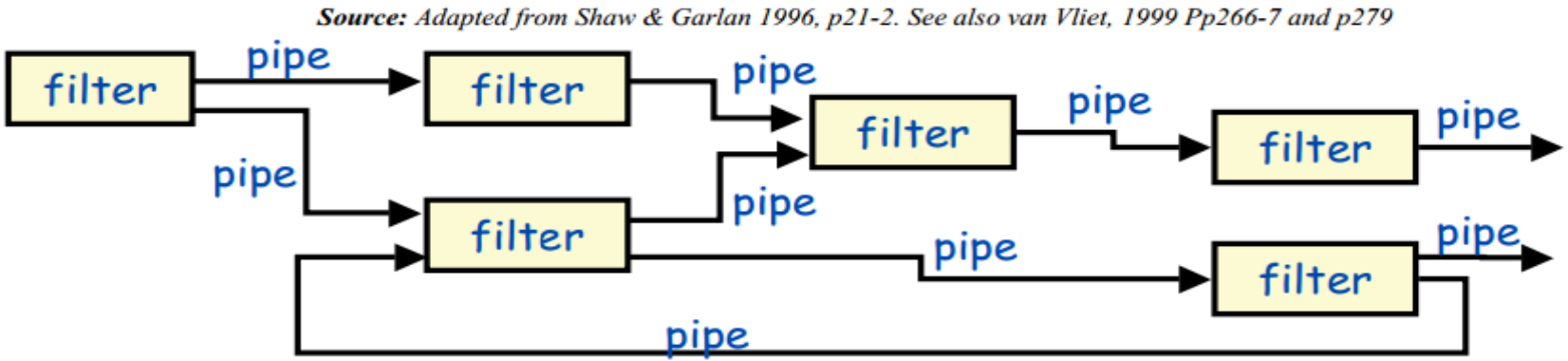




Think about (and re-use) design patterns.

Here are a few

Pipe and Filter



Examples:

- Unix shell commands
- Compilers

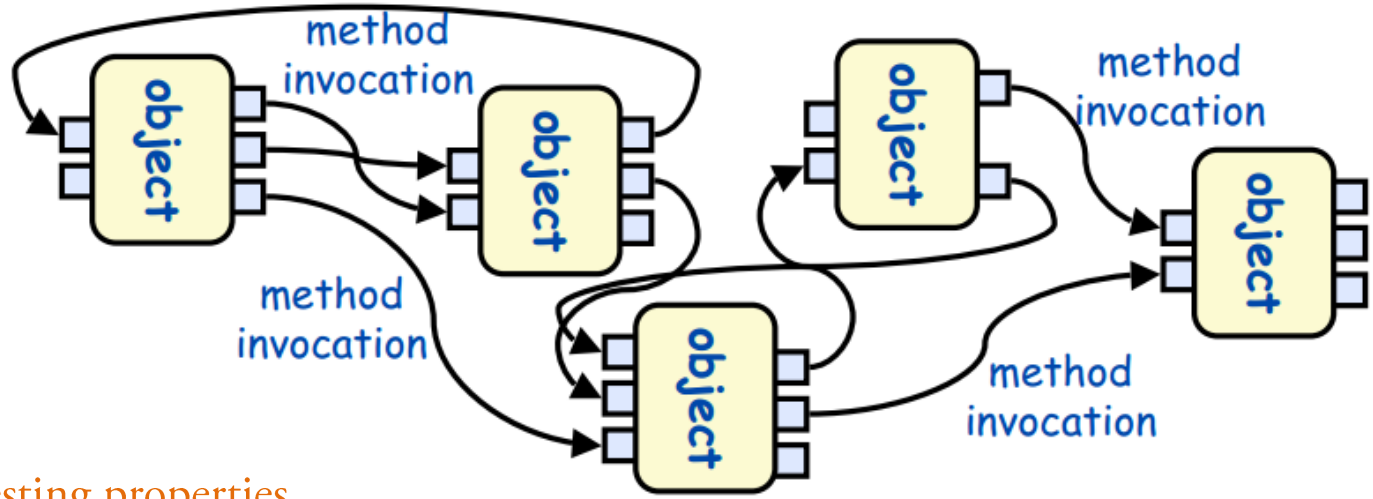
  Lexical analysis -> parsing -> semantic analysis -> code generation
- Signal processing

Interesting properties:

- Filters don't need to know anything about what they are connected to
- Can be parallelized
- System behavior is defined by the composition and behavior of filters

Object Oriented

Source: *Adapted from Shaw & Garlan 1996, p22-3.*



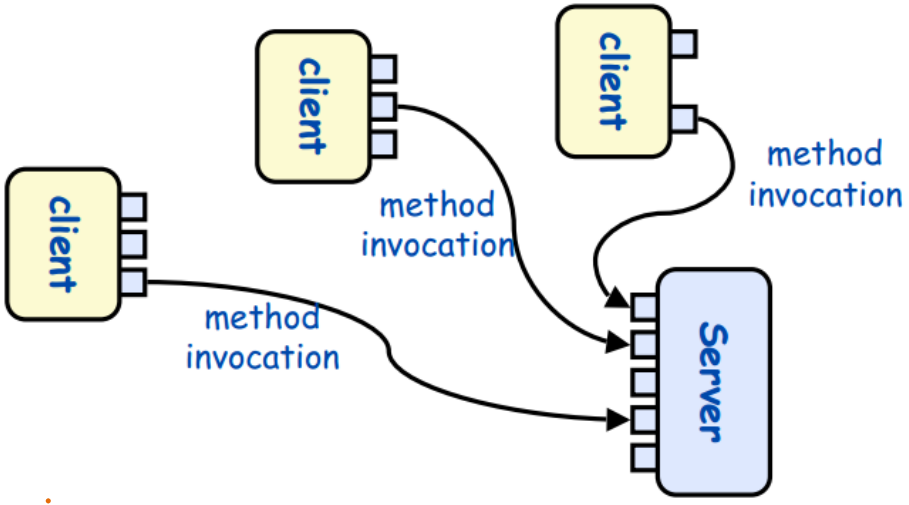
Interesting properties

- Data hiding (e.g. private, public, protected)
- Can be multi-threaded or single threaded
- Can decompose problems into sets of interacting agents

Disadvantages

Objects must know the identity of other objects they need to interact with

Object Oriented: Variant 1 – Client Server



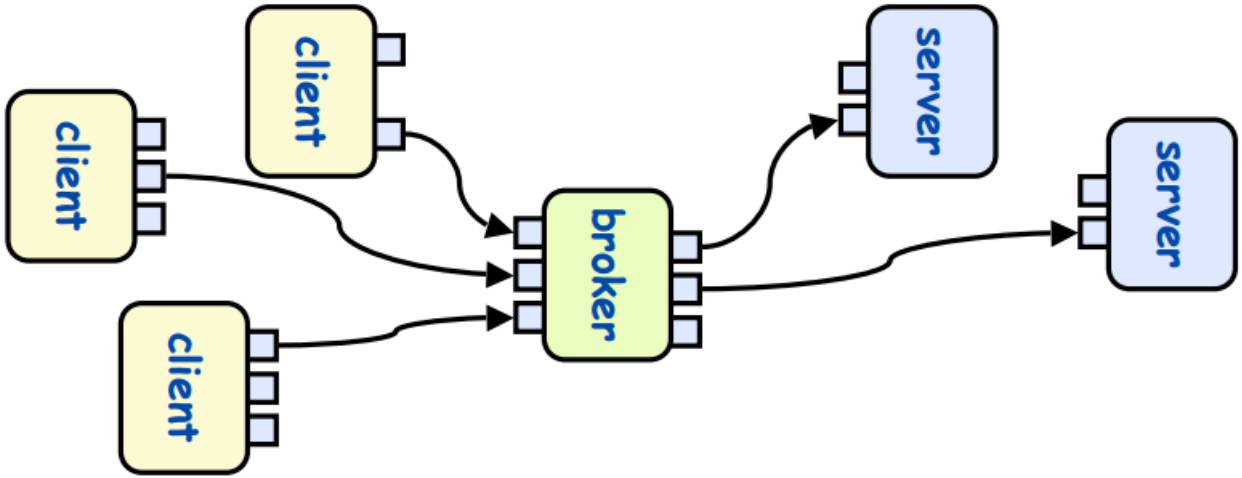
Interesting properties

- Object oriented properties
- Clients do not need to know about one another

Disadvantages

Objects must know the identity of other objects they need to interact with

Object Oriented: Variant 2 – Object Brokers



Example:

Load balancing, cloud services

Interesting properties:

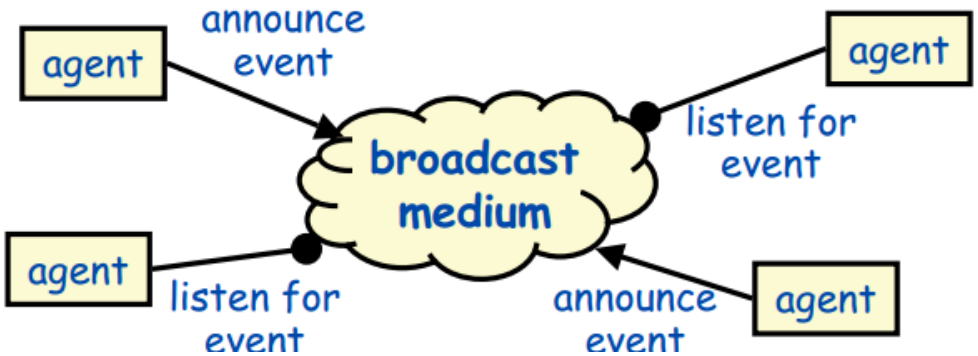
- Clients no longer need to know which server they are using
- Can have many servers
- Clients can move between servers (e.g. mobile location change)

Disadvantages

- Broker can be a bottleneck
- Degraded performance (e.g. response time)

Event Based

Source: *Adapted from Shaw & Garlan 1996, p23-4. See also van Vliet, 1999 Pp264-5 and p278*



Example:

- Debugging systems (listen for breakpoints)
- Dbms (data integrity checking)
- GUIs – listen for on-click or on-key events and handle them somehow

Interesting properties:

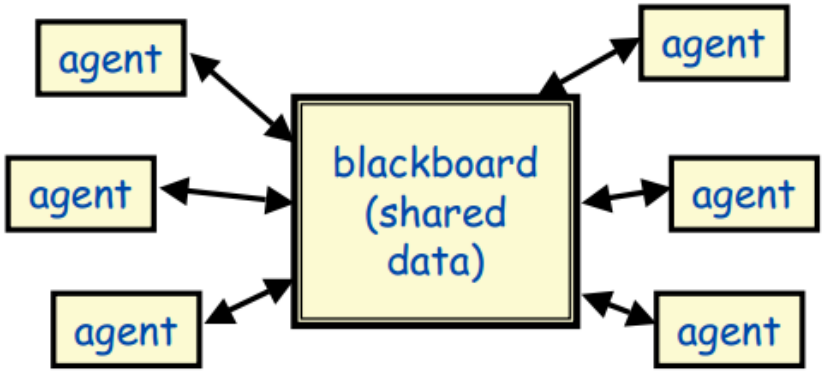
- Announcers don't need to know who will handle the event
- Ease of re-use and emergent evolution (add new agents)
- Asynchronous

Disadvantages

- Components have little-to-no control over temporal execution flows
- Asynchronous systems can be harder to code

Repositories

Source: *Adapted from Shaw & Garlan 1996, p26-7. See also van Vliet, 1999, p280*



Example:

- Databases
- Blackboard expert systems
- MVC frameworks

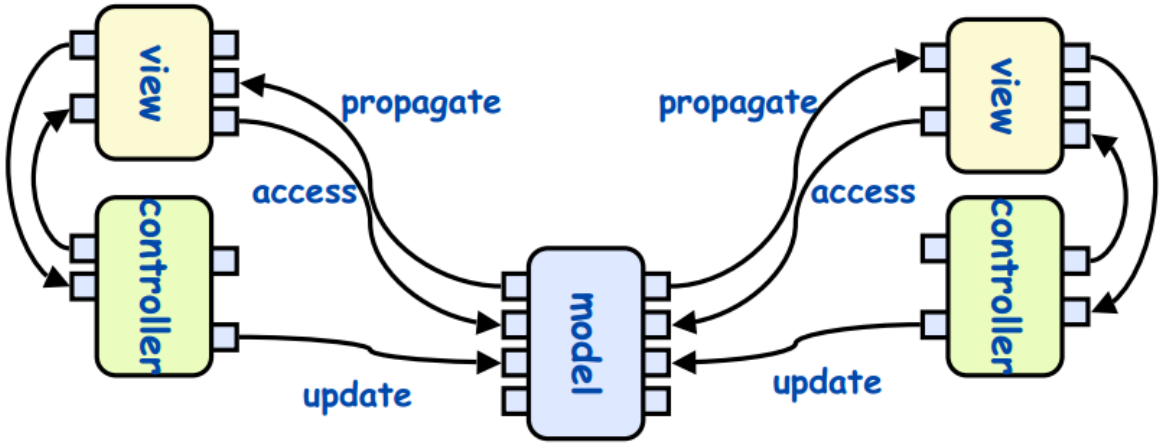
Interesting properties:

- Can choose where control is (agents, blackboard, both)
- Reduces the need to duplicate complex data

Disadvantages

- Blackboard can become a bottleneck (e.g. SQLite)

Repositories: Most important variant



Interesting properties:

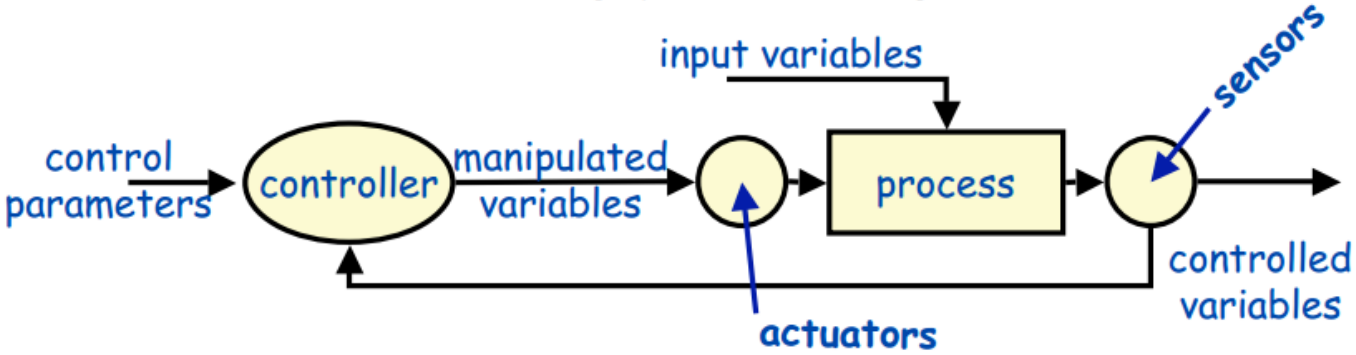
- One central model, many viewers
  Changes to the model are propagated to all viewers
- Each view has a controller
  - Handles all user updates to model
- Separation of concerns much like a 3-layer architecture
  - Business logic in the controller
  - Database logic in the model
  - UI logic in the View

Challenges

- Ensuring separation of concerns (e.g. making sure UI logic stays out of the controller)
- Static vs asynchronous loading

Process Control

Source: *Adapted from Shaw & Garlan 1996, p27-31.*



Example:

- Aircraft/spacecraft flight control systems
- Controllers for industrial production lines, power stations, etc
- Chemical engineering

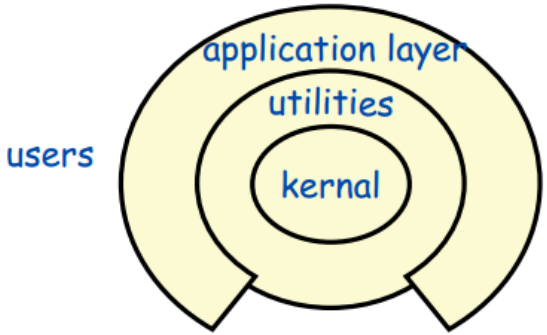
Interesting properties:

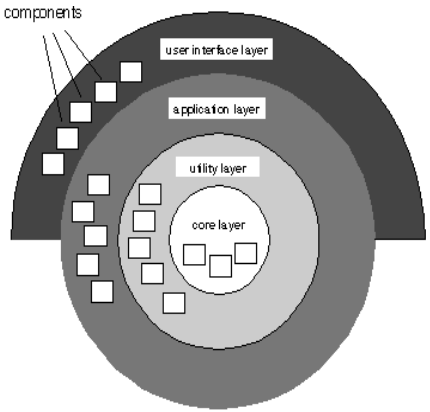
- Separates control policy from the controlled process
- Handles real-time, reactive computations

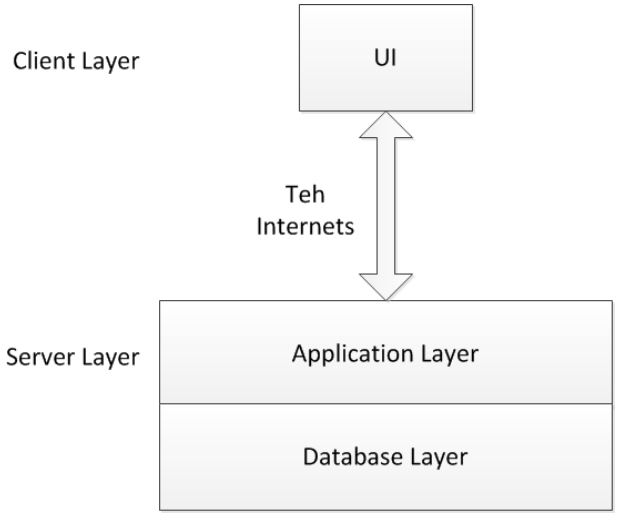
Disadvantages

- Difficult to specify timing characteristics and disturbance responses

Layered







Example:

- Operating systems, django, clouds (e.g. AWS), VMs

Interesting properties:

- Support increasing levels of abstraction during design
- Support enhancement and re-use
- Allows for separation of concerns using standardized layer interfaces

Disadvantages

- May not be able to cleanly separate layers

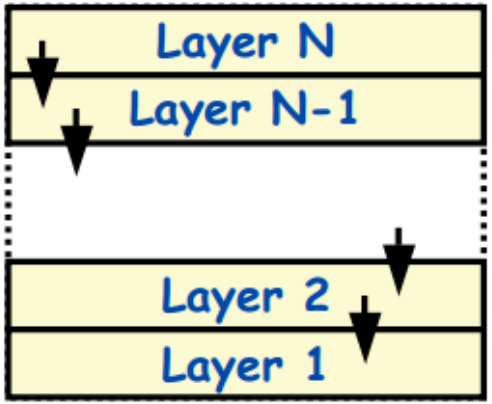
Layered: Open vs Closed

Closed architecture

Each layer only uses services of the layer below it

Advantages:

- Minimizes dependencies between layers
- Reduces the impact of layer change cascading



Open architecture

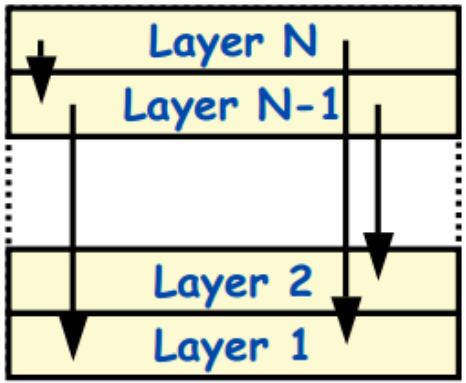
Each layer can use services from any lower layer

Advantages:

- More compact code (less middleware function calls)

Disadvantages:

- Breaks layer encapsulation increasing dependencies between layers
- makes layer change impact higher

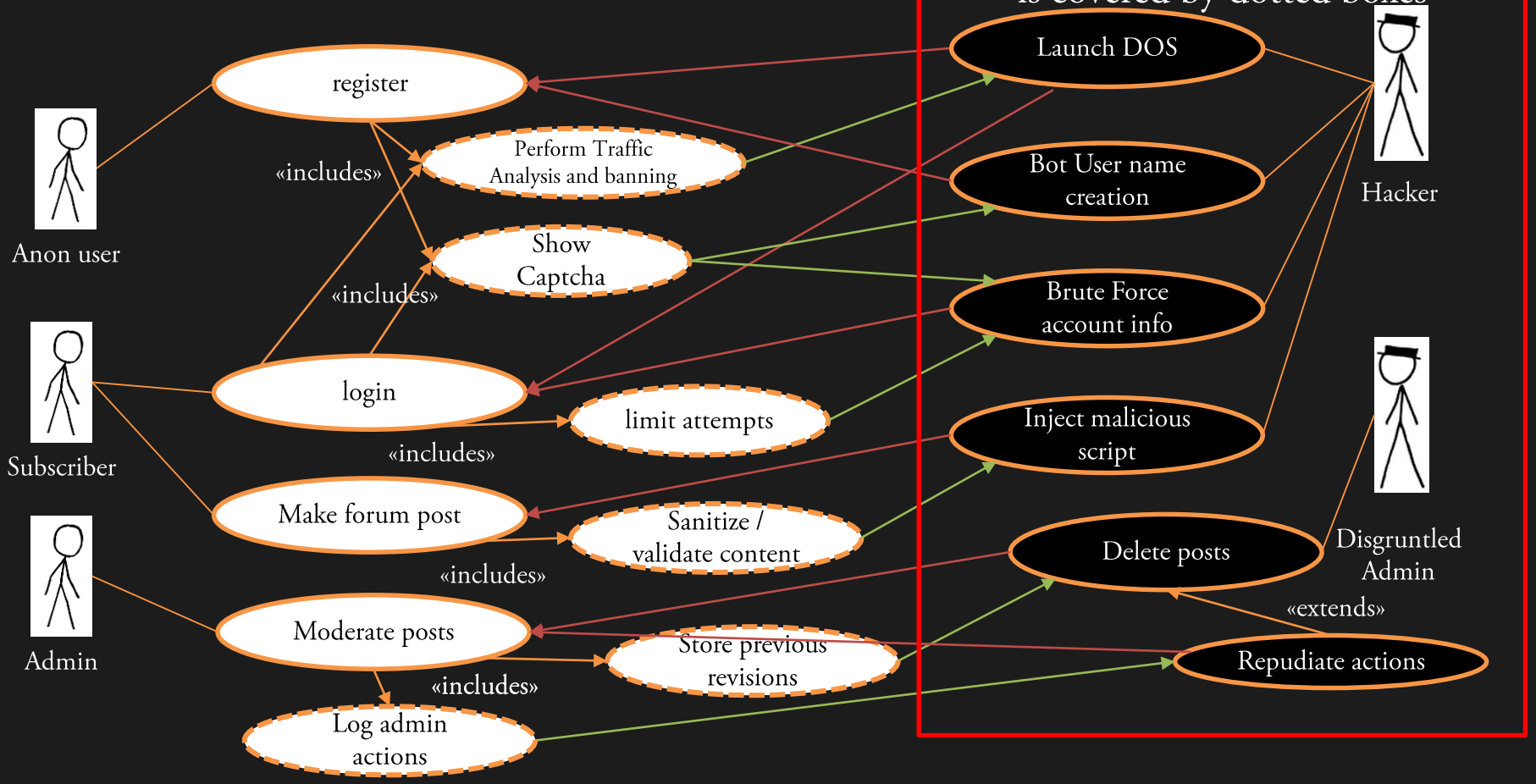


Ok... how do developers make architecture diagrams?

→ They use requirements and use cases

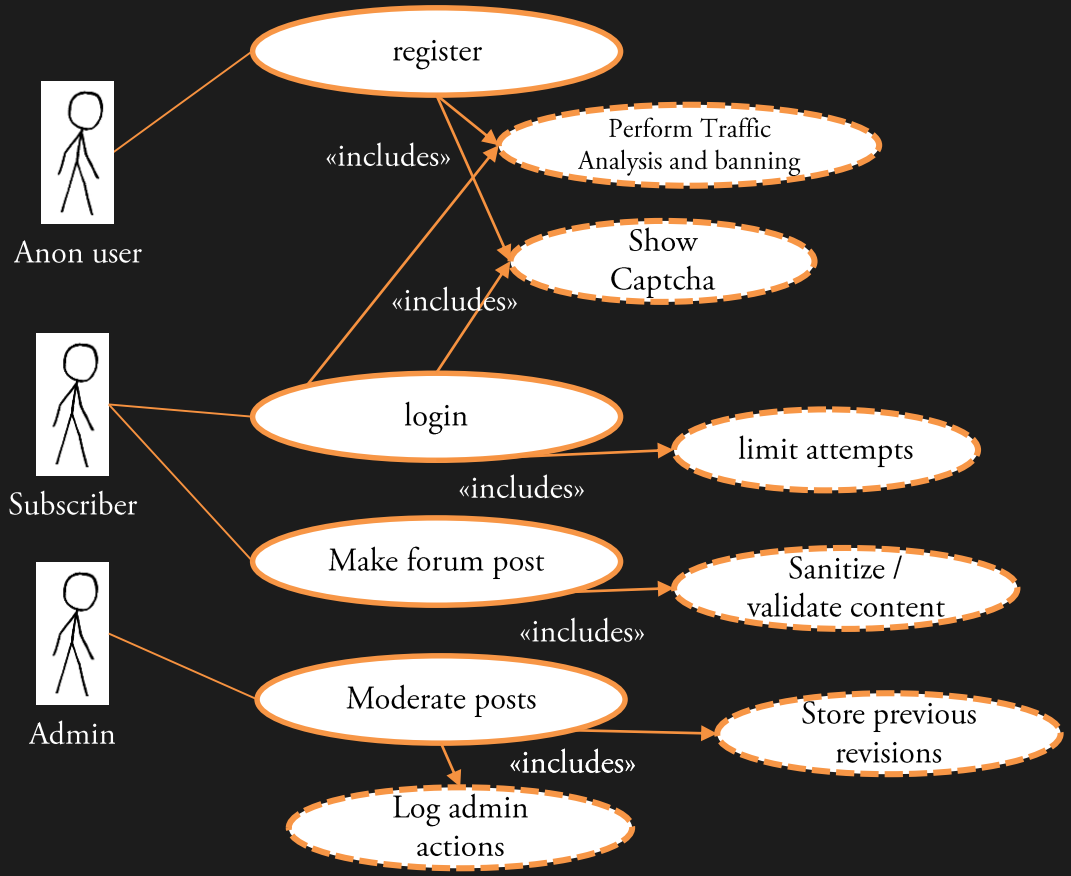
Returning to our system example (a forum)

Translate use cases to components



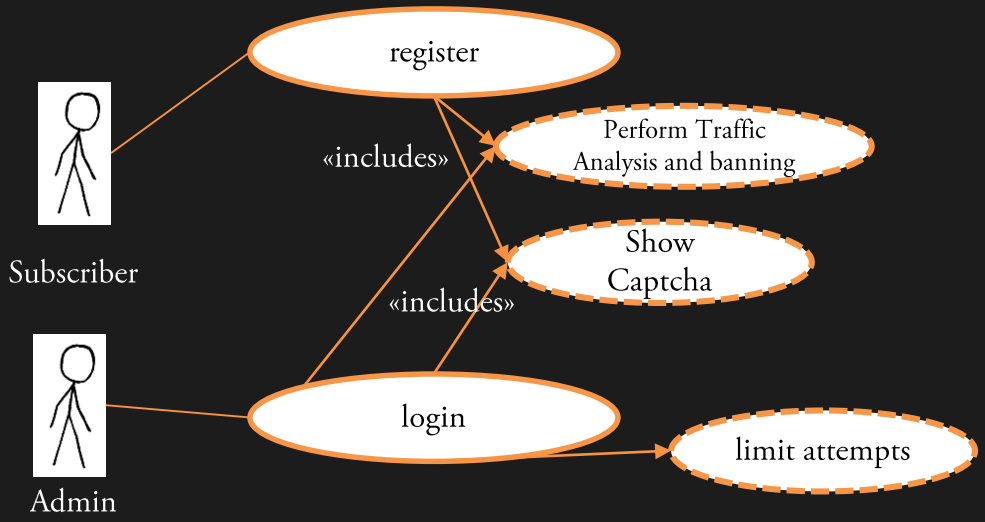
Returning to our system example (a forum)

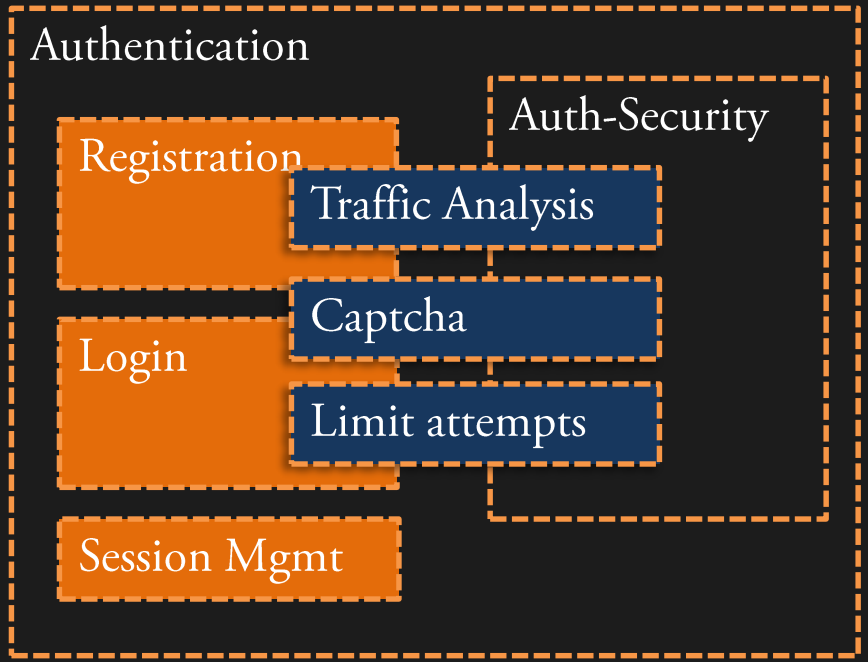
Translate use cases to components



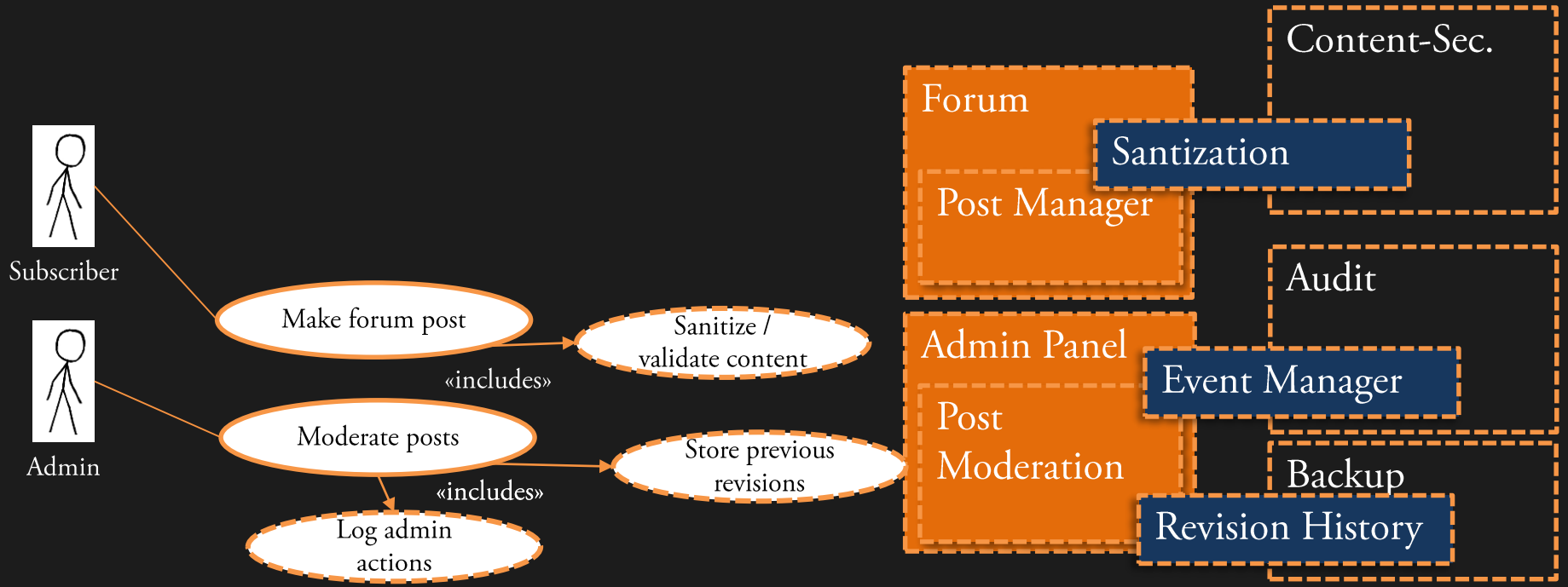
Functional security requirements are realized by building components

Mentioned this before, now it makes more sense.

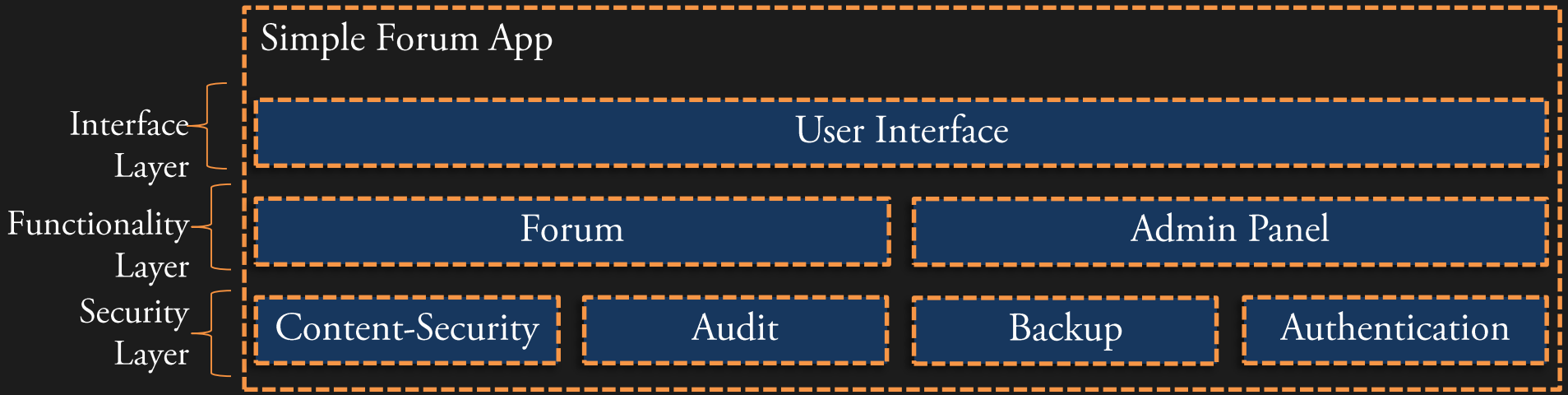




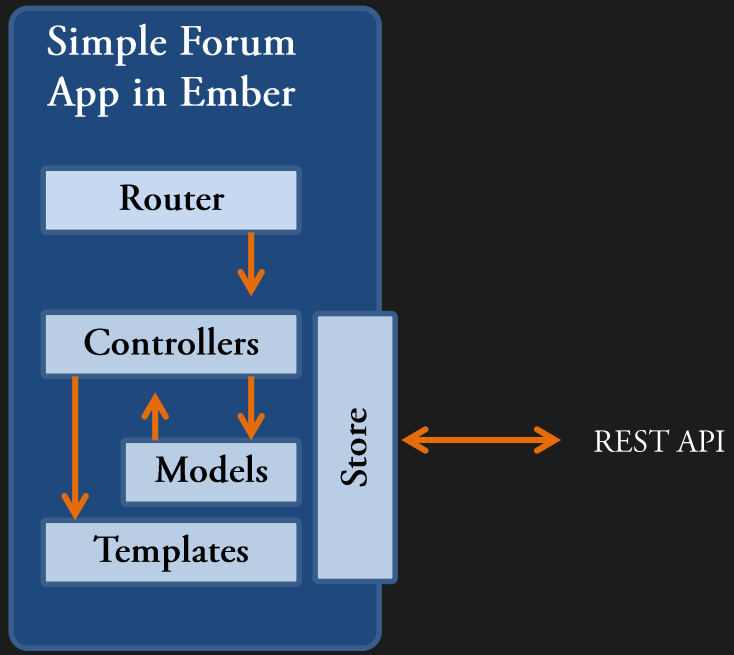
Functional security requirements are realized by building components

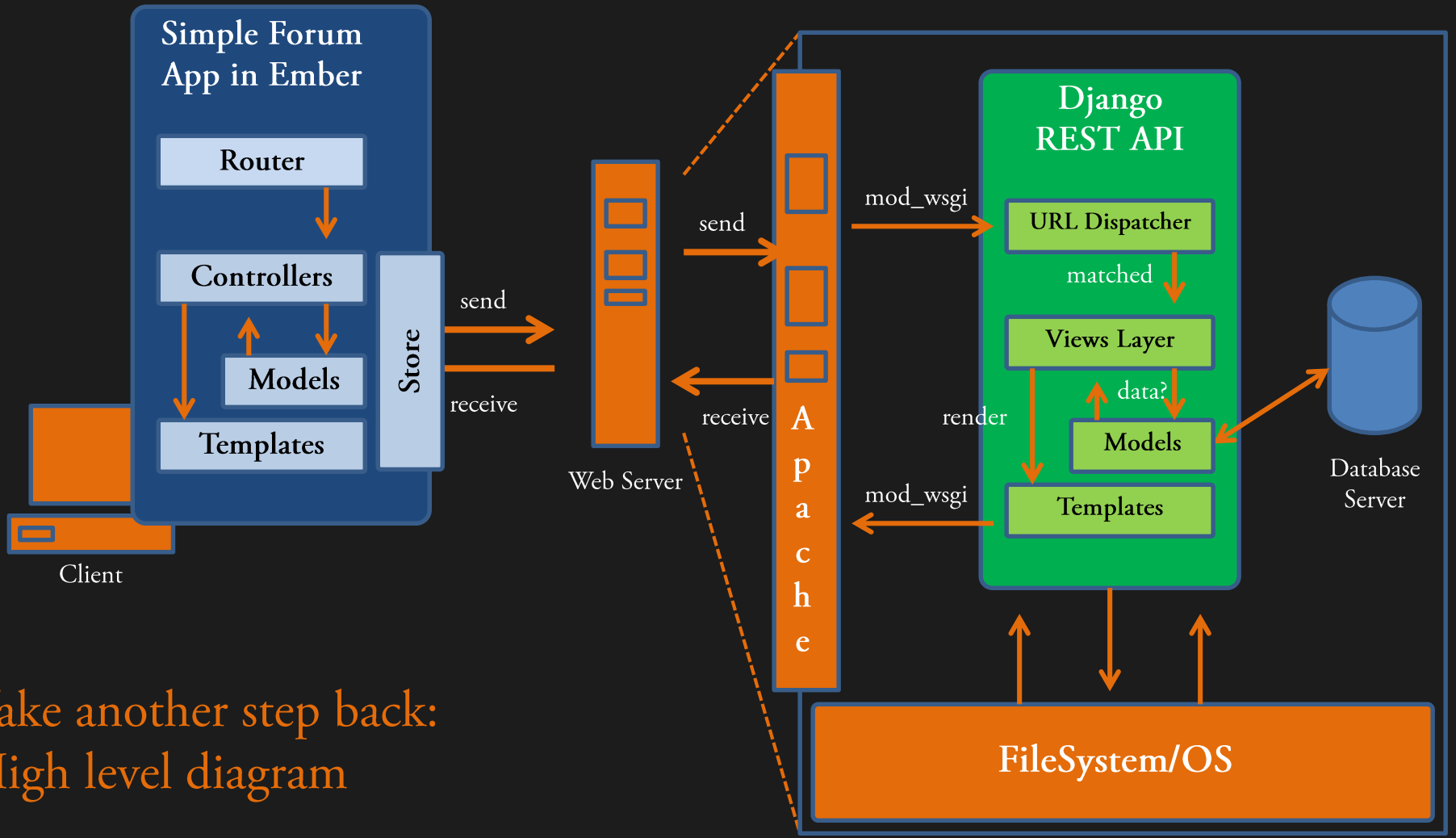


Take a step back: Application level (layered) diagram



Take another step back: Framework Level diagram





Take another step back:
High level diagram

The point

An architecture isn't limited to one diagram.
Each (use cases, application level, high level, etc) has its purposes.

Types of Diagrams

Informal Graphical Modeling

- Easy to represent ideas and structure conceptually
- Low learning curve, easy to use
- Most architectures are this

Formal

UML

- Easy-moderate to represent ideas and structure
- Moderate learning curve, harder to use (correctly)
- Provides standard parsability and automatic processing
- A lot of architectures are this

Others (e.g. Wright, Cross-UNITY, etc)

- moderate-hard based on language type
- Huge learning curves
- Potential for formal proofs of properties on systems
- Few architectures are this

Types of Diagrams: Informal Graphical Modeling

Generally produced in tools like Visio, PowerPoint, or OmniGraffle

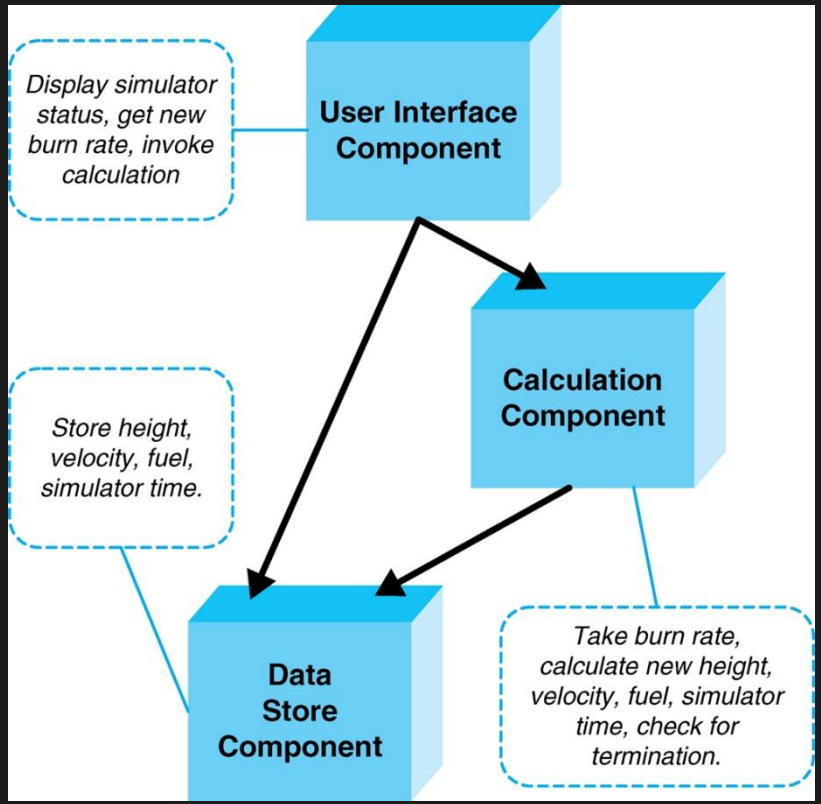
Advantages

- Can be aesthetically pleasing
- Size limitations (e.g., one slide, one page) generally constrain complexity of diagrams
- Extremely flexible due to large symbolic vocabulary
- Low to no learning curve

Disadvantages

- Ambiguous, non-rigorous, informal
  - But often treated otherwise
- Cannot be effectively processed or analyzed by machines/software

Types of Diagrams: Informal Example



Types of Diagrams: UML (Unified Modeling Language)

13 loosely-interconnected notations called diagrams that capture static and dynamic aspects of software-intensive systems

Advantages

- Support for a diverse array of viewpoints focused on many common software engineering concerns
- Ubiquity improves comprehensibility
- Extensive documentation and tool support from many vendors
- High precision and accuracy, minimal ambiguity

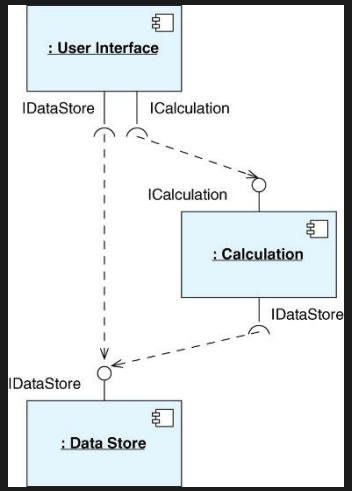
Disadvantages

- Needs customization through profiles to reduce ambiguity
- Difficult to assess consistency among views
- Difficult to capture foreign concepts or views
- Medium-high learning curve

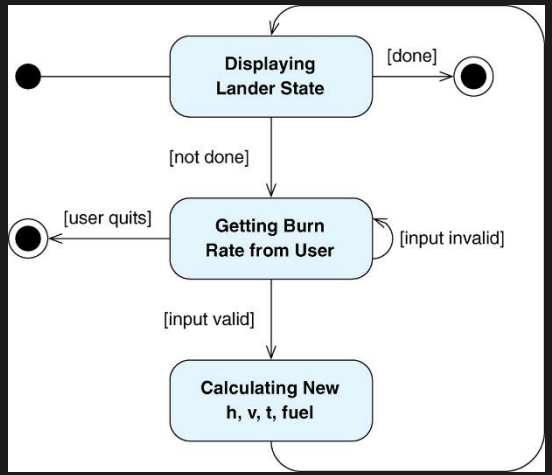
Types of Diagrams

UML Examples

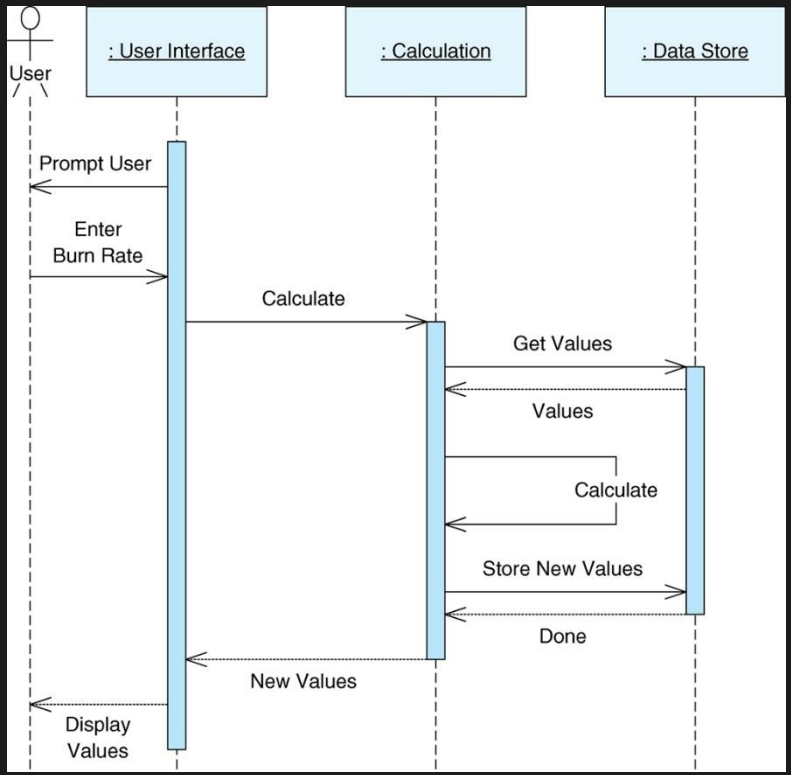
Component Diagram



Activity Diagram



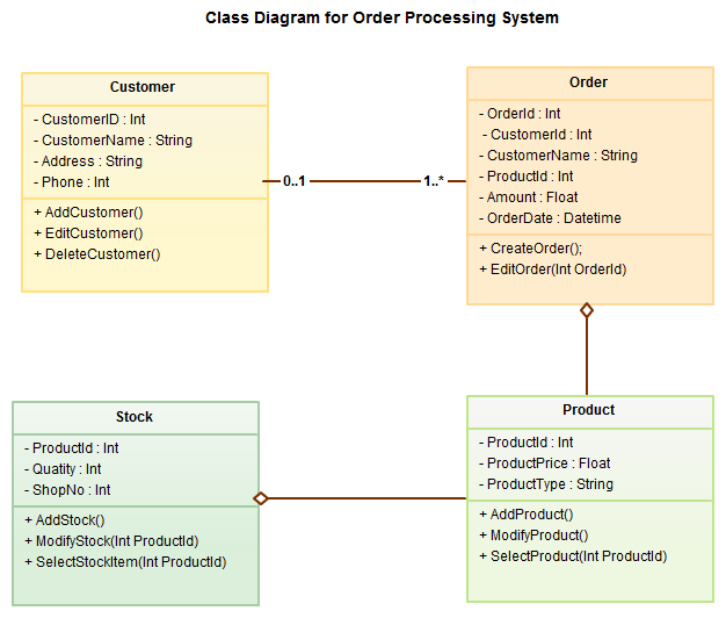
Sequence Diagram



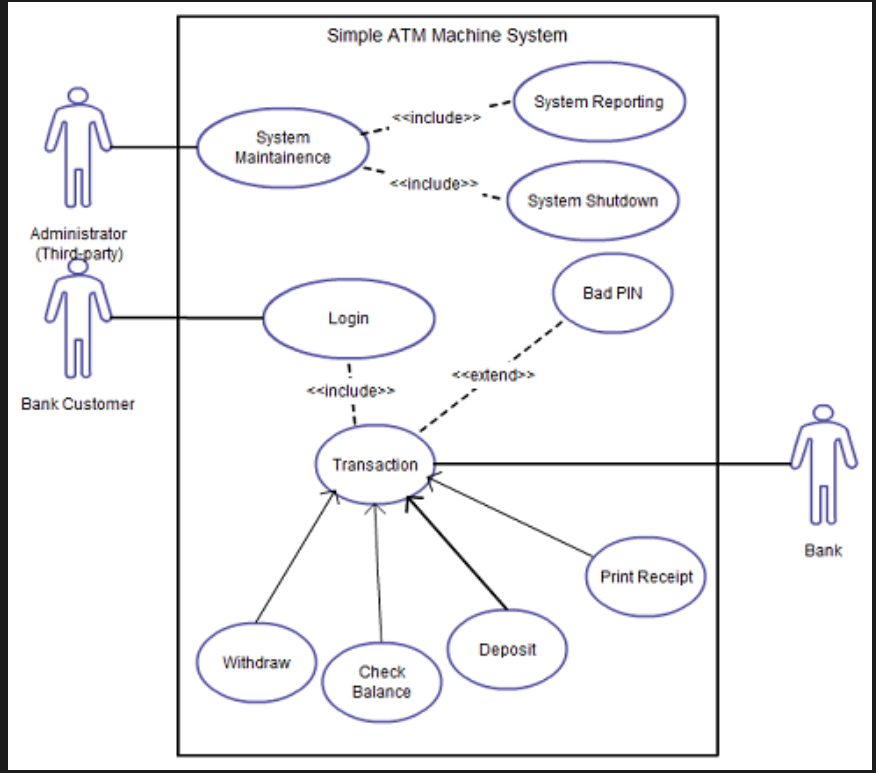
Types of Diagrams

UML Examples

Class Diagram



Use Case Diagram



Other types

Types of Diagrams: Other Formal Specification Languages

Other specification languages can be very highly formal and can be used for things such as security property verification (e.g. security controls from NIST SP800-53)

Advantages

- Can be used to verify system functionality
- Can provide mathematical guarantees
- Low or no ambiguity
- High precision and accuracy

Disadvantages

- Very high development time
- Verification only as good as the model
- High learning curve

Types of Diagrams: Formal Specification Languages – X-Unity

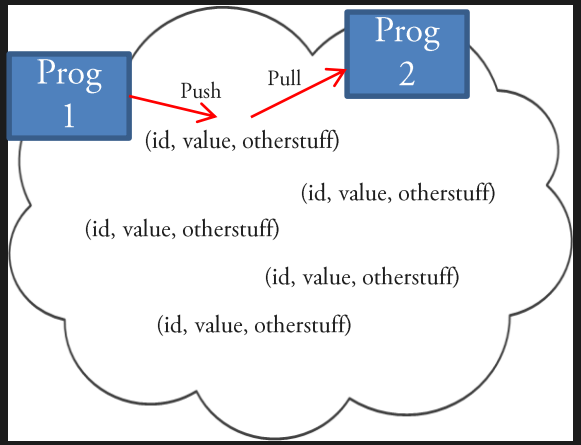
X-UNITY

- Follows a coordination paradigm
- Uses a formal mathematical representation called a tuplespace
- Has a proof logic that allows one to reason over system specification and prove properties about it
    - similar to Dijkstra's weakest precondition refinements

Types of Diagrams

X-Unity

- X-Unity: A coordination language
  - Core concept: Services interact using *tuples* of data
  - Services are represented as X-UNITY systems of *programs* that push and pull data from the *tuplespace*
  - Execution model non-deterministically selects a program statement from a service program and modifies a data tuple
  - Temporal logic tracks state changes across program execution to allow reasoning over the cloud system



SoS *SoSName*
  System $SystemName_1$
    Program$_{1..n}$ <*reference* and *parameters*>
      Components
        Instances of programs declared with parameters
      Governance
        Global impact statements such as **Promote**
  end $SystemName_i$

  …
  System $SystemName_k$ … **end** $SystemName_k$
end *SoSName*

Types of Diagrams

X-UNITY Example
Expressing a cloud web service composition

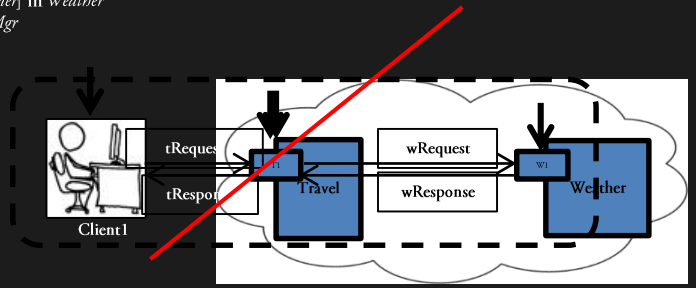
Matthew L. Hale, Michael T. Gamble, and Rose F. Gamble. 2013. "A Design and Verification Framework for Service Composition in the Cloud," in *Proceedings of the IEEE Ninth World Congress on Services*, IEEE Computer Society, Washington, DC, USA, 317-324, 2013. doi: 10.1109/SERVICES.2013.46

```
Proxy Client-Proxy (pid : PID)
  declare
    internal
      userInput : (FromDate, ToDate, From, To)
    exposed
      id : PID
    context
      plan : JSON
      travelPartner : ProxyID
  initially
    id, userInput := pid, GetUserInput()
    travelPartner, plan := ∅, ∅
  assign
    userInput := (∅, ∅, ∅, ∅) reacts-to plan ≠ ∅
  context
    travelPartner uses s!clientChannels[id] in SessionMgr
      given travelPartner = ∅
      where   travelPartner becomes s
    plan uses r!tRequest[travelPartner] in Travel
      given userInput ≠ (∅, ∅, ∅, ∅)
      where   userInput impacts r
    plan uses p!tResponse[travelPartner]  in Travel
        s!clientChannels[id] in SessionMgr
      given p ≠ ∅
      where
        plan becomes p
        travelPartner becomes ∅
        ∅, ∅ impacts p, s
end Client-Proxy
```

```
Proxy Travel-Proxy (pid : PID)
  declare
    exposed
      id : PID
      tRequest : PID → (FromDate, ToDate, From,To)
      tResponse : PID → JSON
    context
      weatherPartner : PID
      forecast : JSON
  initially
    id := pid
    tRequest, tResponse := id → (∅, ∅, ∅, ∅), id → ∅
    weatherPartner, forecast := ∅,∅
  assign
    ⟨tResponse(id) := CreateTravelPlan(tRequest(id), forecast);
      tRequest(id) := (∅, ∅, ∅, ∅); forecast := ∅⟩
      reacts-to forecast ≠ ∅
  context
    weatherPartner uses s!travelChannels[id] in SessionMgr
      given weatherPartner = ∅
      where   weatherPartner becomes s
  forecast uses r!wRequest[weatherPartner] in Weather
    given
      r = (∅, ∅)
      tRequest(id) ≠ (∅, ∅, ∅, ∅)
    where
      (tRequest(id) [0], tRequest(id) [3]) impacts r
  forecast uses   p!wResponse[weatherPartner] in Weather
        s!travelChannels[id] in SessionMgr
    given  p ≠ ∅
    where
      forecast becomes p
        weatherPartner becomes ∅
      ∅, ∅ impacts p, s
end Travel-Proxy
```

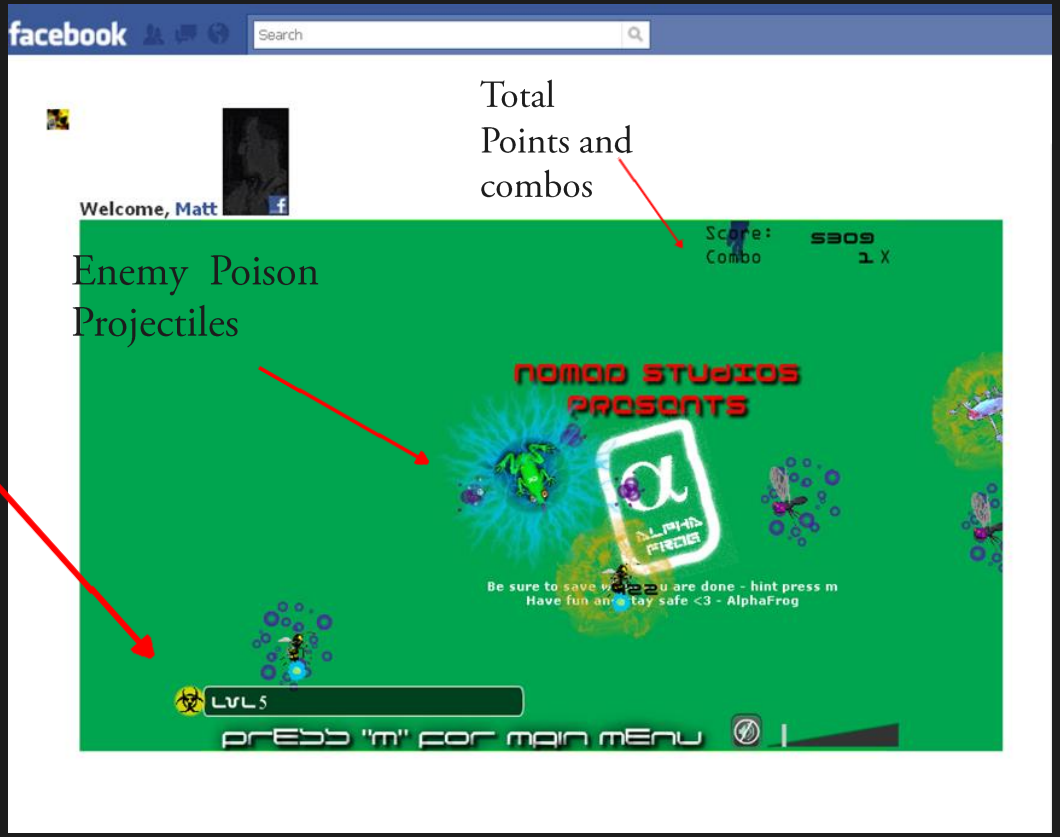
```
Proxy Weather-Proxy(pid : PID)
  declare
    exposed
      id : PID
      wRequest : PID → (Date, Location)
      wResponse : PID →JSON
    context
      weatherPartner : PID
    initially
      id := pid
      wRequest, wResponse, weatherPartner := id → (∅, ∅), ∅ → ∅, ∅
  assign
    if wRequest(id)  ≠ (∅, ∅) ∧ weatherPartner ≠ ∅
        ∧ wResponse(weatherPartner) = ∅
    then
      wResponse(weatherPartner) := CreateForecast(wRequest(id));
      wRequest(id):= (∅, ∅)
  context
    weatherPartner uses s!weatherChannels[id] in SessionMgr
      given weatherPartner = ∅
      where   weatherPartner becomes s
end Weather-Proxy
```

Session Management
(not pictured here)



Real Diagrams

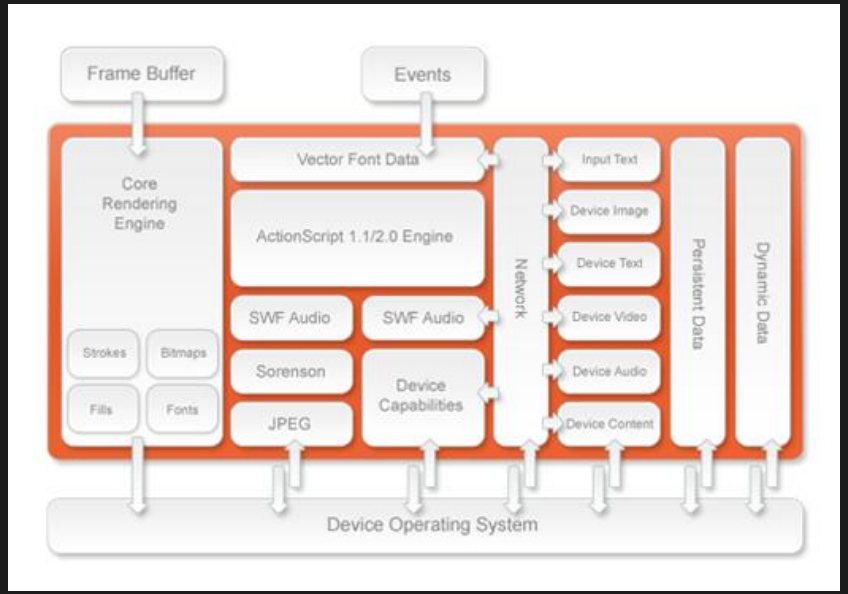
Example 1 – AlphaFrog (facebook connect)

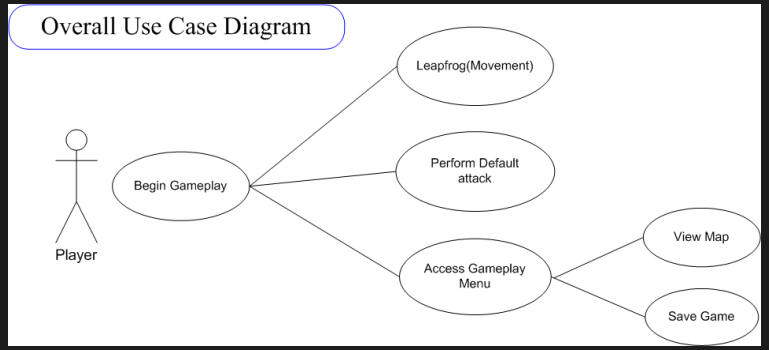


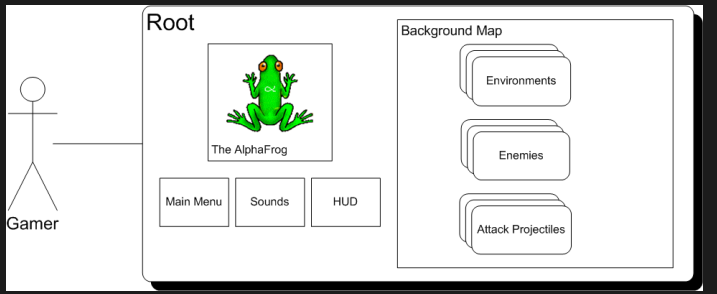
https://dl.dropboxusercontent.com/u/4594444/frog.swf

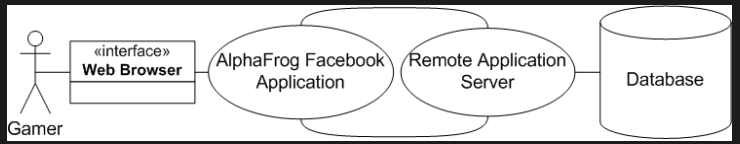
Real Diagrams

Example 1 – AlphaFrog (facebook connect)





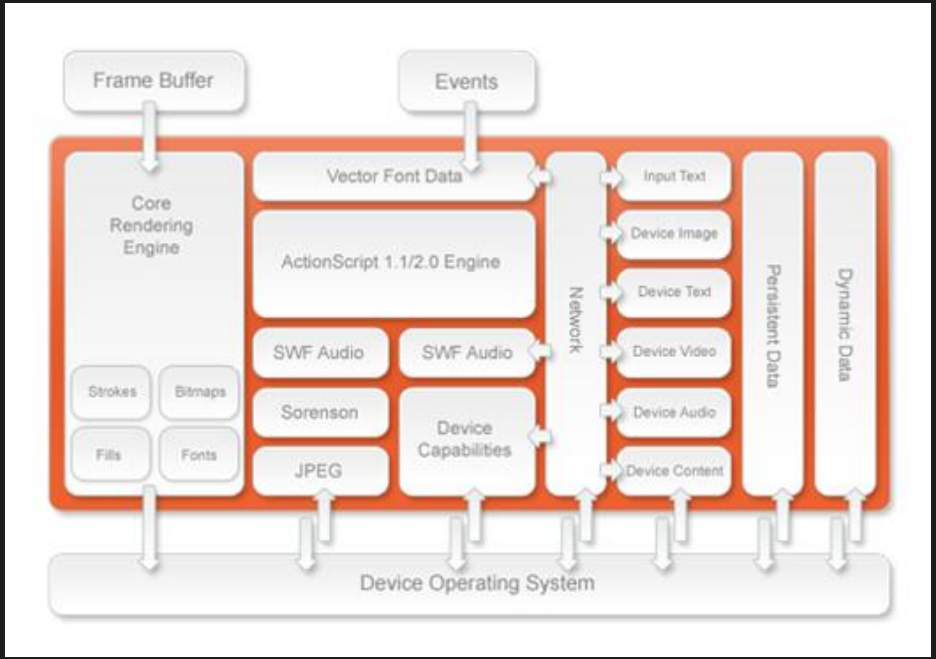




http://www.personal.utulsa.edu/~matt-halc/frog.swf

Real Diagrams

Example 2 – VisGA (Genetic algorithm in flash)



http://www.personal.utulsa.edu/~matt-hale/vis-ga/GA.swf

HUD

Text Feedback

GA algorithms

Crossover Operators

Mutation Operators

Population Management

Selection Methods

Pathfinding

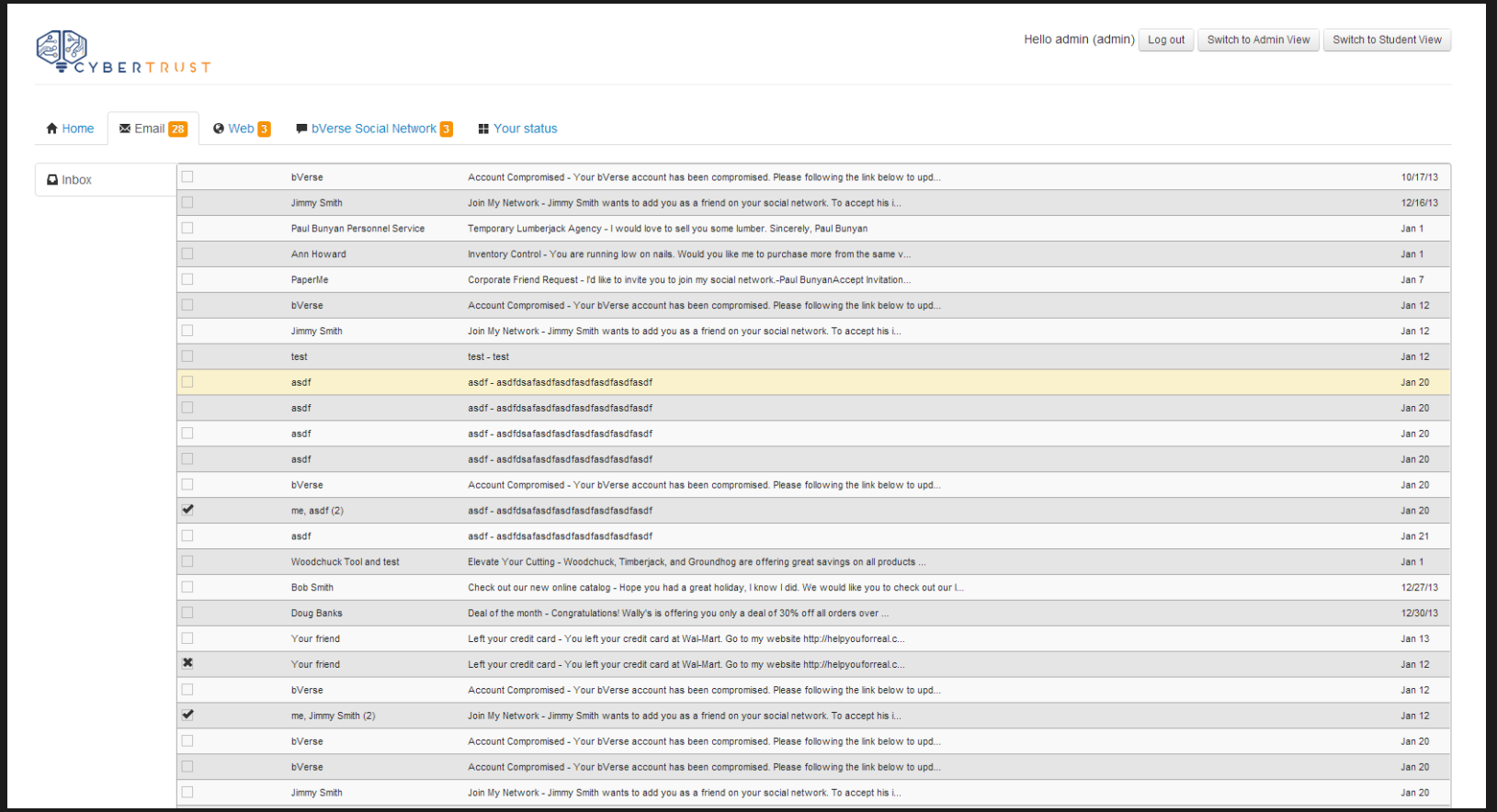
Line Generation

Line Drawing

Source and Obstacle Generation

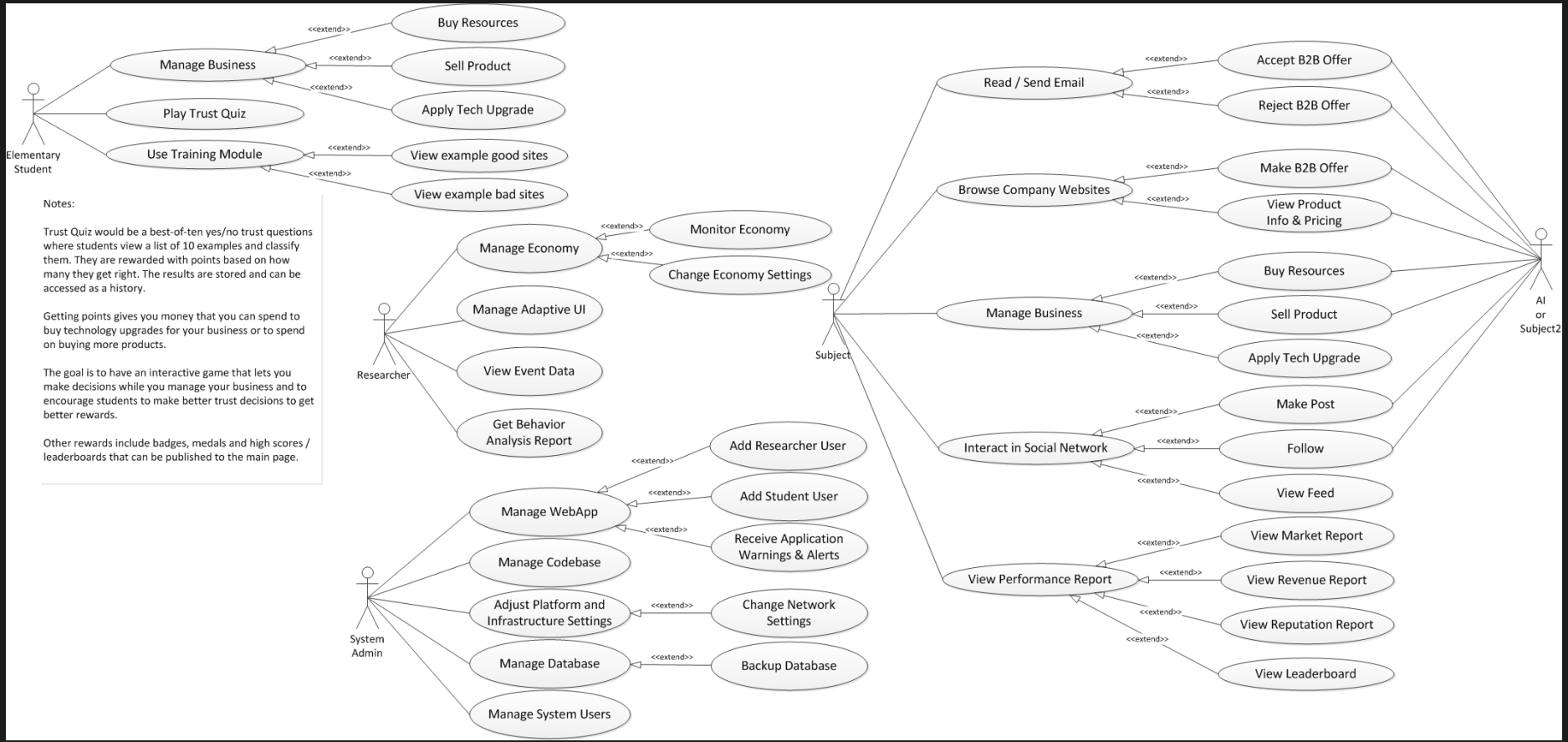
Real Diagrams

Example 3 - Cybertrust



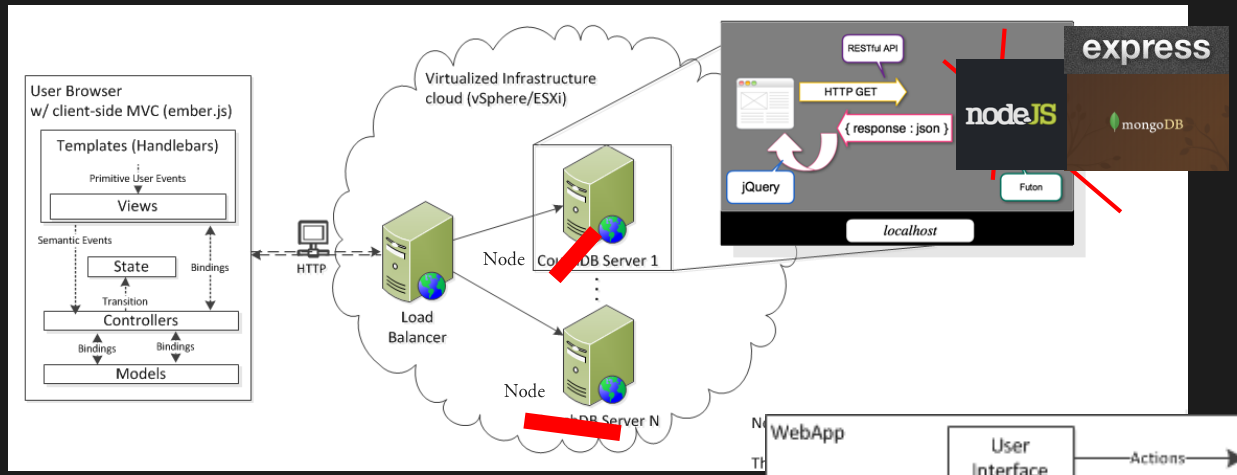
Real Diagrams

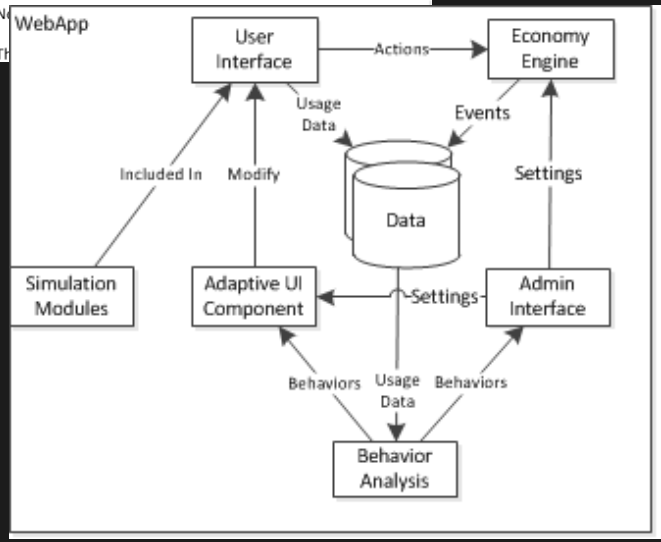
Example 3 - Cybertrust

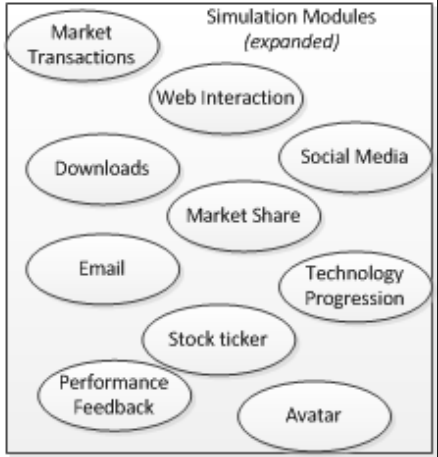


Real Diagrams

Example 3 - Cybertrust







http://cybertrust.mlhale.com/

Hands on

Draft an application level architecture for your app

Questions?

Matt Hale, PhD

University of Nebraska at Omaha

Interdisciplinary Informatics

mlhale@unomaha.edu

Twitter: @mlhale_