

# SFWR ENG 3A04 Summary

---

Author: Kemal Ahmed  
Instructor: Dr. Ridha Khedri  
Date: Fall 2014

*Math objects made using [MathType](#); graphs made using [Winplot](#).*

Please join GitHub and contribute to this document. There is a guide on how to do this on my GitHub.

## Table of Contents

Lecture 2 .....	2
Hierarchy of Requirement Specifications .....	2
Traceability Matrix .....	2
Early Assignment Details .....	2
Requirements Cont.....	3
e.g. 1) .....	3
Design Space.....	4
Diagram Types.....	6
Structural.....	7
Behavioural .....	8
Abstract Data Types.....	10
Object Oriented Analysis & Design .....	10
Design Principles .....	11
Types of Architectures.....	11
Data Centred Software Architectures .....	11
Model View Controller (MVC) .....	11
Presentation-Abstraction-Controller (PAC).....	12
e.g.) .....	12
i don't know... ..	13
Blackboard Architecture .....	14
Hierarchy Architecture.....	14

Main-Subroutines Architecture.....	14
Master Slave Architecture.....	14
Layered Architecture .....	15
Virtual Machine .....	15

## Lecture 2

### Hierarchy of Requirement Specifications

Pre Requirements:

- Requirements:
  - Requirements Document
    - System Specifications
    - Other Documents
      - Legal
      - Security
      - Privacy
  - Architectural Design
    - Types:
      - Dynamic
      - Stable
      - Determined by:
        - Elements
        - Connectors
    - Detailed Design

### Traceability Matrix

**Traceability Matrix:** a method of showing how each of the elements satisfies a requirement. You can use this to determine if a feature is necessary or if you are missing a feature.

Elements ( $E_i$ ) \ Requirements ( $R_i$ )	$R_1$	$R_2$	$R_n$
$E_1$		P	P
$E_2$	T		
$E_n$			

### Early Assignment Details

- The assignment can be submitted to a contest
- 2014-15 connect
- [dx.org/connect](http://dx.org/connect)
- Deadline: April 1<sup>st</sup>, 2015

- Prize: \$2000

## Requirements Cont.

**Business Event (BE):** the first, initiating input to a system that, but worded in the form of an event

Note: time can be an event, e.g. time to update your clocks

**Environment / system interactions:**

- *I/O between system and user*
- look at the system as a black box
- the last output occurs when the “business has been carried out”

**Viewpoints (VP):**

- *A target set of requirements*
- Think of it as different perspectives of how someone would want the system to be designed
- Includes things like who is using your product, but also who will be affected, such as economic perspective, i.e. cost

The more viewpoints you have, the better the representation of the system because you get a better overall perspective.

## e.g. 1)

For a  $BE_1$ , you have a list of VPs from  $VP_1$  to  $VP_n$ , and for  $BE_2$  you have a list of VPs from  $VP_1$  to  $VP_m$ .

If you have 2 viewpoints that have little relevance, you don't get rid of it. Instead, you mark them as void. This is because you may need it for the next BE(s)

**Functional Requirements:** fundamental reason for the system to exist

**Non-functional Requirements:** properties the system must have, e.g. precision, availability, security, usability, look, etc.; it is based on the environment of the system; more qualitative

**Constraint:** global issue that shapes the requirements; quantitative limits

Determine functional, *then* non-functional requirements.

**Scenario:** interactions between the system and the user / environment (could be time)

**Mode:** what you think it means, but formally, a non-empty set of equivalent states

- reflexive
- transitive
- symmetric
- $x'Ry$  and  $y'Rx$

Complete graph with  $n$  nodes is  $K_n$ .

## Design Space

- Hardware-hiding modules:
  - Language to communicate with the hard drive
  - Virtual Machine hiding module
- Behaviour hiding modules:
  - Controller classes: sequence of events
  - Change due to requirements
- Software decision-hiding modules:
  - Algorithms
  - Physics constants
  - Theorems (i.e. math)
  - Data types
    - $n$ -Tuple; a record
      - $n$  gets
      - $n$  sets
    - Set
      - IsMember
      - IsEmpty
      - Insert
      - Remove
    - List
      - IsEmpty
      - GetHead
      - GetNext (last element)

**Asynchronous operation:** process operates independently of other processes

**Synchronous operation:** other processes finish before some other process has finished

**Blocking:** process causes other processes to stop

**Non-blocking:** process runs without stopping other processes

[More](#)

**Semaphore:**

**Protocol:** a method of communication

**MVC:** the way every software program is analyzed

**Model:** (a.k.a. Data level) constants and stored data the system interacts with

**View:** (a.k.a. Interface) what the users see and how they interact with the system

**Controller:** (a.k.a. Business Logic) what processes the data from the model

**Connector:** an indicator of interaction among components

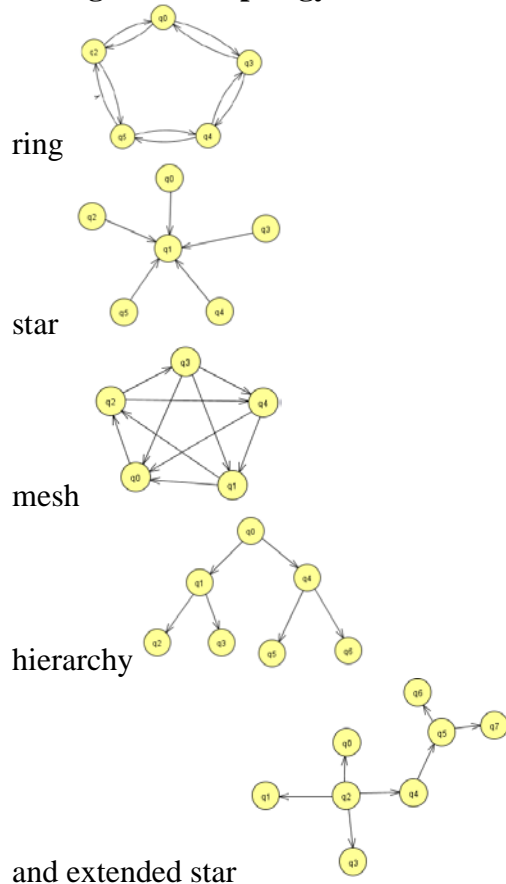
**Signature-based connector:** single operation; works as long as you communicate using the correct inputs (like Radio)

**Protocol-based connector:** multiple operations; when communicating, both communicate with each other and confirm a connection (like WiFi)

**Formal model:** a representation of what you are going to build, based on math

**Informal model:** not formal

**Configuration topology:** different shapes of networks, including bus ignore arrows



### Unified Modelling Language (UML):

Class Name
Attributes: name: String address: String
Operations / Functions

It is usually organized in **structural diagrams**, which show relationships between classes through connectors.

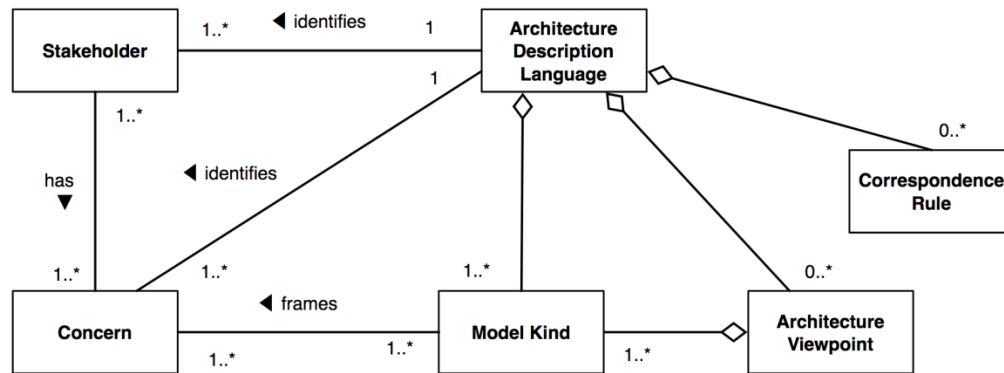
+: public

-: private

## Diagram Types

**Architecture Description Language (ADL):** languages describing the software and hardware architecture of a system

- usually graphical syntax
- Supports the tasks of architecture creation, refinement and validation
- Provide a basis for further implementation



**Inheritance:** [*identified by arrows*] the child gets some of its data / functions from the parent objects, although local functions have higher precedence; the child class can access the parent's classes publicly

Example:

```
class Dog
    Eat;
    Walk;
    Bark;
    Play;
end;
class Cat extends Dog    //is
    Purr;
    Bark = null;
end;
```

**Aggregation:** [*hinge identified by hollow diamonds*] something is made of independent parts that can exist without the parent object (think: is it useful on its own?); the child can access the parent's classes privately

```
class Cat includes Dog; //has
    Eat = Dog.Eat;
    Walk = Dog.Walk;
    Play = Dog.Play;
    Purr;
end;
```

**Composition:** [*hinge identified by black diamonds*] parts are dependent on the parent object to exist

```
class Pet
```

```

    Eat;
    Walk;
    Play;
end;
class Dog extends Pet    //is
    Bark;
end;

class Cat extends Pet    //is
    Purr;
end;

```

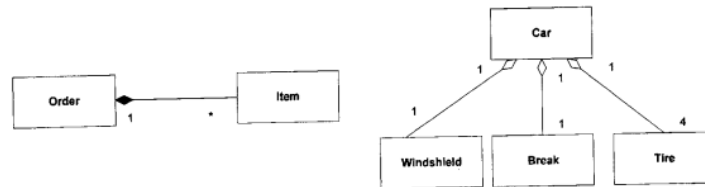


Figure: Composition (left) Aggregation (right)

*The item doesn't exist without the order; the windshield is useful without the car existing.*  
 Note: "include" and "extend" mean different things here than in use case diagrams

It's especially important to have low coupling when you can't change the higher level object

**Dependencies:** [identified by dashed arrows] if a class, X, depends on another class, Y, then changes to the elements Y will lead to the changes of X

## Structural

### Composite Structure Diagram

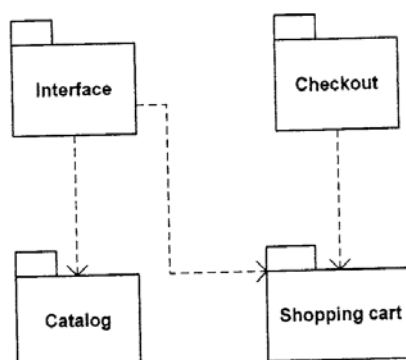
- Rectangle: structural classes
- Ellipse: abstract construct of relationship between classes

### Component Diagram

- Balls: class that outputs
- Sockets: class that takes input from balls

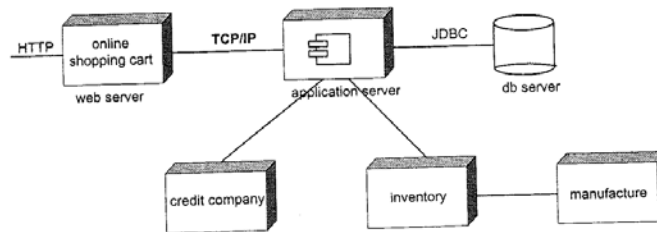
### Package Diagram: package structure

- Folders: packages



**Deployment Diagram:** physical hardware, software, network connections

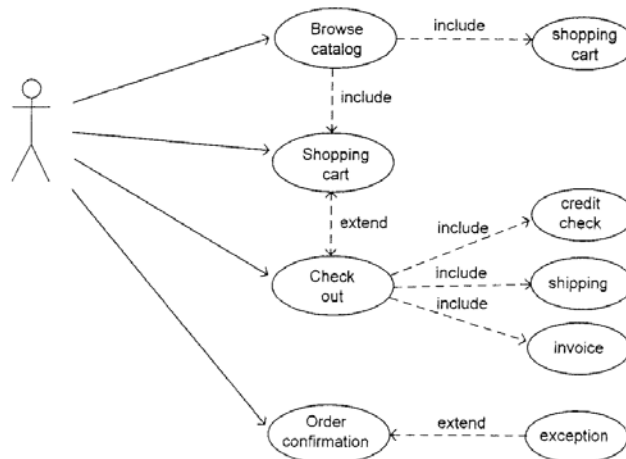
- Cubes: computing resources
- Cylinders: database [sometimes]



## Behavioural

**Use Case:** how system reacts to BEs

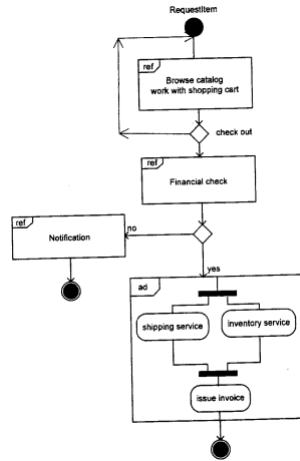
- Communication between actors
- **Actors:** *[represented by a stick figure]* does not have to be a human
  - provide BEs
- **Include:** mandatory behaviour; the child needs the parent to exist
- **Extend:** optional behaviour; the child can exist without the parent
- Uses:
- “Use Case” ⇔ “Scenario”
- Each ellipse is a use case



**Activity Diagram:** data and control flow of system

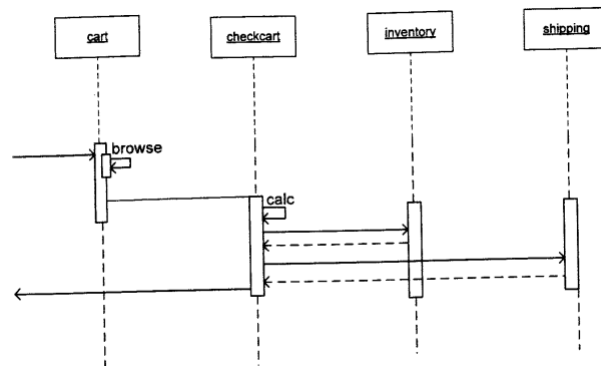
- Rounded rectangles: actions in system
- Solid hub: fork and joint points
- Surrounded disk: terminate
- Diamond: decision
- Disk: start point





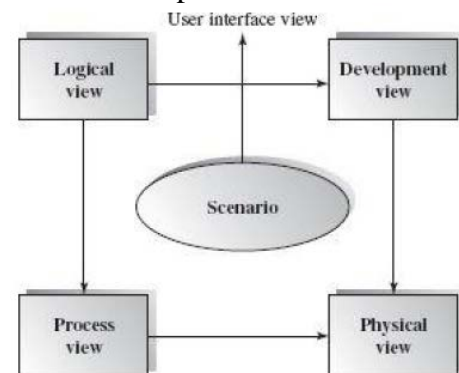
**Sequence Diagram:** how flow thru classes to fulfill requirements

- Rectangles on top identify classes
- Arrows show flow of data and how they fulfill requirements
- Smaller boxes inside the bigger boxes are other implementations of the same object



**4+1 Model:**

- Scenario: overall encompasses other views
- Logical View:
- Physical View: how software interfaces with equipment, hardware, etc.
- Development View: how classes and directories are organized
- Process View: communication between classes
- User Interface View: look & feel of product



## Abstract Data Types

ADTs: the study of structures

Types of ADTs

For a given Set, what are the Functions of the set?  $(S, F_S)$

$$\begin{aligned} (\mathbb{N}, F_{\mathbb{N}}), \\ (\mathbb{Z}, F_{\mathbb{Z}}) \subseteq (\mathbb{R}, F_{\mathbb{R}}) \end{aligned}$$

Algebra:  $(\mathbb{C}, \{+, \cdot, \dots\})$

Signature defines how number types change after an operation

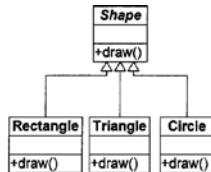
$$+ : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

Numbers in an ADT must be:

- Finite
- Discrete
- Countable: there is only one number for each number
  - $f : \mathbb{N} \rightarrow S$

## Object Oriented Analysis & Design

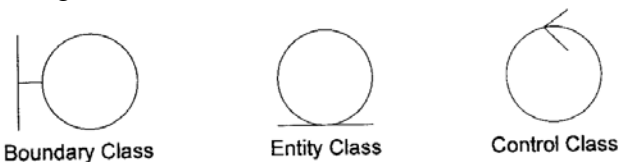
Generalization / pattern: [denoted by hollow triangle arrow] inheritance relationship



## Order Processing System (OPS):

Secrets:

- Boundary classes:
  - Hardware-hiding
  - Virtual Machine
  - Interface
- Entity classes: data structure
- Controller Classes: algorithm



**Polymorphism:** being able to access different functions with the same function name

- Horizontal overloading: having multiple functions within the same class, usually for different input types

- Vertical overloading: having functions from a parent and child class
  - Take a Lion, Tiger, Bear, and Fish. They are all Animal objects. Say the animal object has a function, hasClaws=true. The Fish object also has a function hasClaws, except its value is hasClaws=false

## Design Principles

**Inductive reasoning:**

**Deductive reasoning:**

**Open-closed design principle:** open for addition, closed for modification

## Types of Architectures

- block flow architecture (62)
- batch sequential architecture (63-64)
- pipe and filter architecture (64-68)
- Process control architecture (68)
- Repository architecture (71-73)
- Blackboard architecture (75-78)
- Hierarchical architecture (81)
- Main-Subroutines Architecture (82)
- Master-slave architecture (83)
- Layered Architecture (84-86)
- Model-View-Controller (MVC) Architecture (102-107)

## State Diagrams

input/output

## Data Centred Software Architectures

There are multiple ways of organizing your data stores.

- One data store for your attributes
  - If the data store goes down, you're screwed
- One proxy that guides you to multiple data stores
  - You are at risk of accessing out-of-date data
  - You need to sync the servers
  - More expensive to set up multiple data stores

## Model View Controller (MVC)

**Model:** data

**View:** how the person interacts with the system; representation of the data

**Controller:** processes data

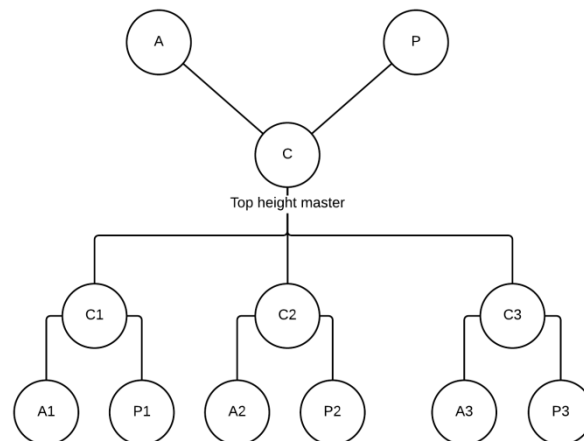
- All 3 modules are interconnected
- This is useful for user interfaces, where data changes all the time. Think “look and feel” features in GUI applications.
- Data model changes are difficult

## Presentation-Abstraction-Controller (PAC)

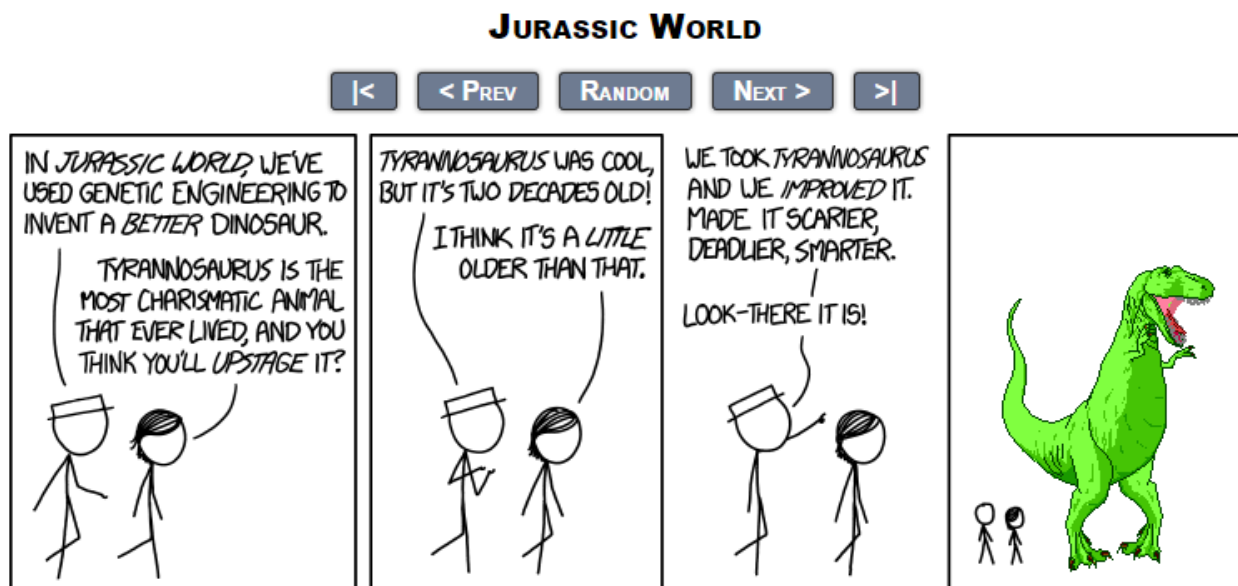
Presentation: view

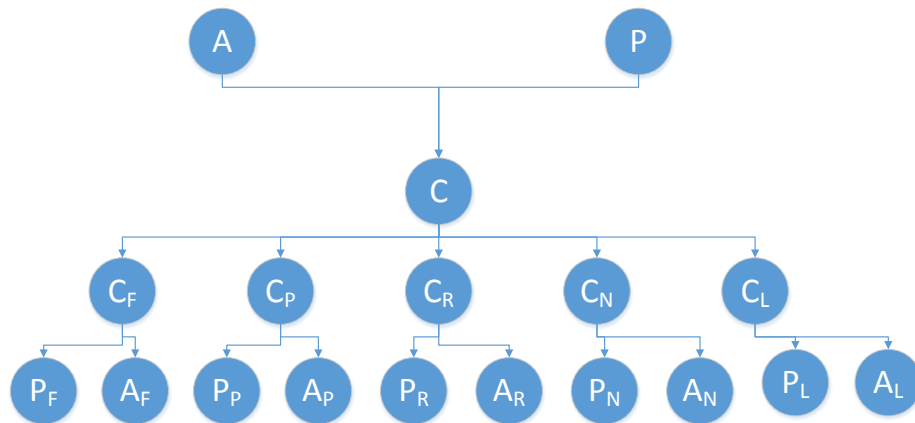
Abstraction: data model

- Supports multi-tasking & multi-viewing

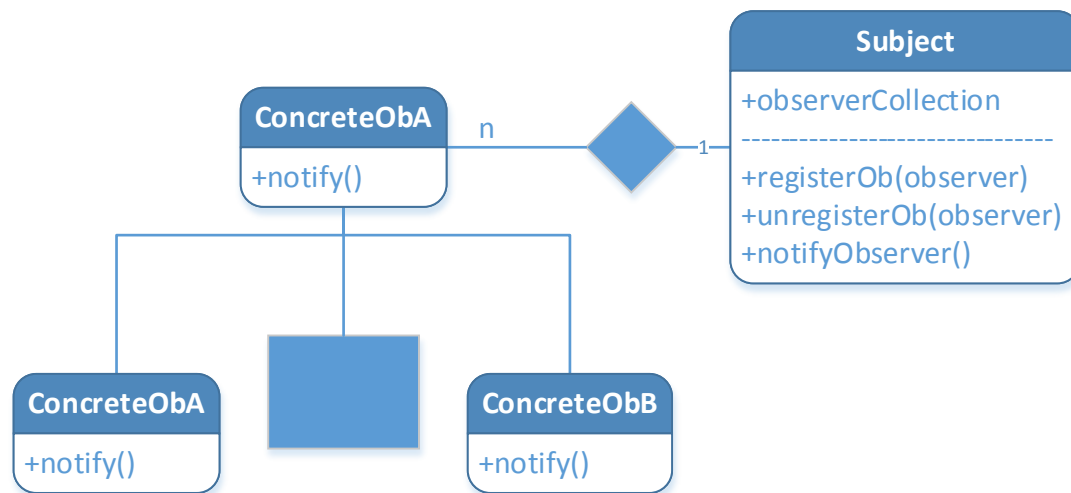


e.g.)





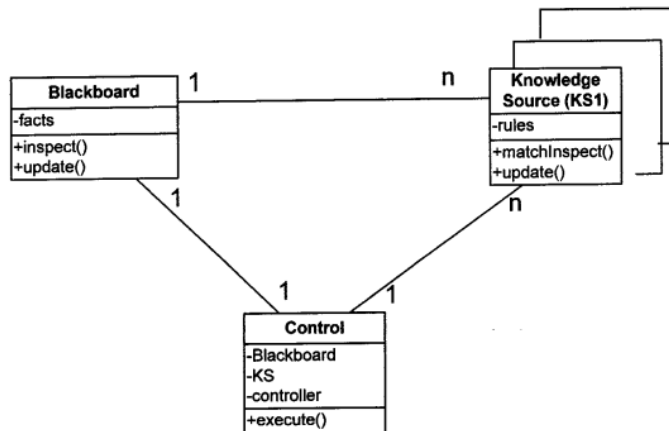
i don't know...



```

notifyObserver()
{
    for observer in observerCollection, call observer.notify()
}
  
```

## Blackboard Architecture



The numbers mean:

*n* means:

This is similar to the Master/Slaves Architecture. However, it is different because:

- Keeps agent in sync
- Each agent serves a different function
- Agents do not communicate to tell each other how much of the work to do

**Knowledge Source (KS):**

## Hierarchy Architecture

User level

Kernel level

Hardware level

*Execution point of the system / main / maestro* is at the top

## Main-Subroutines Architecture

**Main-Subroutines Architecture:** A type of hierarchical architecture

A huge change from the old systems with the `goto` statements is when they introduced the idea of calling functions, which made everything more sequential.

This architecture involves mapping out the structure of what functions you are going to call for each option.

**Benefits:** easy to decompose based on tasks

## Master Slave Architecture

**Master/Slave Architecture:** A type of hierarchical architecture

**Agent:** something that you're

Split the land into multiple sections and separate the task by workers

- Slaves communicate to split up the tasks evenly
- Slaves are only slightly different from each other (if at all)

Sometimes, slaves can have slaves. However this can be bad because too many slaves requires a lot of resources for communicating and putting the information together.

Find a sequence in a part of the data

**Layer:** how many levels of slaves

## **Layered Architecture**

**Layered Architecture:** A type of hierarchical architecture

Each layer can communicate only sequentially because of separation of concerns

Each layer has 2 interfaces:

1. Layer to send to next layer
2. Layer to receive from previous layer

Higher layers are more abstract or generic than lower ones

## **Virtual Machine**

Having a common intermediate coder that simulates

Portable, simple