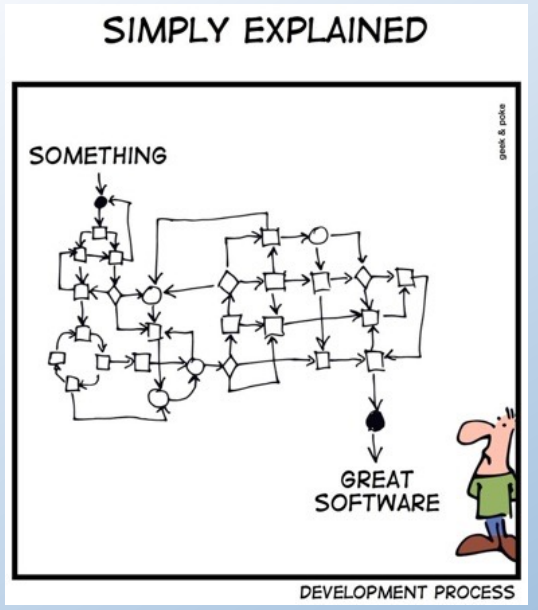
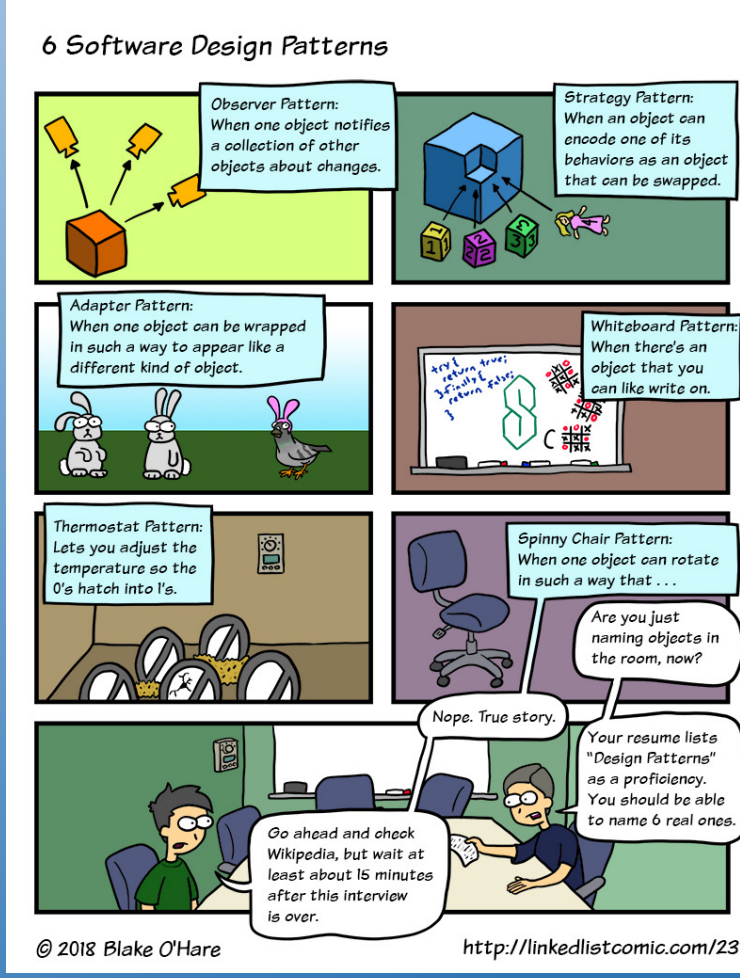






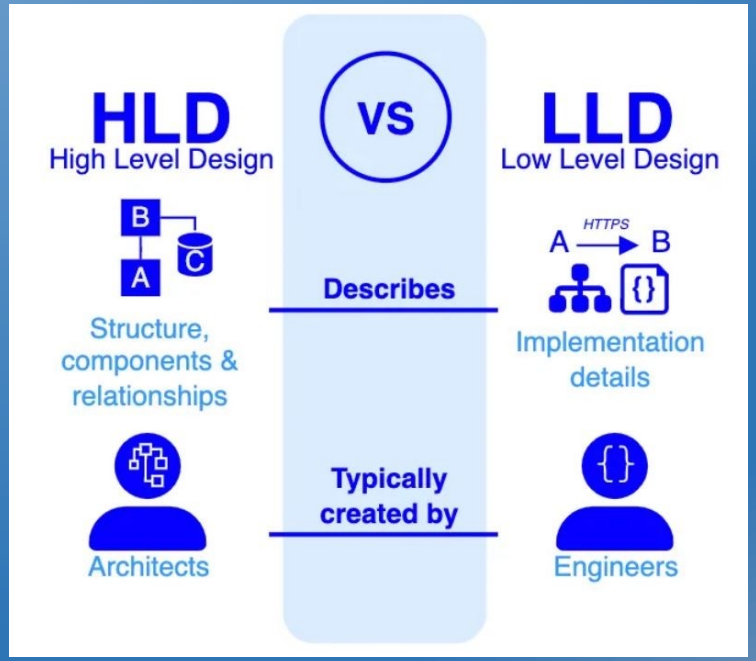

# High Level Design

## CSCI2340: Software Engineering of Large Systems

Steven P. Reiss

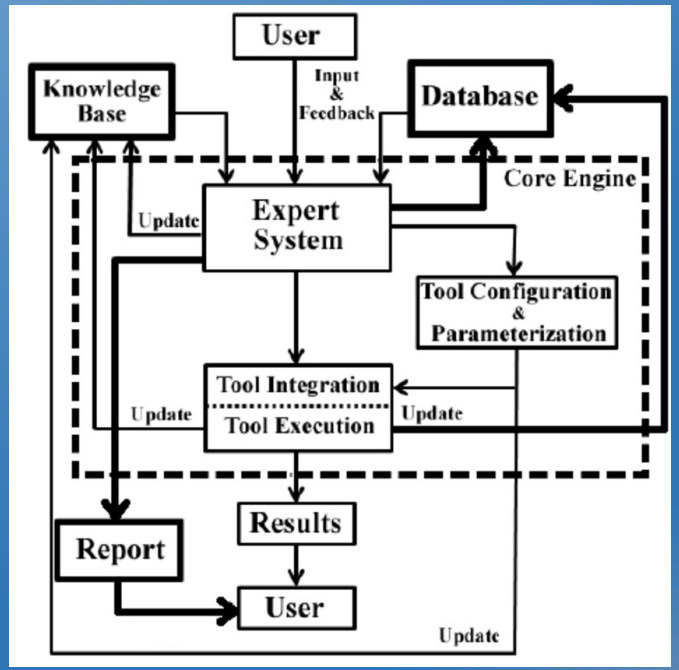
# High Level Design

- Suppose you know your software architecture
  - You next must design its software constituents
  - Both its components and communications
- To do this you need to consider:
  - What are the **goals** of your design
  - What are the **actual** software components
  - How to **represent** the design
- This is what we cover next
  - Today and next time

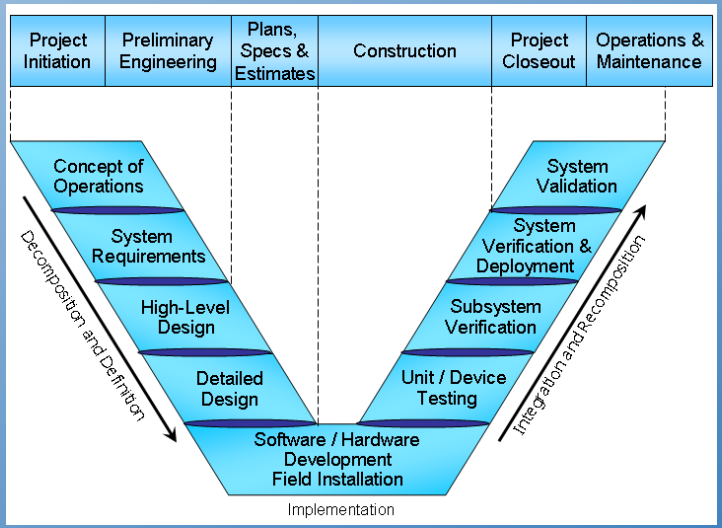


# What is High-Level Design

- Bridge between software architecture and detailed design
  - Overlaps with software architecture
    - Elucidation of the software architecture
  - Overlaps with detailed design
    - Starting point for low-level design
- Typically based on software architecture
  - Determine how the nodes and links might be implemented
  - Develop a consistent, implementable structure
- Each architectural component can be
  - Single process in a multiple process system
    - Might need to be broken down more
  - Single subsystem (sets of packages)
    - These will need to be broken down more
  - Single package (set of classes)
  - Single module (set of functions/classes/…)
- Design components can combine or split these

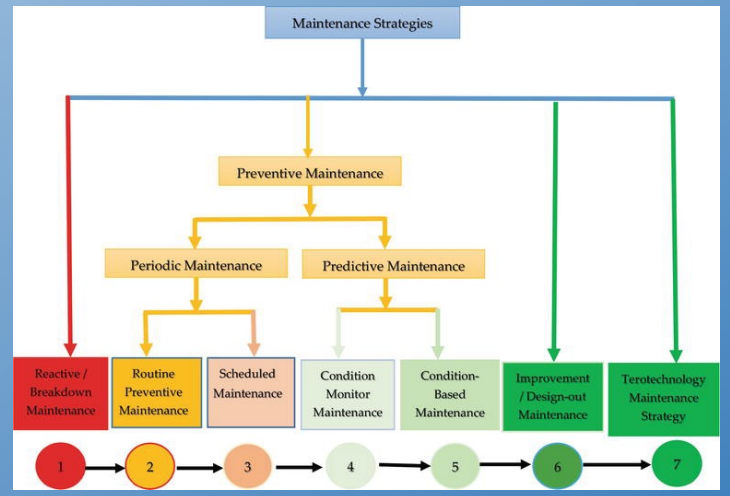


# Goals for High-Level Design




- Working system with minimal effort
  - But initial system is a small part of development
- Maintainability
  - The resultant system must be maintainable over time
  - The design must accommodate changes
- Evolution
  - Easy to add new features, handle changing needs, handle changing environments
  - Without degeneration (code deteriorates and convolutes over time)
- Risk
  - Need to minimize immediate and future risks
- Security, Privacy, & Ethics
  - These should be considered as part of the design
- Team Development
  - Design should allow independent work by team members
  - Design should build on the strengths of the team

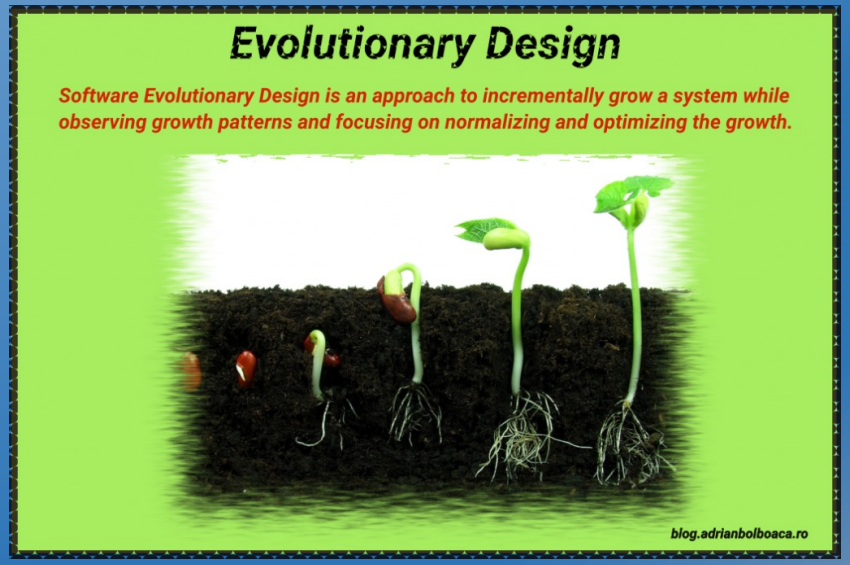
# Design for Maintenance



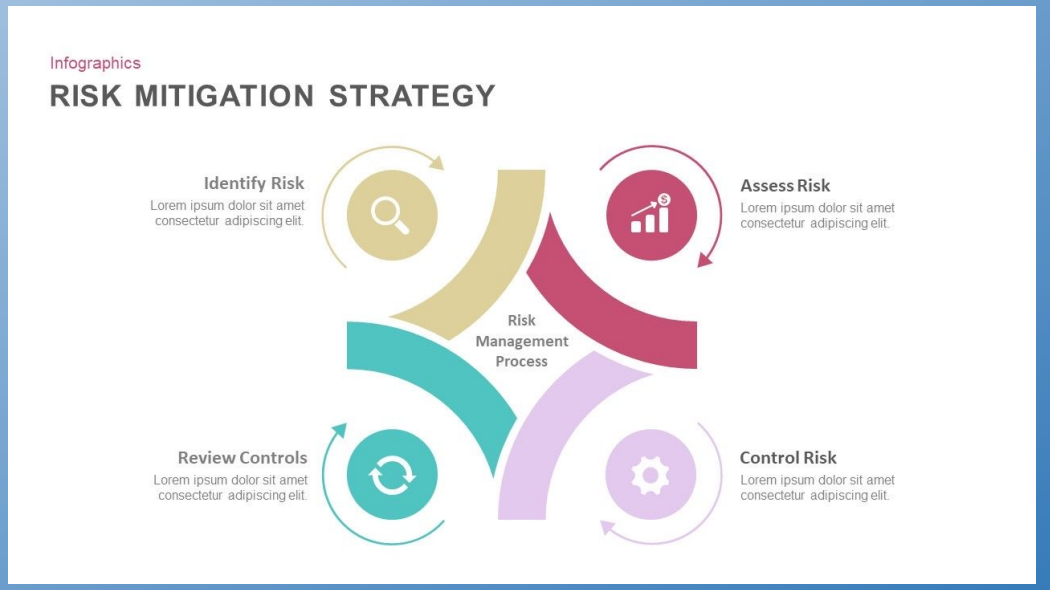

- Determine what outside things might change
  - OS interface, DBMS interface, User interface
  - Algorithms (e.g., LLM to use)
  - Ensure these are isolated as much as possible
  - Make these easy to change without affecting whole system
- Make it easy to find and isolate problems (bugs)
  - Defensive design
    - Error handling, exceptions, …
  - Logging
  - Testability
  - Incorporate these into initial design (high-level and detailed)

# Design for Evolution

- Determine what parts of the system are likely to change
  - Ensure these are isolated (single component if possible)
  - Changes should be local where possible
- Determine what features might be added
  - Required, higher-priority, short-term requirements (beyond Core)
  - Optional, lower-priority, long-term requirements
  - Design the system so that adding these is possible
    - Without changing too much of the system
    - Higher priority -> easier to add
    - Things requiring a major rewrite won't be added
  - This is why I suggest full requirement specifications
- Support agile development
  - Make it easy to add new features
  - Features should be in a small number of components



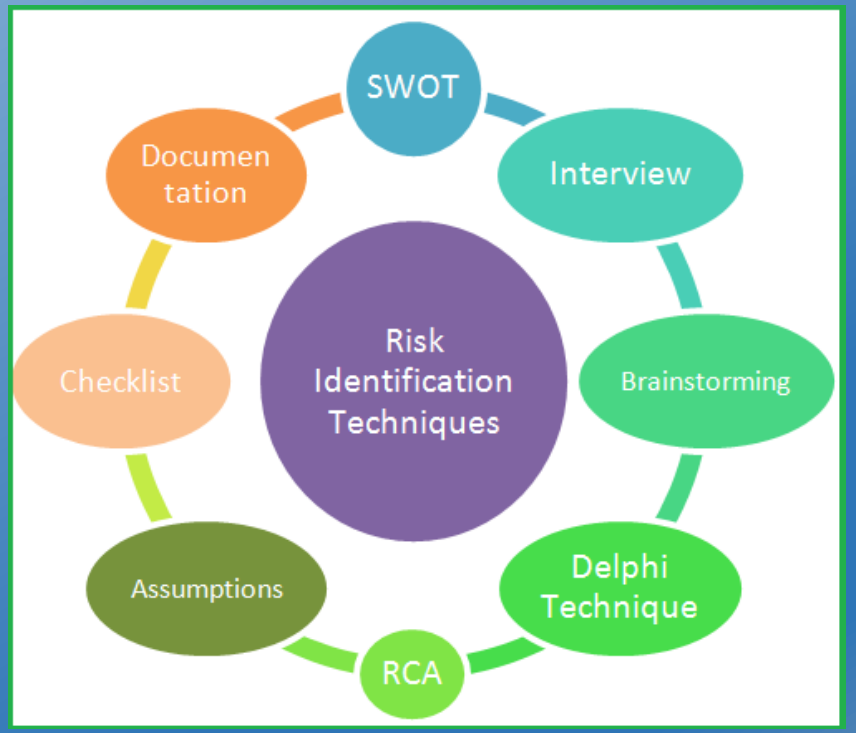
# Design for Risk



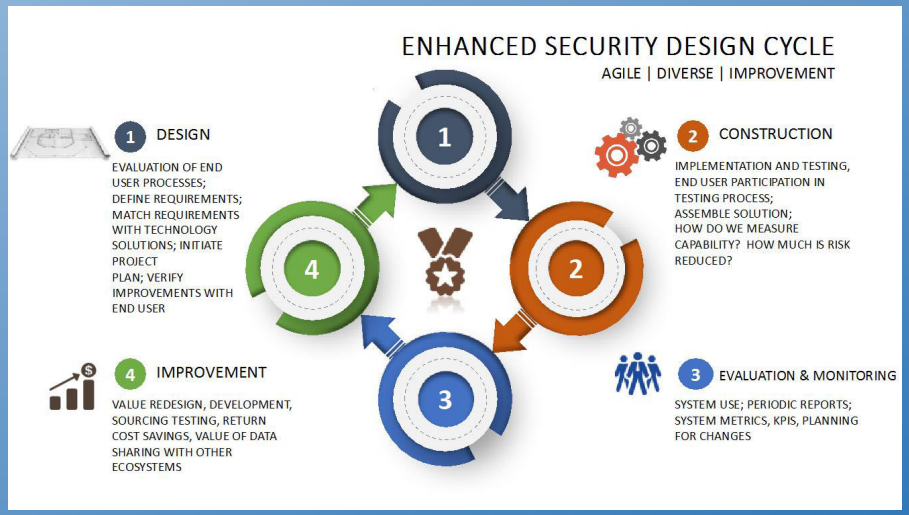

- Identify potential risks
  - Requirements, specifications, skeptic
- Address these risks
  - Either isolate them
    - Make it easy to try alternatives
    - Make it easy to change solutions in the future as needed
  - Or design the system around a solution for them
    - Concentrate the design on the risk
- Prototyping to check out potential solutions
  - Ensure design will work with test system

# Identifying Risks

- What can go wrong (skeptic)
  - Always be skeptical of your own code (and others)
- What do you not understand
  - How to implement something
  - How complex some code is
    - How long something it will take to implement
  - How long code will take to run
  - How large data might be
  - Whether something will work or not
- External assumptions that might change
- User interface risks
  - The user interface is going to change
- Competitive risks
- Personnel risks

# Design for Security, Privacy & Ethics




- What data needs to be secured
  - Or kept private
  - Or is legally restricted in some way
  - Or is company confidential
- Isolate that data in one component
  - Even if it's a separate component just for the data
  - Then securing the data involves a single component
  - Keep it in a single component as you break down the design
- What are the ethical risks of your system
  - Difficult to determine how the system will eventually be used
  - But you can take a first step
  - Appropriate checking and feedback mechanisms in the initial design
  - Can these be avoided

# Design for Team Development

- Each team member should have their own code / components
  - Independence improves programmer efficiency
  - Independence allows asynchronous development
  - Allow individual testing, debugging
  - Addressing the strengths of the team
- Well-defined **interfaces** between people
  - You know precisely what to code
  - Know how to use other's code
  - Others know how to use what you code
  - But you should not need to know the others actual code
- Number of components vs team size
  - Ensure there is a components for each team member
  - Can have additional components
  - If fewer, ensure the components are separable
    - Multiple team members on one component -> component can be split

# Correct vs Incorrect Designs

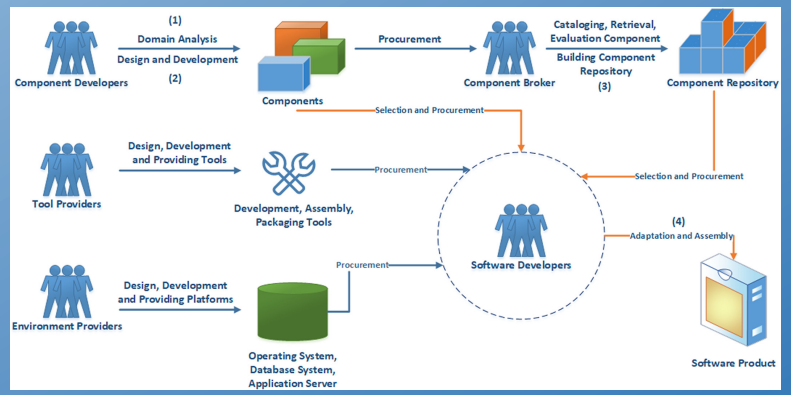

- Almost any design can be made to work
  - That doesn't make it correct however
- Addressing these design goals makes life easier
  - Initially (creating the system)
  - More importantly as the system evolves & is extended and maintained
- A good design can cut the workload significantly
  - Half the amount of code
  - Less refactoring and rewriting needed
  - Less time adding new features (easier to evolve)
  - Easier to debug (finding and isolating problems)
  - Easier to maintain and evolve
  - Easier to test and deploy

# EXERCISE

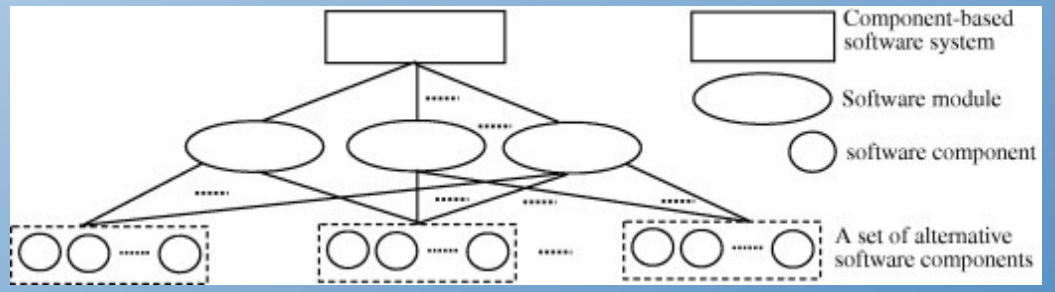
- Let’s assess our initial programming efforts
- We will split into small groups (<4). Within each group show each other the initial version of your programming assignment and provide feedback to the others.
  - Feedback should be constructive
  - Get ideas from others for your own program
  - Give others ideas on how theirs might be improved
- 10 minutes

# Approaches to High-Level Design

- First step: Identify components
  - Architectural components are a starting point
  - Break down components into subcomponents
  - Identify necessary components based on goals
  - Think in terms of packages or modules or processes
- Keep breaking down components until
  - Component can be handled by an individual or two
  - Component implementation does not affect the rest of the system
  - Component implementation not affected by rest of system
  - Component is a single package or module or service
  - Component is well understood
  - The overall design is understood

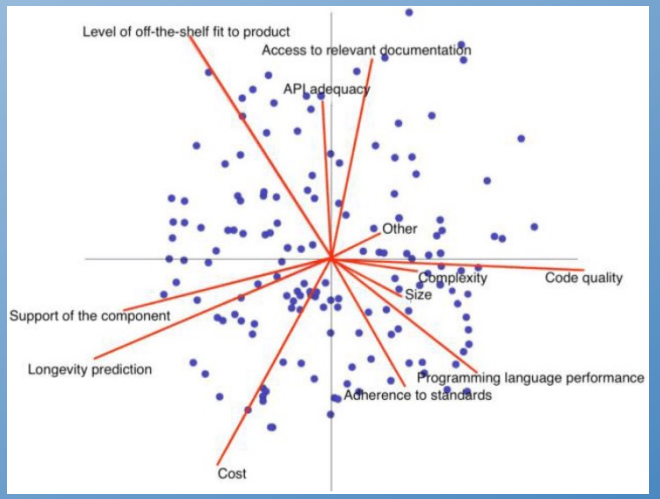


# Component Selection



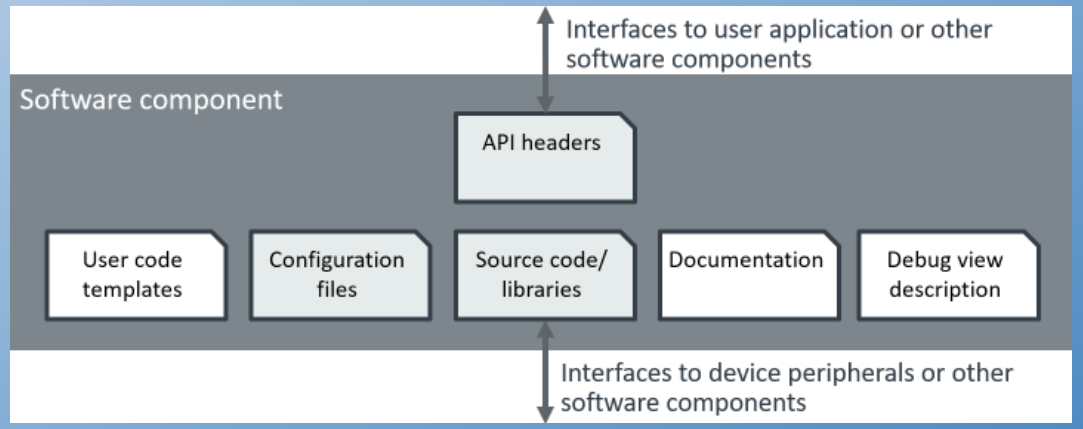

- Top-level components reflect the software architecture
  - Reflect the process structure
  - Separate architectural components are separate
  - Break these components down into subcomponents as needed
  - Top-down approach to finding components
  - Find commonalities (DAG, not a tree)
- Isolated elements should be in a single component
  - Add these as components
  - Isolated for maintenance (OS, UI, DBMS dependencies)
  - Isolated for evolution (interface for new features)
  - Isolated for risk (unknown algorithms)
  - Bottom-up approach to finding components
- Shared data structures should be in a single component
  - But it should be represented as functionality (not directly accessible)

# Component Selection (cont.)




- Complex functionality should be in a single component
  - Complex algorithms as well
  - These are likely to change over time
- New features should be easy to add
  - Adding the feature changes only one component
  - The feature might be added as its own component
    - That should fit into the overall design
  - If not a single component, then a small set of components
    - Front end + back end on web application
    - No potential feature should affect a large set of components
- Components can be assignable to team members
  - Either individually or in small groups
  - Inner workings do not affect the rest of the system

# Component Description



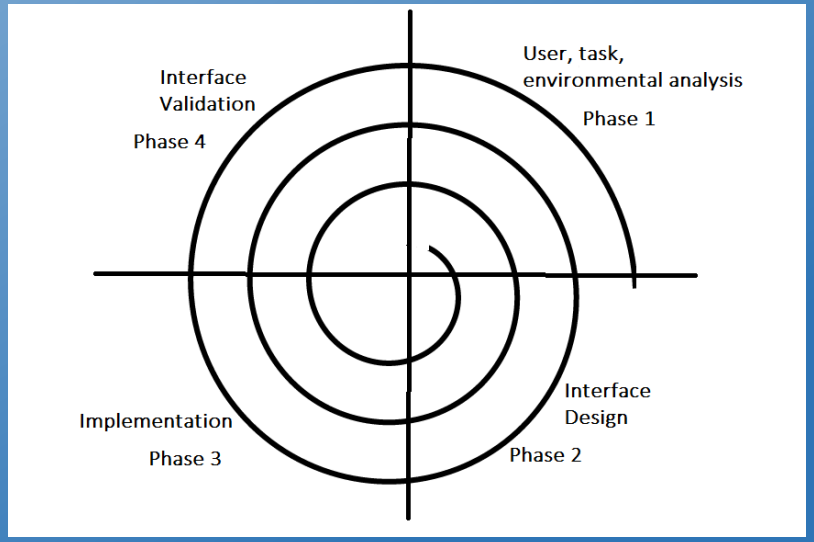

- Goal and purpose of each component
  - Name
  - Single short phrase or sentence
    - Clear, meaningful
    - Avoid ANDs (two components), etc.
- Once you know the components you can define them in detail
- What is important is the component **interface**
  - How the component interacts with other components
    - What it provides; what it requires
  - High level design means **defining interfaces**
    - As well as identifying the components
  - The component implementations are mere details
    - As long as we are satisfied that the implementation can be done
  - The interfaces are the design

# Interfaces

- High Level Design is the design of Interfaces
- Each component needs an interface
  - How it is used by other components
  - What it can and cannot do
- The set of component interfaces is the high-level design
- Concentrate on the interfaces before implementation
  - Both in what is provided
  - And in what is needed by others
  - Have a complete set of interfaces before doing coding

# Interface Goals

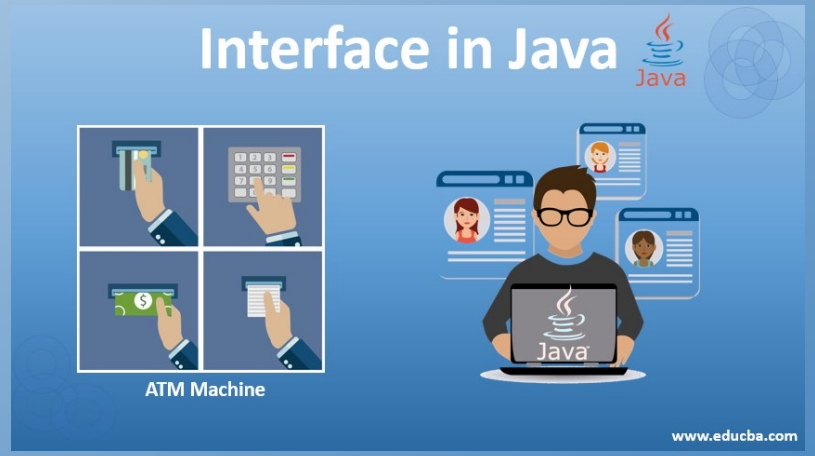
- Provide a concrete definition of the component
  - Understanding of what is needed and what is provided
- Enable others to use the component
  - Without knowing its internals
  - Develop code even before component is available
  - Develop test cases
  - Write a mocking library to emulate component
- Ensure the design is correct
  - Ensure you can implement each interface function
  - Easy to change an interface while doing design
  - Finish the interfaces before coding
  - Check that all specifications can be met
  - Ensure that other components have all they need



# Interface Goals

- "Interface" is defined loosely
  - Can be interface class, set of calls, a set of messages, command line options, RESTful urls, …
  - Can be bi-directional
    - Often includes callbacks to offer functionality
- Interface Definitions
  - Signatures with meaningful names and types
    - For each method, function, message
  - Includes descriptions of functionality
  - Includes error handling
    - Exceptions, what happens if …

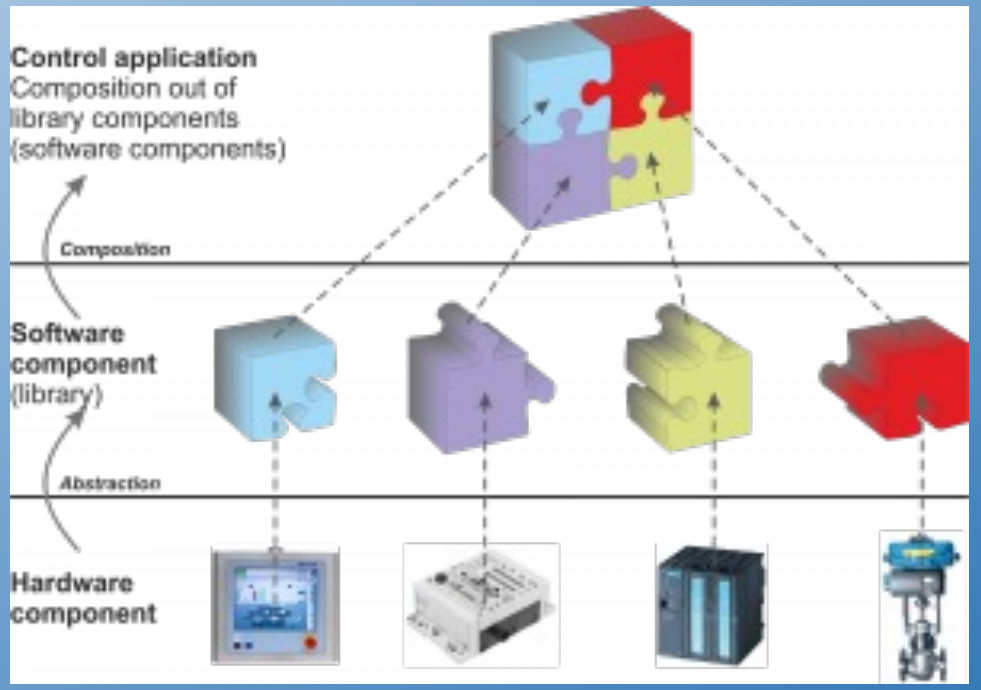
# Interface Design




- Provide the needed functionality
- Keep it as simple as possible
  - Single interface class, possibly with inner interfaces
    - Shouldn’t be a large set of classes
    - Shouldn’t be a hierarchy (these are represented by the root)
  - Small set of methods or functions or messages
    - Minimum parameters, simple types
    - Not fields or variables
    - Minimize constraints on ordering, call sequences, etc.
- Provide room for expansion
  - Identify possible future classes/methods/messages
    - What is going to be needed for evolution and maintenance
  - Its okay to define interfaces that won't be implemented right away
  - Its okay to include low priority functions that won’t be implemented right away

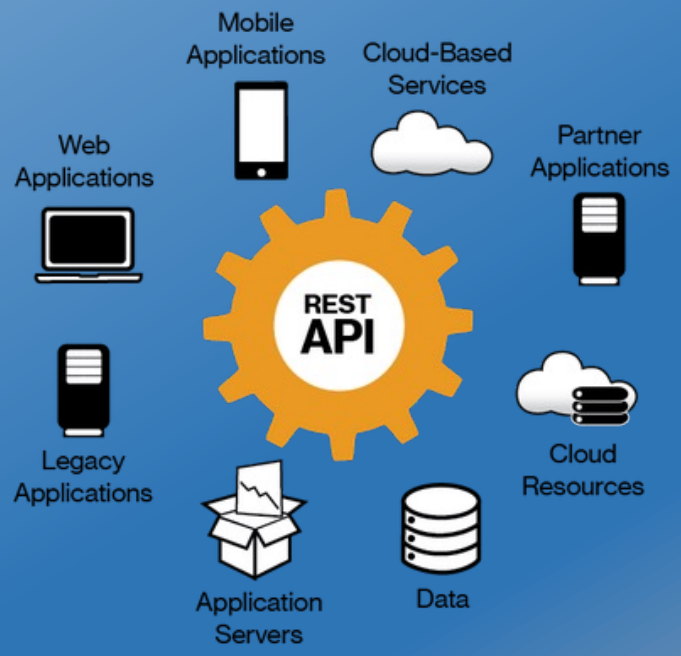
# Interface Design

- Document the interface
  - Provide a description of each element
    - Parameters, results, what it does
  - Provide constraints
    - What is expected of the inputs
    - What must be done before the element is invoked
    - What outputs are given under what circumstances
  - Include error handling
    - What happens if inputs don't match constraints
    - What exceptions can be thrown
    - What happens if remote server fails
- The interface will change
  - Implementation will require changes, additions, deletions
    - Negotiations between implementers and clients
  - Needs of both can change over time
  - Interfaces will get more complex over time
  - Changes may require work in other components (avoid)



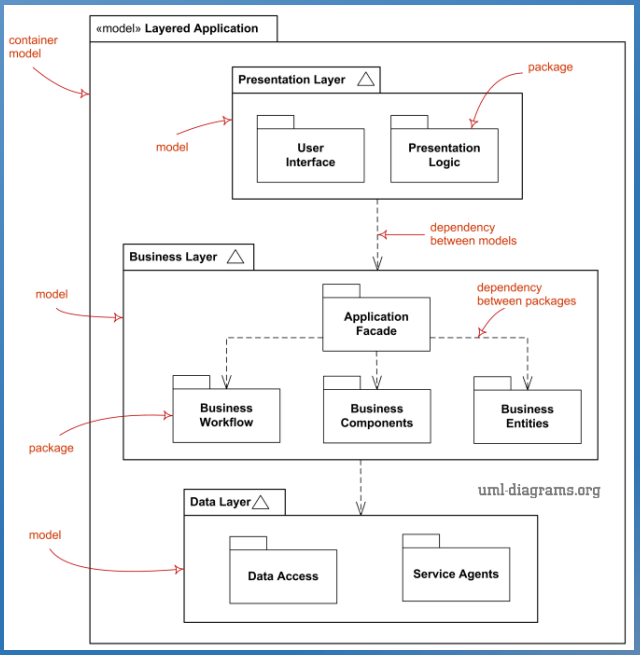
# Representing a High-Level Design

- Goals
  - Define the components and their interfaces
  - Represent these without doing the implementation
  - Provide a basis for detailed design and coding
- API-based Design (for each component)
  - Application Program Interface
    - Defines calls and requests
    - Defines data formats
    - Defines conventions, call orders, ..
    - Defines callbacks
  - This is what we need: how to represent it?



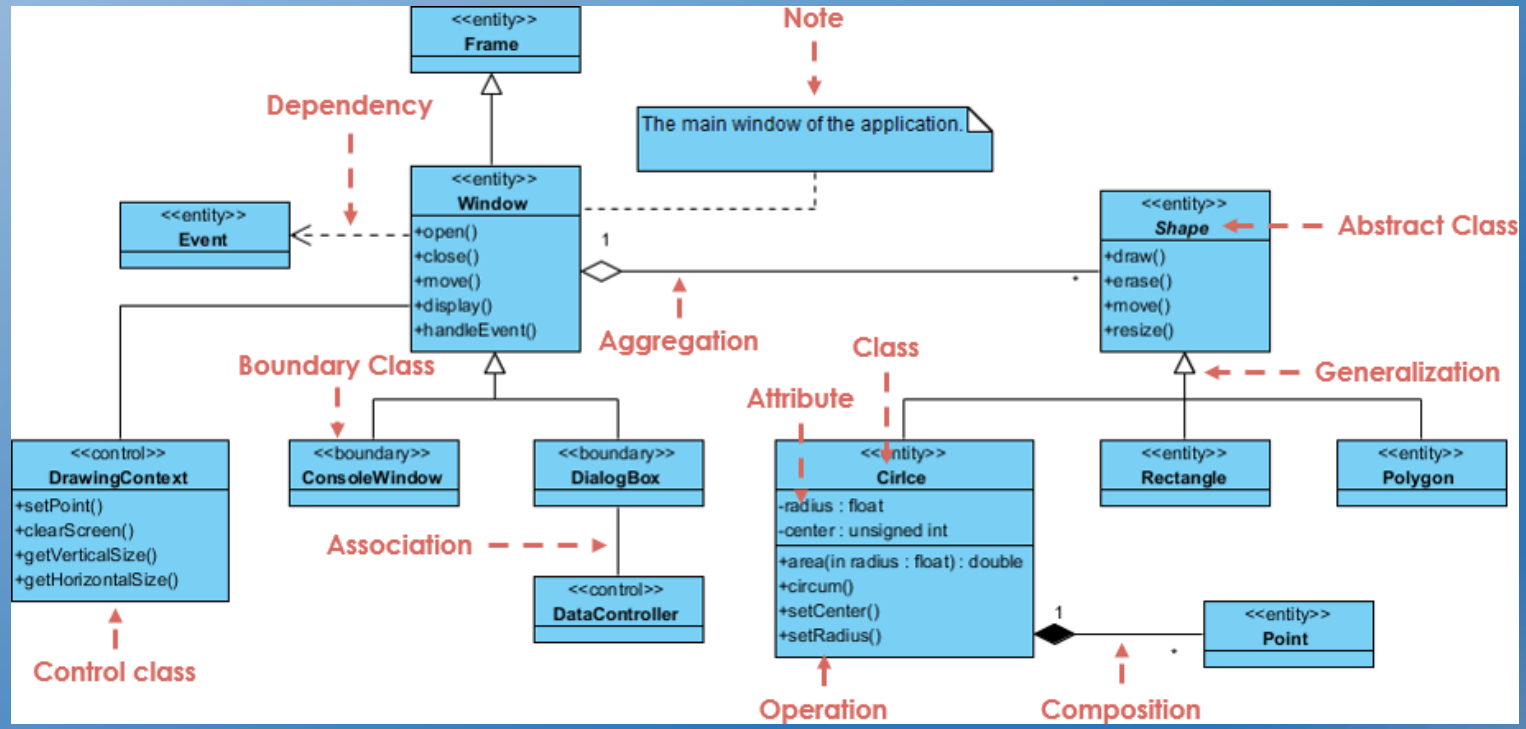
# UML-Based Design Representation

- UML class diagrams can be used to represent a design
  - Components can be represented as classes
  - Components can be represented as packages
  - Methods in the classes represent the interface
    - Might not actually be methods
  - Links represent potential component interactions
- These are language independent
  - UML does not commit to a language

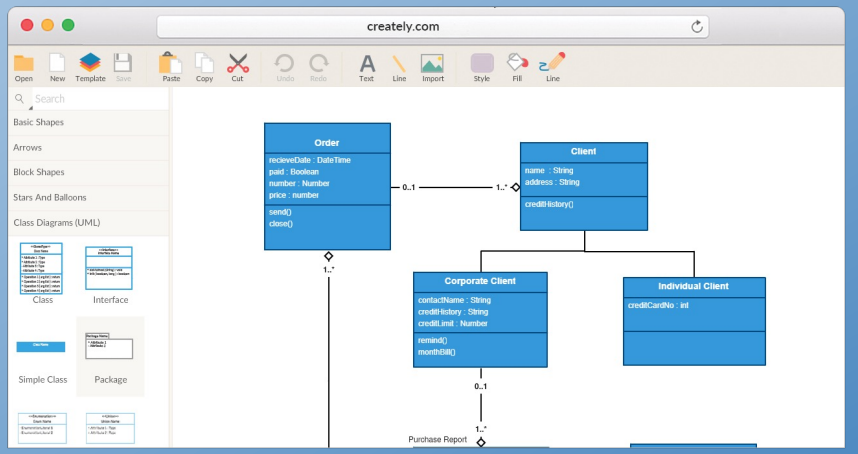


# UML Class Diagram Basics

- Classes
  - Name, attributes, operations
- Inheritance links
  - Generalization
- Dependency links
  - Associations
  - Aggregations
  - Dependencies



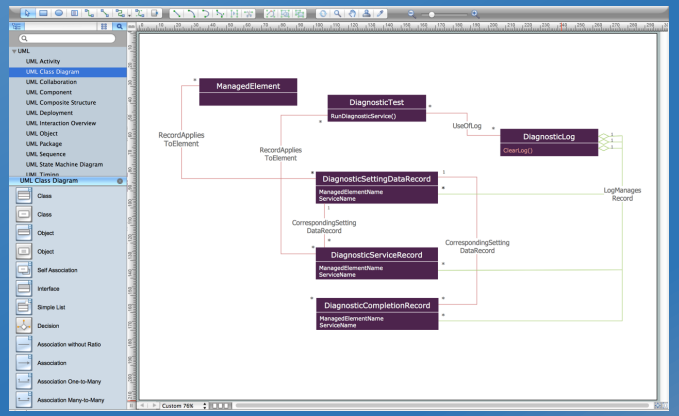
# UML for High-Level Design




- Meaningful information
  - Fields can give a sense of the class (but won't be used)
  - Links imply usage connections, not contained data
  - Basic methods provide the interface definition
- Diagram in levels
  - Keep size of each level small (5-10 classes)
    - Then use a separate level to define those classes
    - Facades, interfaces represent a level
  - Keep the diagrams simple
  - Gives a better sense of the design than 100 interfaces
    - Easier to implement as well

# UML For High-Level Design

- UML diagram can be used to sketch out the design
  - Easier to change then a text or code file
  - Easier to move things around for grouping, organization
- Start with all possible component candidates
  - Group these in a logical fashion & eliminate overlaps
    - Hierarchies represented by their root or interface
    - Choose one set where things overlap
    - Common elements merged using a façade
    - Internal components removed
  - Continue until you have a small number left



# System to Design

- I want to control my HO trains from my computer
  - Creating hardware
    - Embedded sensors in tracks
    - WIFI control of switches, signals
  - Using hardware
    - WIFI control of engines
- Want to direct trains to follow defined path
- Want to ensure safety
  - Avoid collisions, derailments
- Want a current display of everything
  - Want detailed control from the computer
- Smart HO Railroad Environment (SHORE)

# Software Architecture for SHORE

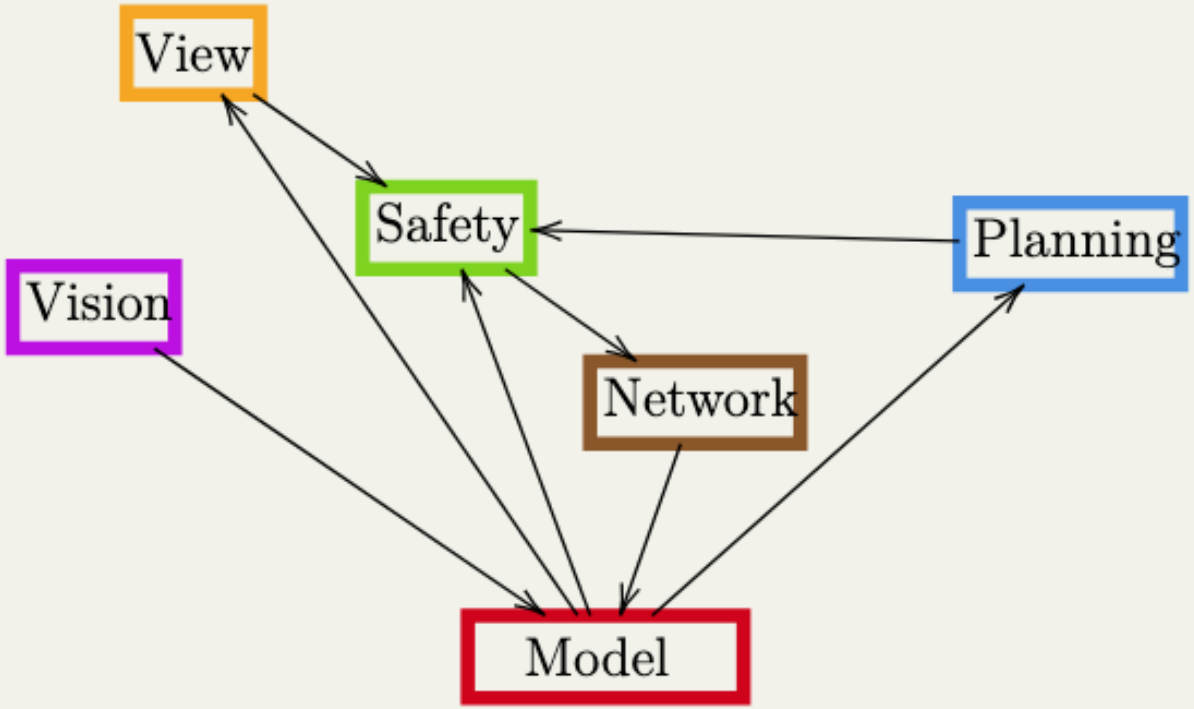
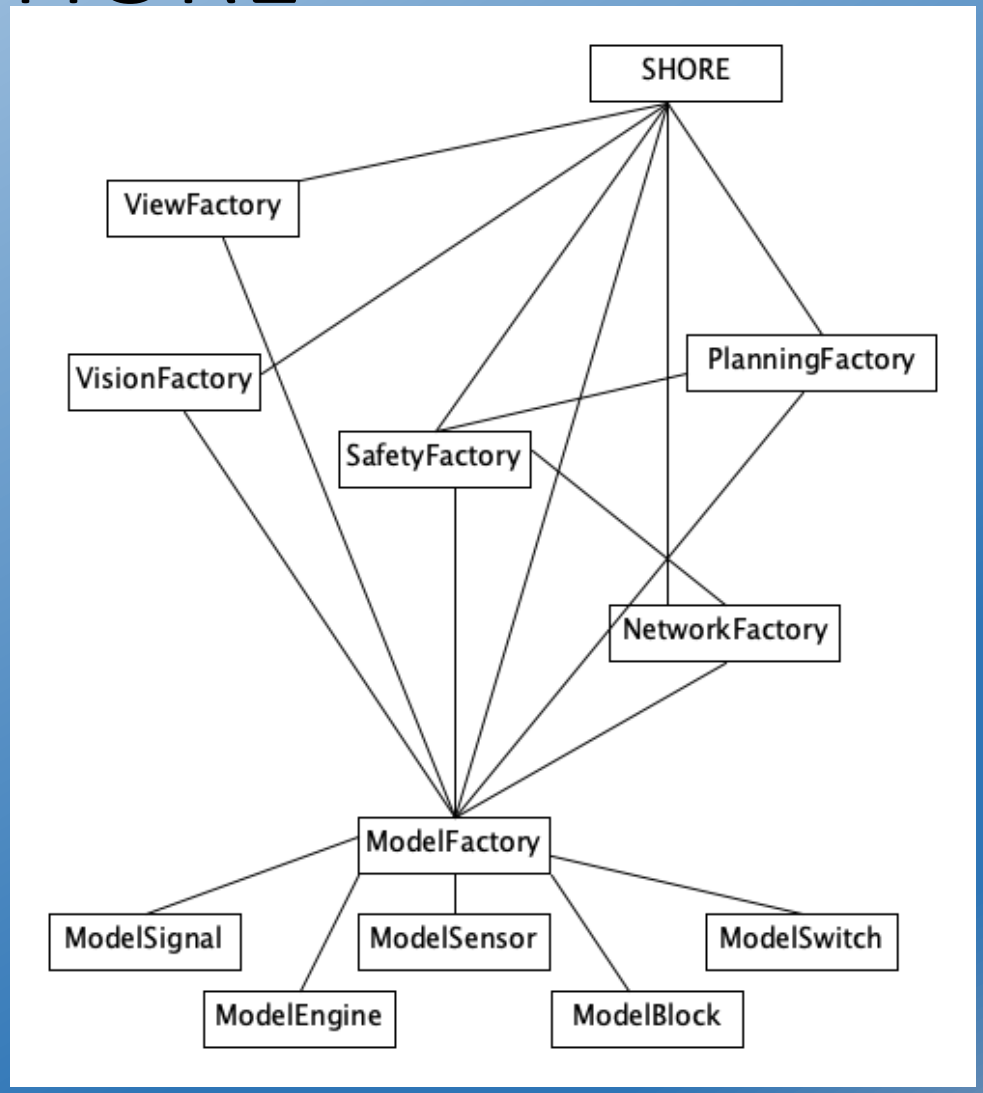


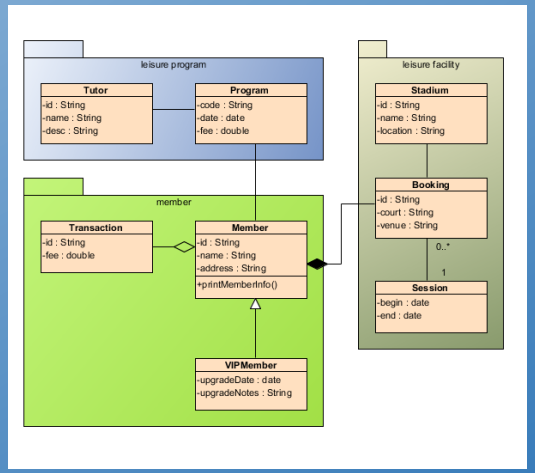

Figure 5.7: The architecture of Shore

# UML Design Diagram for SHORE

- Added Control component SHORE
- Create Façade Components
  - View, Vision, Safety, Planning, Network, Model
- Added classes for model data needed by others
  - Signal, Engine, Sensor, Block, Switch
- Note this doesn't include interfaces
  - But these could be added



# UML-Based Design Representation




- Advantages
  - Useful for initial exploration
  - Can use graphic editor to explore possibilities
  - Result is a nice visual representation of the system
  - Can be done on paper or using UML editors
- Disadvantages
  - Adding methods/fields is messy
    - Editors; syntax
  - Probably won't have a complete interface
  - Not something one can compile against or work with
  - Won't be part of the actual system
    - Will get out of date as system evolves
    - Won't represent the current design

# Language-Based Design


- I prefer to work in a programming language
  - Probably the implementation language (not necessary)
  - Editors available
  - Syntax well understood
- Create a design that can be used in the implementation
  - Starting point for the implementation
  - Code against the interfaces
  - Design evolves with implementation – always up to date
- But still simple enough to be easy to change as part of design
  - Adding, removing, changing components
  - Adding, removing, changing methods
  - Can ”play” with it as we do with UML diagrams
- Multiple approaches to this
  - We’ll cover these in the next class

# Language-Based Design

- Pros
  - You can use it as the start of the implementation
    - It in the target language
    - When it is complete and ready
  - Code against the design directly
    - Becomes part of the system
    - Evolves with the system
  - Done using known editors, syntax, …
- Cons
  - You need to know the target language
    - We'll cover choice of language next time
  - Not all languages support clean interfaces
  - Not all interfaces are language oriented (e.g., messages, RESTful)

# PROJECT HOMEWORK

- Start thinking about the high-level design for your project
  - How would you break it up: what are the components
  - Try allocating specification items to components
  - Possibly use a UML diagram to play with possibilities
    - Install UMLet, argouml, umbrello
    - Web-based UML tool (visual pradigm, creately)
- GitHub repo should include
  - Requirements, specifications, architecture, code style
  - Simple use of GitHub Issues
- Project Presentations on Tuesday 10/8
  - 10 minute presentations (not too detailed)

# Programming Assignment Homework

- Update your programming assignment hand in
  - Using feedback from today's breakout session
  - Ensure it uses project code style
  - Ensure there is a header comment, meaningful names
  - Ensure naming conventions work
    - Name implies its kind and definition location
    - Easy to locate items in the file
- You might try creating a UML diagram for the assignment

# Further Reading

- UML: IBM UML Introduction
- Textbook, chapter 6