Architecture, Design, and UML

Principles of Embedded Software

with **Bruce Montgomery**

Covering today

- Project 2 review
- Installing MCUXpresso for KL25Z (& PC) development
- WBSes
- Architecture & UML (?)
- KL25Z Handouts

Learning Objectives

- Students will be able to…
  - Understand the architecture and design aspects of embedded systems
  - Use a variety of graphical approaches for refining designs
  - Consider and apply design practices for embedded systems

- Large part of the material here comes from the textbook, Chapter 2, Making Embedded Systems, White, 2011, O'Reilly

Architecture

- Design - Breaking a large problem into smaller manageable pieces – can be done in a modular manner – planning for how to proceed
- Architecture - Deciding what modules you need, what they look like and how they connect together
- Modular design is applicable to any non-trivial project
- Modules may contain sub-modules, lower levels of design

- Object-oriented design of encapsulated classes with single responsibility is another form of modular design
- Project Management is a plan (or a design) for how a project might succeed

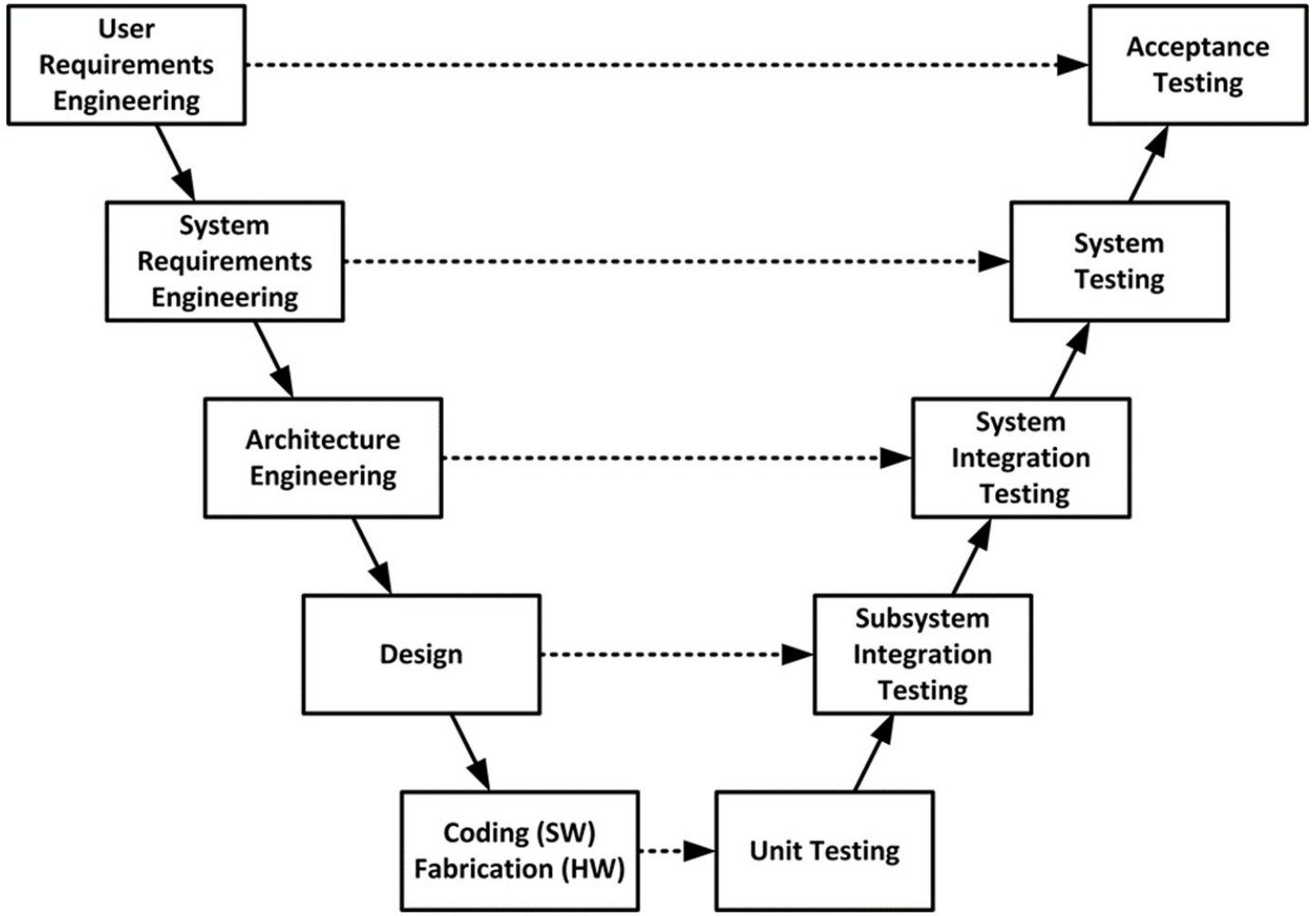
Where do projects originate?

- Improvements to existing/completed projects
- Feature requests
- Repurposing
- Me too
- Spinoffs
- New ideas
- Heroic inventor
- Skunkworks

- Different sources of projects may come with different levels of specifications to drive the design
- Some projects require more discovery than development
- All are best done iteratively, building on what came before

Progressive Elaboration

- Each level of design takes us from concept, to specifications, to implementation details



https://insights.sei.cmu.edu/sei_blog/2013/11/using-v-models-for-testing.html

Creating a System Architecture

- System architecture view can have many perspectives
- General hardware elements and their functions
- Design can be
  - from hardware up to system and software details or
  - from the system specification that leads to needed hardware

- Ideally, in our designs we want to start at a conceptual level, then move to specifications, and then to implementation details
  - Help prevent premature decisions and optimization
  - The longer we can stay at concept levels, the easier the design will be to mold and change
  - Once more detail is added change becomes harder, and once code and hardware arrive, change is harder still

System Diagrams

The White textbook recommends

- Architecture block diagrams
- Hierarchy of control organization chart
- Software layer view

- Use these elements to get a system view, identify dependencies and constraints, and consider new feature designs

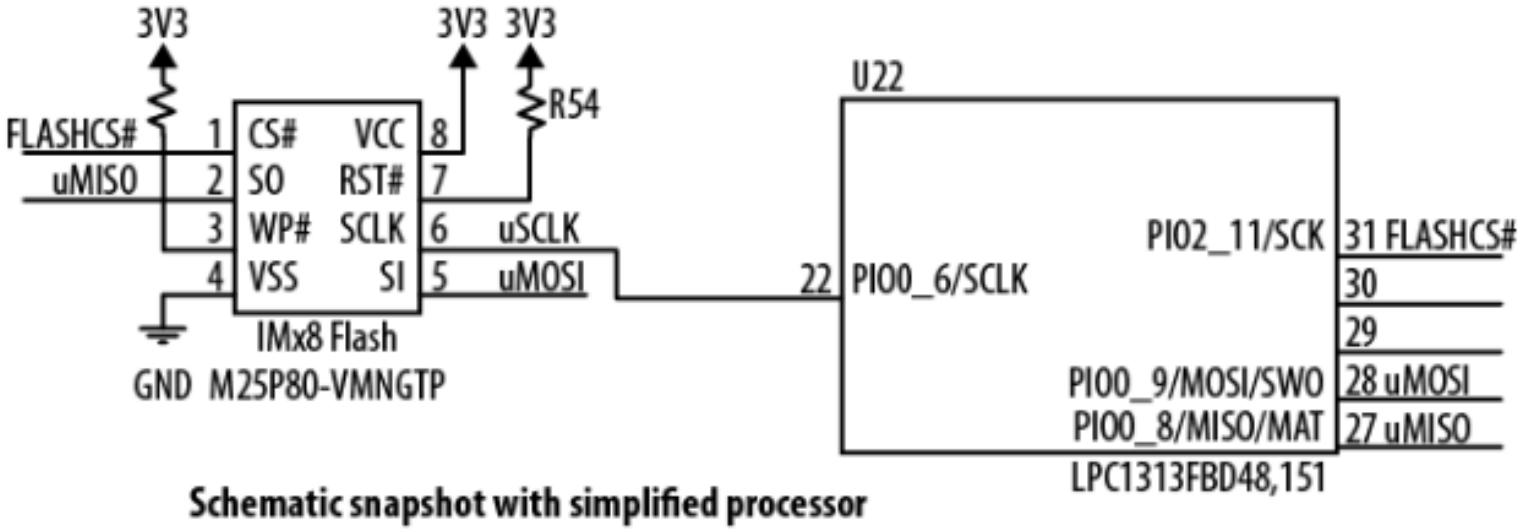
I would include UML diagrams, specifically:

- UML Activity diagrams
- UML State diagrams
- UML Sequence diagrams
- UML Use Case diagrams

Consider designs as object oriented

White recommends considering elements attached to microprocessors as objects, connections as potential communication methods

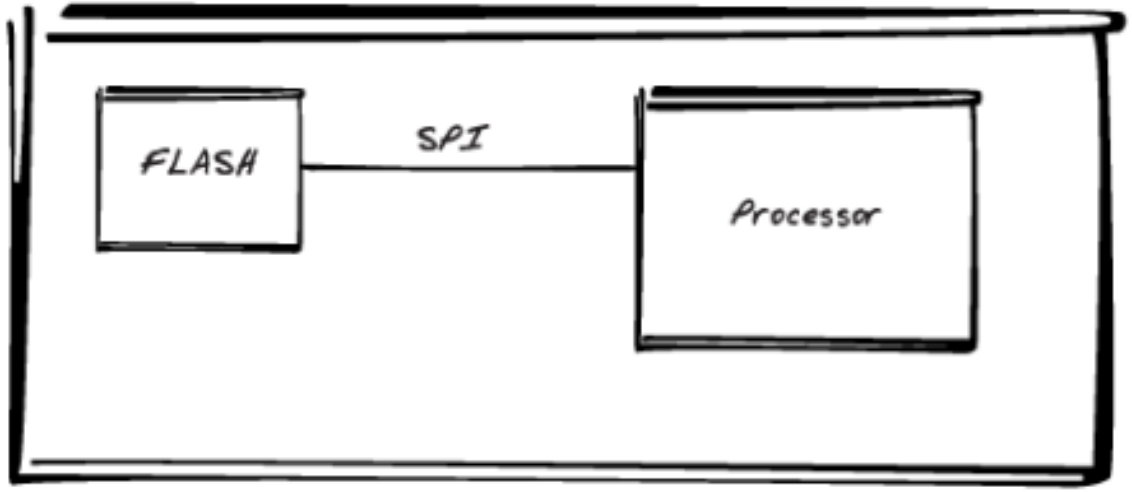
So a hardware diagram showing flash and a processor…



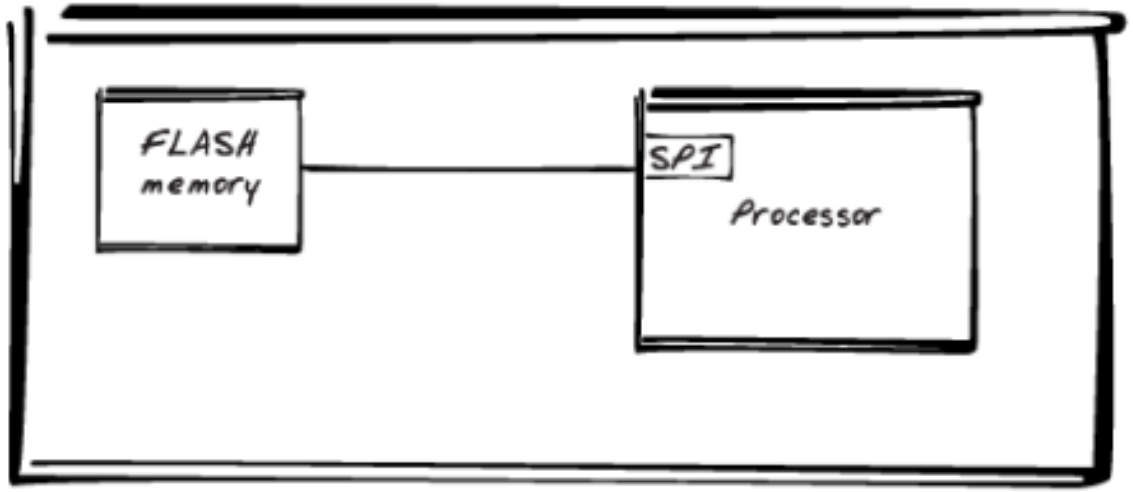
Schematic snapshot with simplified processor

Objects as blocks

…becomes **architecture block diagrams**. These would be analogous to UML Class/Object diagrams where you're identifying the elements that talk to each other – how they talk – and how they are connected…



Hardware block diagram



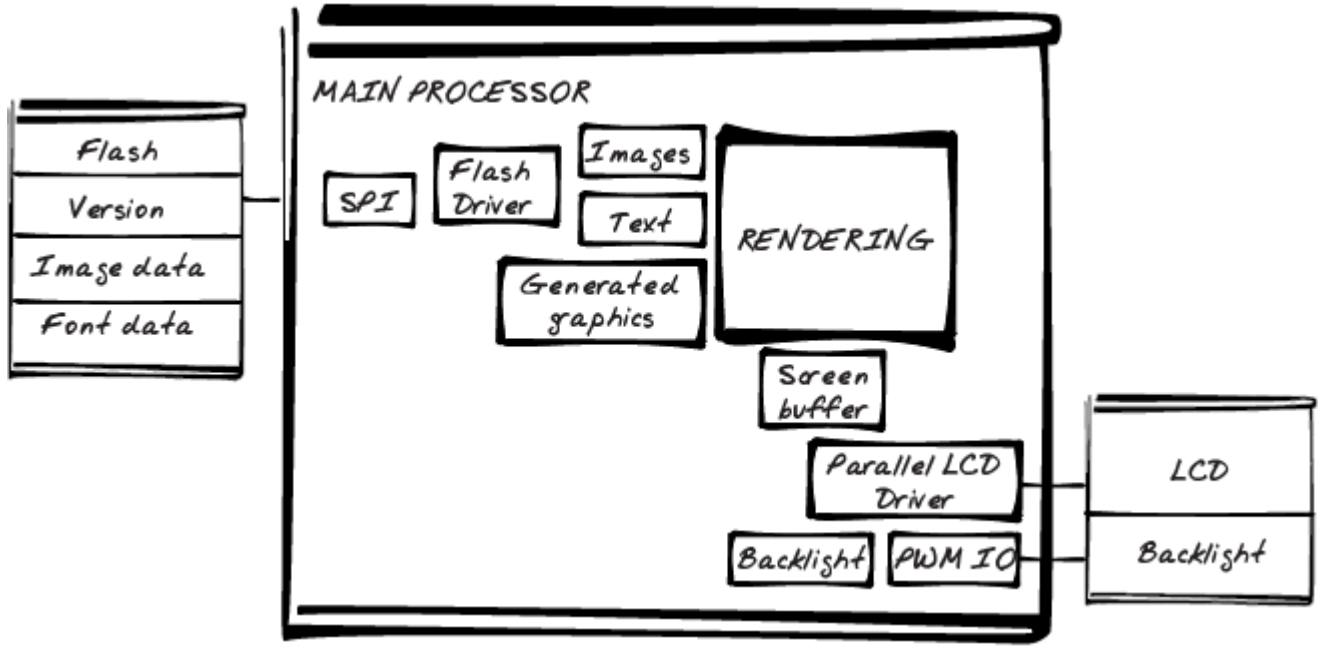
Software architecture block diagram

Adding details to block diagrams

Represent all the elements of the system – including supporting software structures. Looking at the view will help identify bottlenecks, requirements that are unclear, or issues in using the platform for the application.

Looking for different perspectives on a design, and it's often good to have different eyes involved as well.

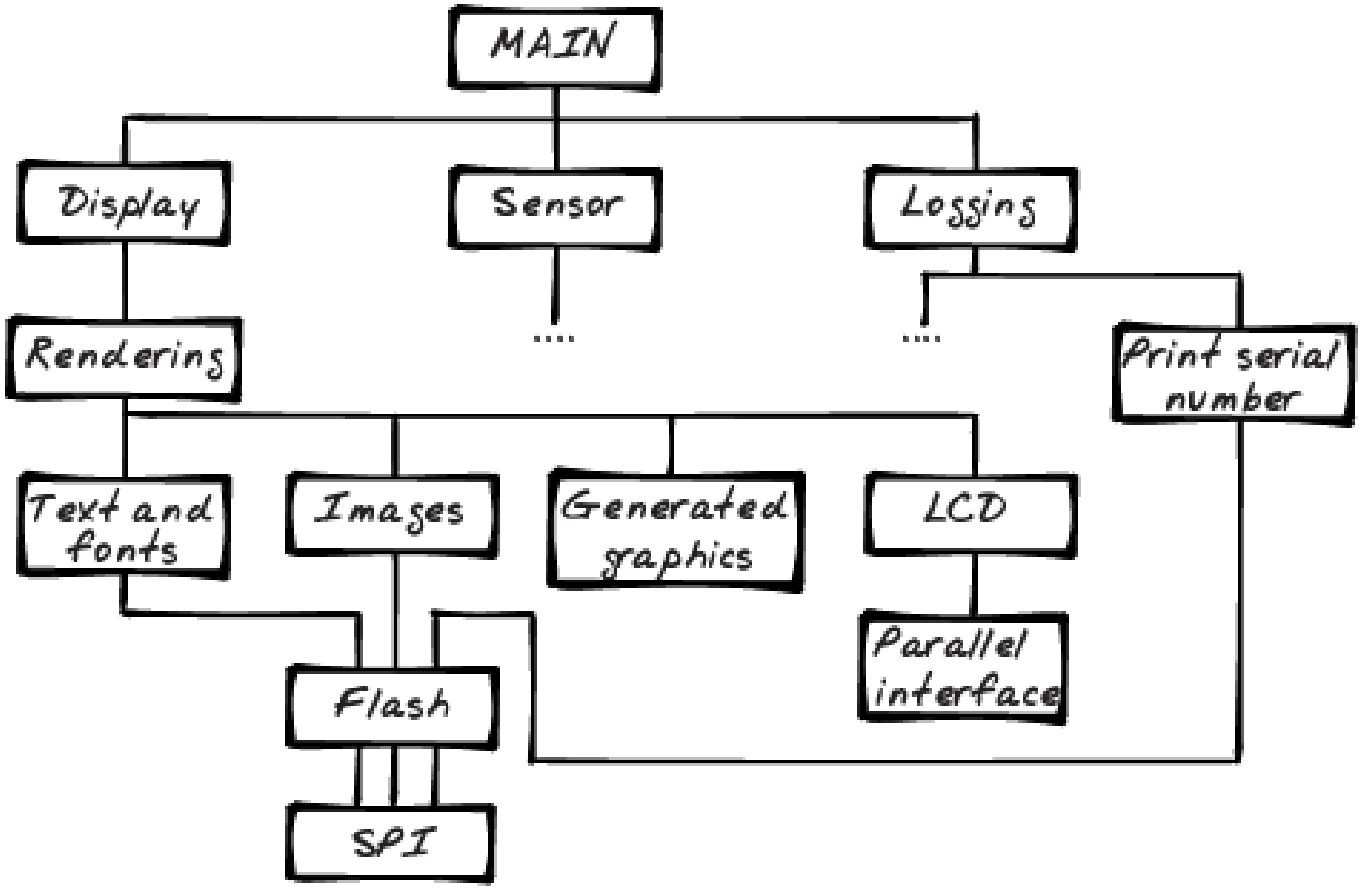
Iterating through designs is really improving a design – more modular, more cohesive, less coupled



Hierarchy of Control Diagram

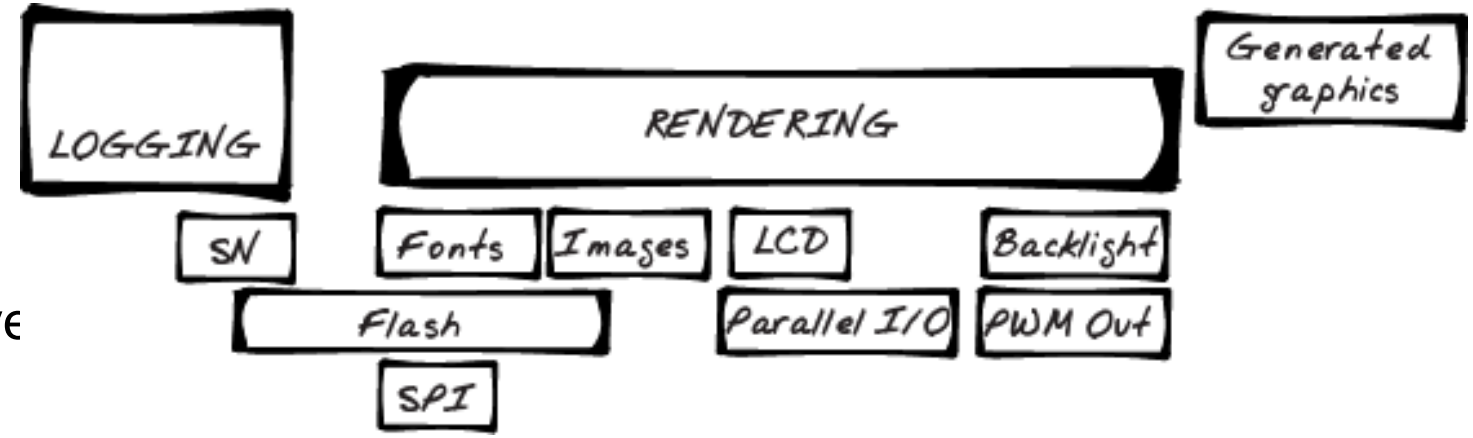
Starting with main, what are the discrete components in the application and how do they talk to each other?

As you walk through the design, you may realize that some system elements are shared by multiple processes, and you can begin to consider how you'll deal with that – threads, critical sections, etc.



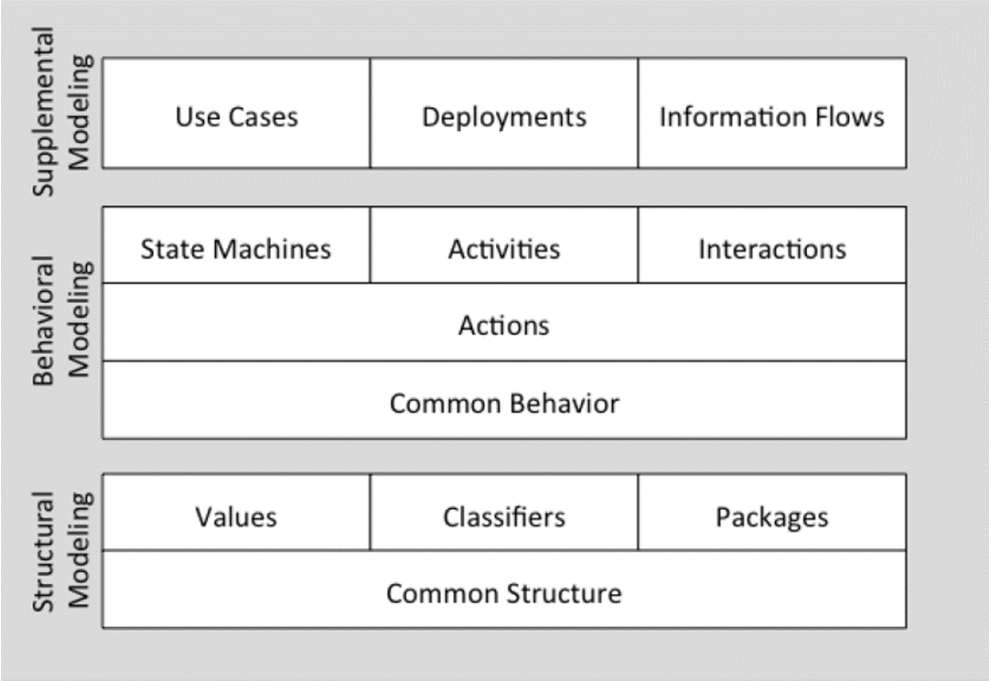
Software Layer View

Starting at the bottom – consider elements that come directly from the processor. Then add items that use those elements. Objects should be larger if they are more complex or have more connections. Build up to the higher level functions your application will provide.



UML?

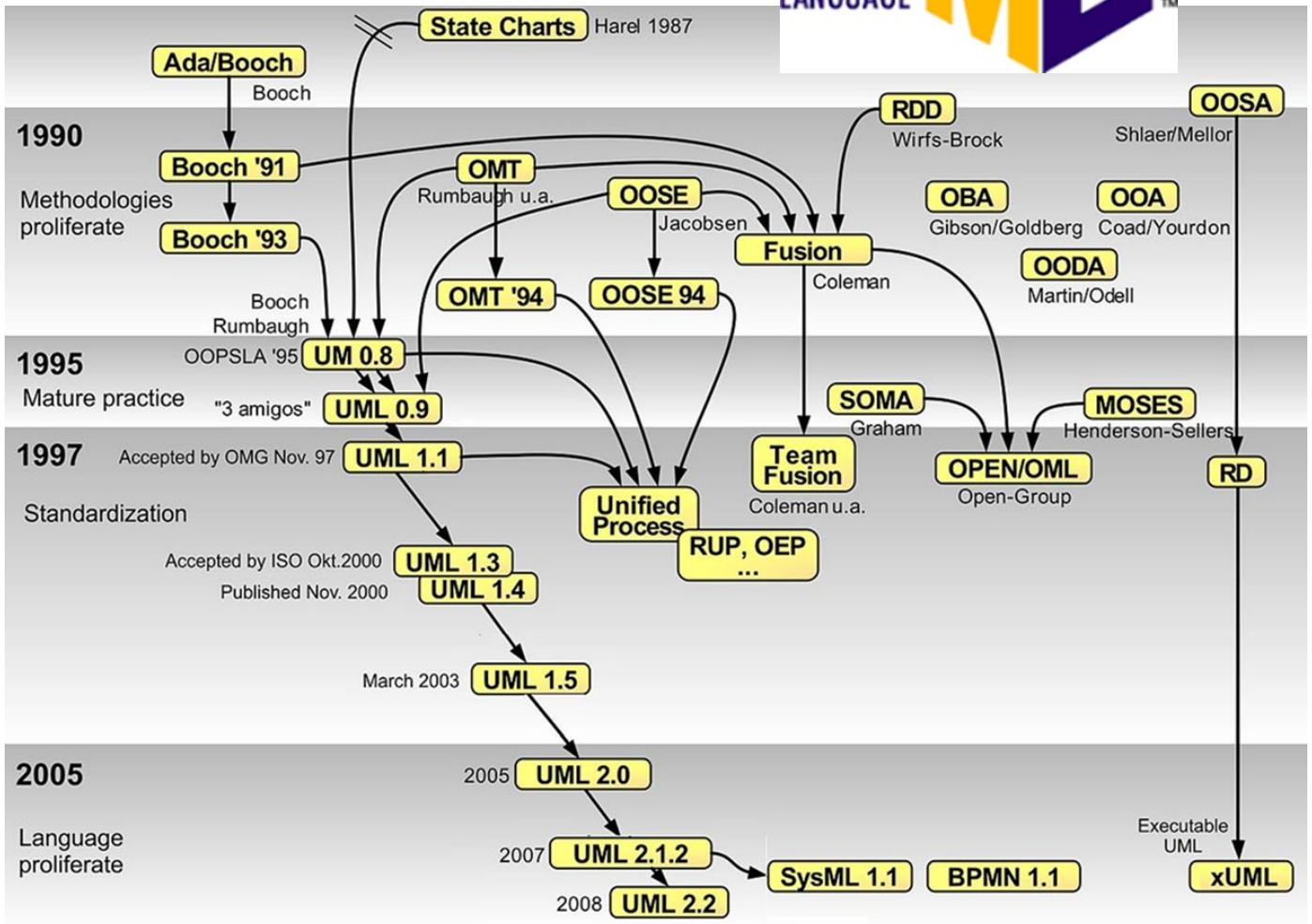
- UML is short for Unified Modeling Language
- The UML defines a standard set of notations for use in modeling systems
- Diagrams from the current UML release (https://www.omg.org/spec/UML/2.5.1/PDF)



- Structural
  - **Class**
  - Object
  - Package
  - Model
  - Composite Structure
  - Internal Structure
  - Collaboration Use
  - Component
  - Manifestation
  - Network Architecture
  - Profile
- Supplemental (both structural and behavioral elements)
  - **Use Case**
  - Information Flow
  - Deployment
- Behavior
  - **Activity**
  - **Sequence**
  - **State (Machine)**
  - Behavioral State Machine
  - Protocol State Machine
  - Interaction
  - Communication (was Collaboration)
  - Timing
  - Interaction Overview

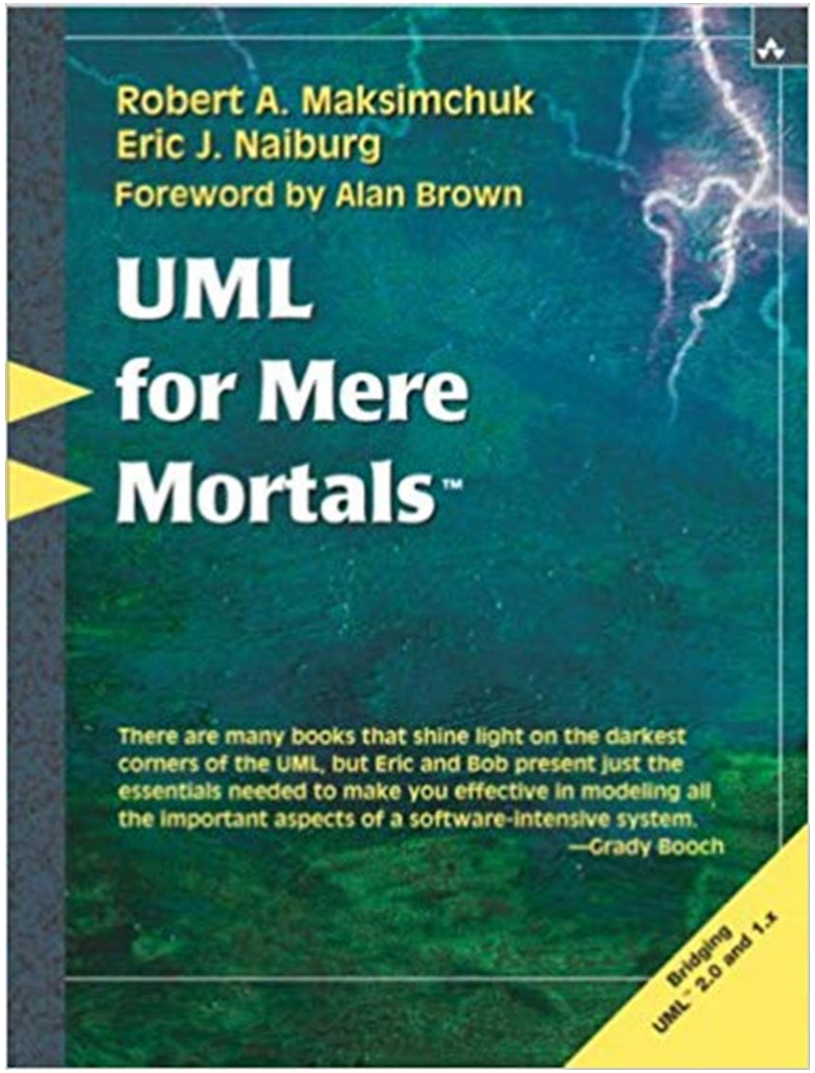
Brief History of the UML

- In the 80s and early 90s, there were multiple OO A&D approaches (each with their own notation) available
- Three of the most popular approaches came from
- James Rumbaugh: OMT (Object Modeling Technique)
- Ivar Jacobson: Wrote “OO Software Engineering” book
- Grady Booch: Booch method of OO A&D
- In the mid-90’s all three were hired by Rational and together developed the UML; known collectively as the “three amigos”
- Latest UML 2.5.1 Dec 2017 https://www.omg.org/spec/UML/



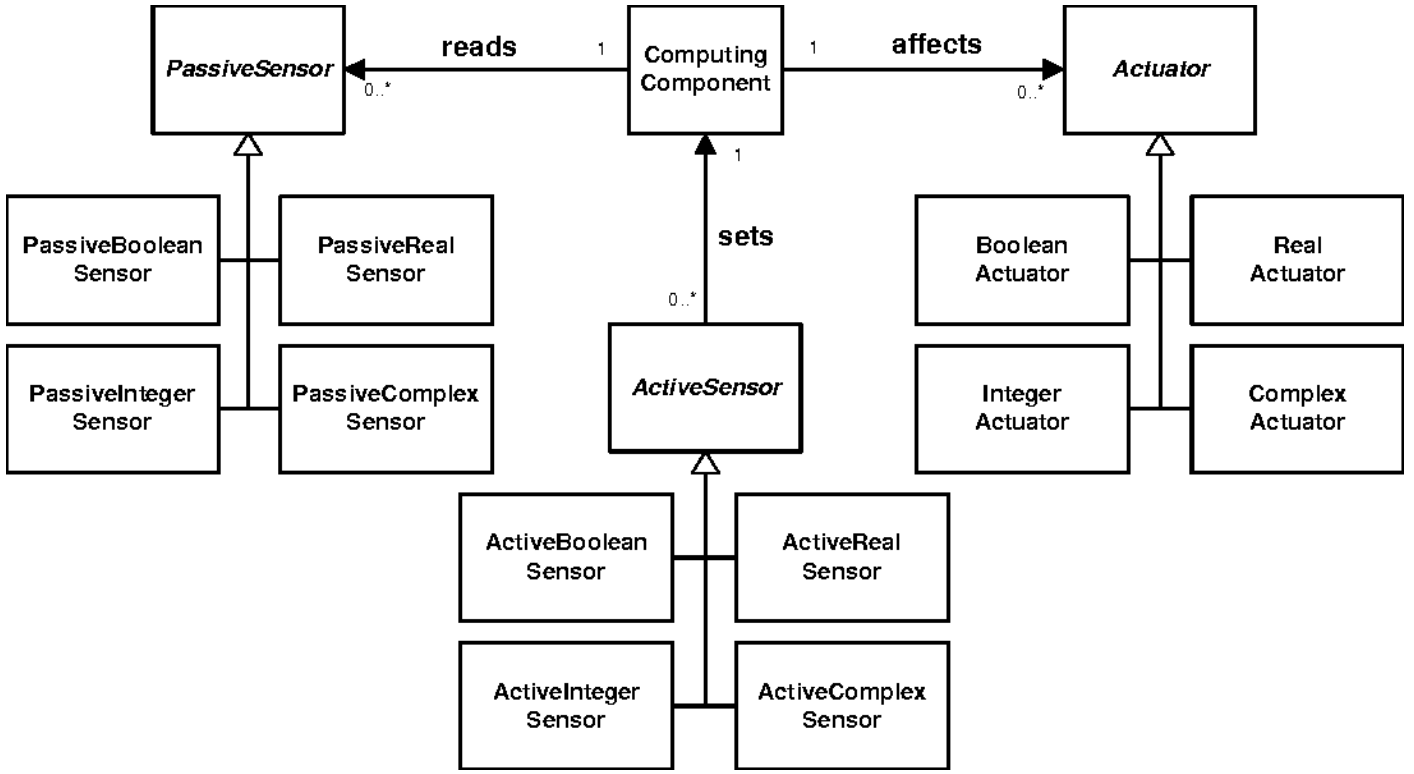
UML Tools & References

- Tutorials
  - https://www.tutorialspoint.com/uml/index.htm
- Book
  - UML for Mere Mortals, Maksimchuk & Naiburg, 2005, Addison Wesley
- Tools
  - Draw.io – has UML tools (Free!)
  - Lucidchart.com – UML Templates
    - (Free access available)
  - TopCoder UML Tool
    - sequence, class, use case, and activity diagrams
    - Free - Requires registration
    - https://www.topcoder.com/tc?module=Static&d1=dev&d2=umltool&d3=description
  - ArgoUML – open source
    - http://argouml.tigris.org/
  - Visio
  - Whiteboards and a phone/camera
  - Paper & pencil



UML Class Diagrams

- UML Class Diagrams are usually used for describing an object-oriented language with classes that contain methods and attributes
- However, for a functional language like C, they could still be used to show modules and how they are associated
- Could be a good alternative to the hierarchy of control model shown earlier…



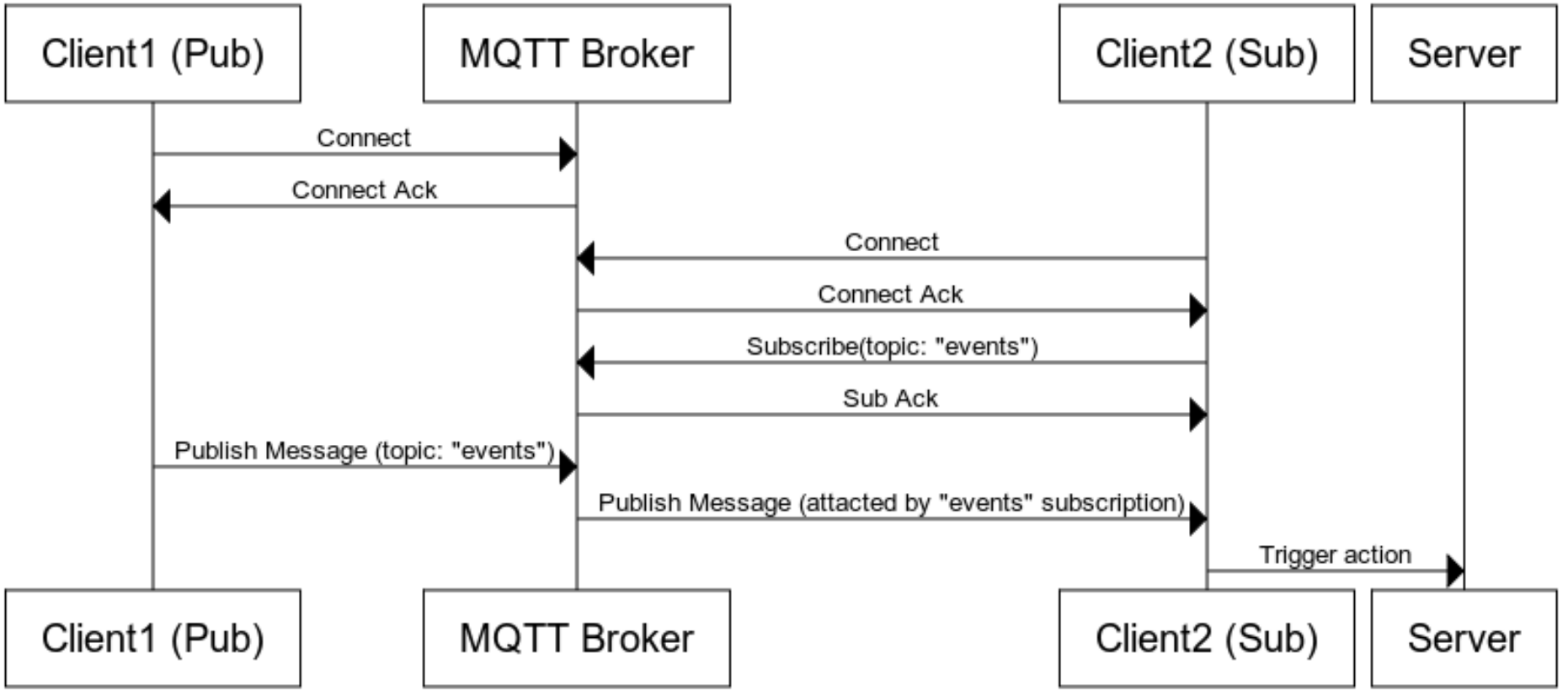
https://www.semanticscholar.org/paper/Requirements-patterns-for-embedded-systems-Konrad-Cheng/b9804373df10fa6c7d5a465330fe1b56c53c8ff5

UML Sequence Diagrams

- These are a common tool for embedded systems dependent on timing and message passing between elements
- Objects or modules are shown across the top of the diagram
- Each object/module has a vertical dashed line known as its lifeline
- When a module is active, the lifeline has a rectangle placed above its lifeline
- If a module closes during the scenario, its lifeline terminates with an “X”
- Messages between modules are shown with lines pointing at the object receiving the message
- The line is labeled with the function being called and (optionally) its parameters
- All UML diagrams can be annotated with “notes”
- Sequence diagrams can be useful, but they are also labor intensive
- May need to document both normal and off-normal situations

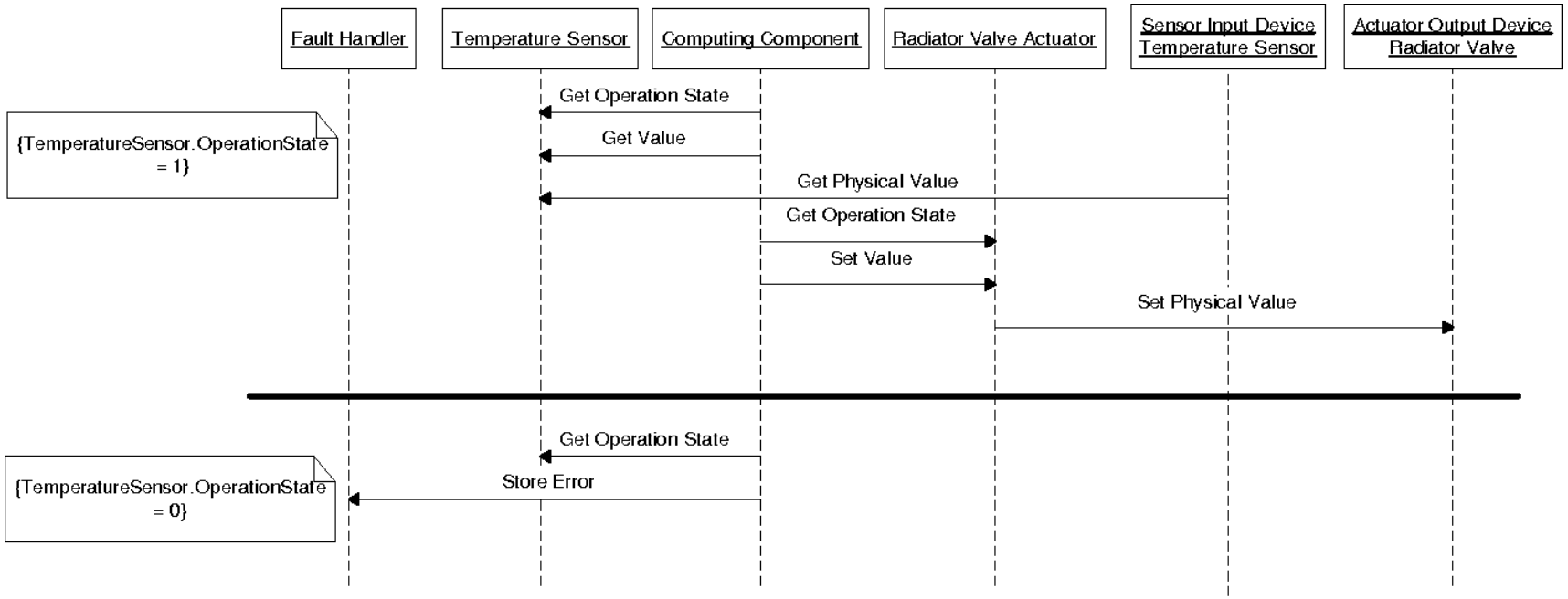
UML Sequence Diagrams

- Example of elements of an MQTT system communicating and passing messages
- From an article on Node.JS publishing events to an MQTT broker
- https://stackoverflow.com/questions/32538535/node-and-mqtt-do-something-on-message



UML Sequence Diagrams

- Example of sensors, components, and actuators communicating in a system
- https://www.semanticscholar.org/paper/Requirements-patterns-for-embedded-systems-Konrad-Cheng/b9804373df10fa6c7d5a465330fe1b56c53c8ff5/figure/2



User Perspective and Use Cases

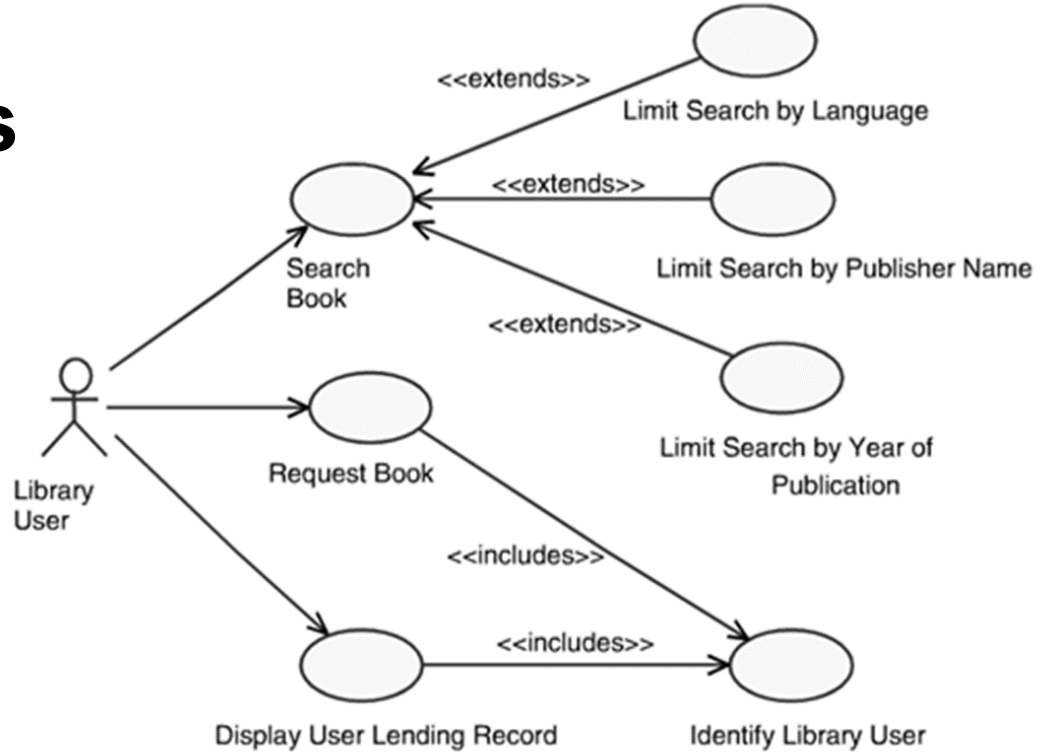
- Some embedded systems we develop will have direct interface with user interactions, in those cases we want to develop a clear understanding of the tasks the user is attempting to perform
- In UX oriented workflows, understanding the user and their tasks are key
  - A typical UX development process might include
    - Analysis and Planning
    - User and Task Research (<- Use cases)
    - Interface and Interaction Design
    - Verification and Validation
- Use cases help maintain the user perspective
  - We identify the different types of users for our system – “who”
  - We then develop tasks for each of the different types of user – “what”
- Use cases are used to capture functional requirements
  - They can be annotated to also describe non-functional requirements but typically the focus is on functional requirements

Actors

- More formally in a use case, a user is represented by an actor
  - Each use case can have one or more actors involved
- Actors have two defining characteristics
  - They are external to the system under design
  - They take initiative and interact with our system
  - During a use case, they have a goal they are trying to achieve
- Each use case describes a task or tasks for a particular actor
  - The description typically includes one “success” case and may include a number of extensions that document “exceptional” conditions

UML Use Cases – Best Practices

- UML Use Cases include Actors and Tasks
- Always design use cases from the actor's point of view
- Model the entire flow of a given operation
- For most systems, use cases should number in the tens, not hundreds
- <include> cases: not optional, base use case not complete without it, not conditional, and doesn't change the base use case behavior
- <extend> cases: Can be optional, not part of base use case, can be conditional or change behavior



WAVE Test for Use Cases (from Maksimchuk)
W: Use case describes WHAT to do, not how
A: ACTOR'S point of view
V: Has VALUE for actor
E: Use case models ENTIRE scenario

UML Activity and State Diagrams

- They represent alternate ways to record/capture design information about your system
- They can help you identify new functions and modules
- They are typically used after use case creation: for instance, create an activity diagram for a given use case scenario
- For each activity in the diagram, (you might) follow-on and draw a sequence diagram
  - Add a module for each object in the sequence diagrams to your class diagram, add functions in sequence diagrams
  - Sequence diagrams may not needed for simple logic

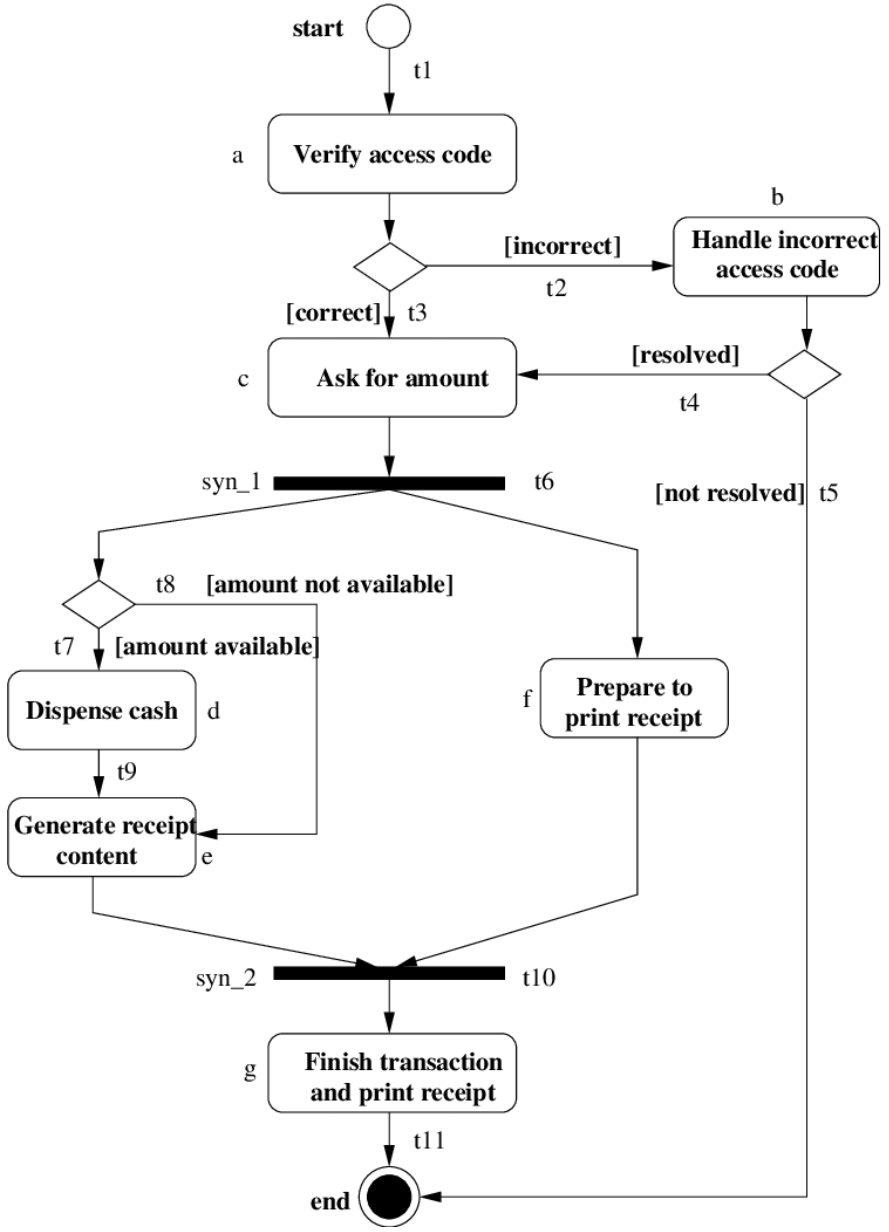
UML Activity and State Diagrams

- Activity Diagram
  - Think “Flow Chart on Steroids”
  - Able to model complex, parallel processes with multiple ending conditions
  - Focus is on verbs (actions)
- State Diagram
  - Shows the major states of modules or the system
  - partition behavior into various categories (initializing, acquiring info, performing calcs, …)
  - documents these states and the transitions between them (transitions typically map to function calls)
  - Focus is on nouns (states)

UML Activity Diagram

Notation

- Initial Node (circle)/Final Node (circle in circle)/Early Termination Node (circle with x through it)
- Activity: Rounded Rectangle indication an action of some sort either by a system or by a user
- Flow: directed lines between activities and/or other constructs
- Flows can be annotated with guards “[clause]” that restrict its use
- Fork/Join: Black bars that indicate activities that happen in parallel
- Decision/Merge: Diamonds used to indicate conditional logic.
- Example – an ATM machine
  - https://www.researchgate.net/figure/The-UML-activity-diagram-of-an-ATM_fig1_220201279



UML State Diagrams

- Each state appears as a rounded rectangle
- Arrows indicate state transitions
- Each transition has a name that indicates what triggers the transition (often times, this name corresponds to a method name)
- Each transition may optionally have a guard that indicates a condition that must be true before the transition can be followed
- A state diagram also has a start state and an end state
- It can have compound states for operations and links to other state diagrams

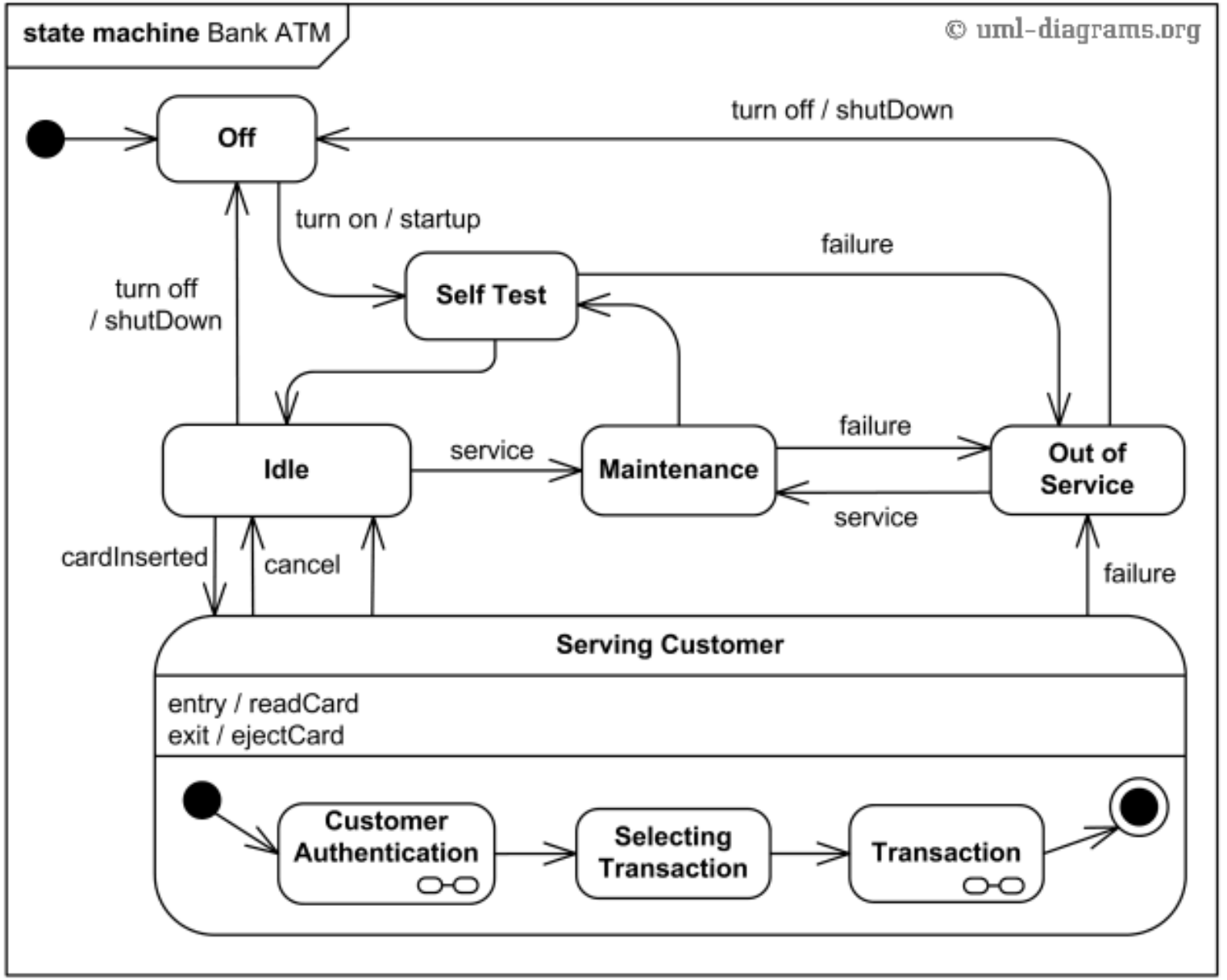
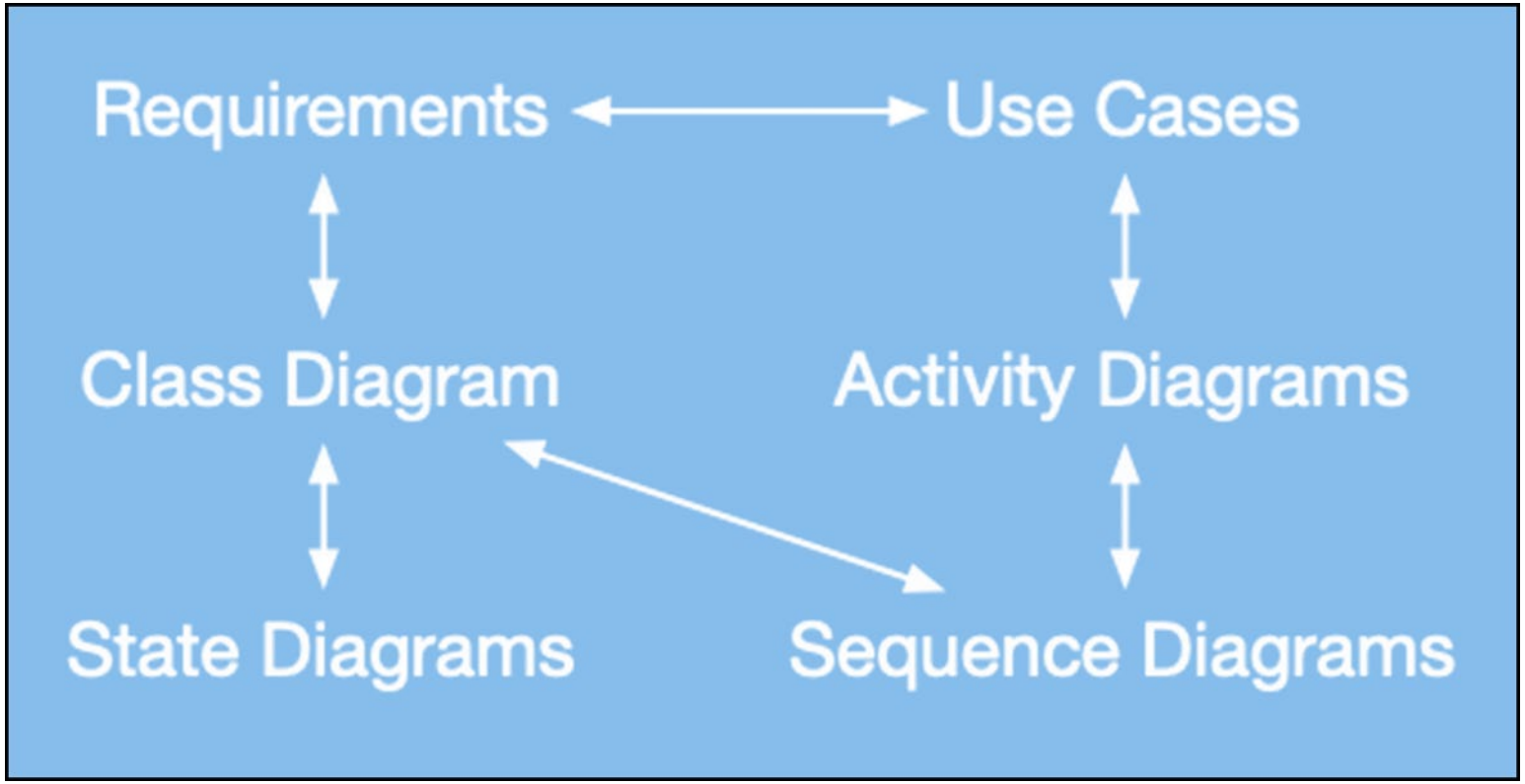


Diagram based iterative development

- Once you have written requirements and use cases to fulfill them
- and you've reviewed the use cases with clients to determine the various alternate paths
- You're ready to start creating class diagrams, activity diagrams, state diagrams and sequence diagrams using information in the use cases as inspiration
- Details are developed in iterative change and review



Steps to move from Diagrams to Architecture

- Encapsulate Modules
  - Do connected modules only depend on each other? Combine them?
  - Are the elements likely to vary or change over time? Separate them?
- Delegation of Tasks
  - Can you logically divide the modules into work that could be done by other people?
  - Even if you're the only one coding, building cohesive modules will ease your work
- Developing Interfaces
  - Can you identify what the interfaces between modules are?
  - Can higher-level interfaces be made to ease connections to complex code?
  - Can you drive towards loose and well defined coupling between modules?
  - Are there standard patterns for the interfaces in the system provided libraries or in your style guide or specifications?
- Using Common Design Patterns
  - Next Lecture…

Considering Driver Interfaces

- One typical design pattern for embedded interfaces is the one used for API calls in Unix systems (per the POSIX standard):
- Open, Close, Read, Write, IOCtl
  - Open - Opens the driver for use. Similar to (and sometimes replaced by) init
  - Close - Cleans up the driver, often so another subsystem can call open
  - Read - Reads data from the device.
  - Write - Sends data to the device.
  - IOCtl - Stands for input/output (I/O) control and handles features not covered by the other parts of the interface
- Similar to database standard transactions – CRUD interfaces
  - Create, Read, Update, Delete

Developing other interfaces

Typical calls for a logger:

- void Log(enum eLogSubSystem sys, enum eLogLevel level, char *msg);
- void LogWithNum(enum eLogSubSystem sys, enum eLogLevel level, char *msg, int number);
- void LogSetOutputLevel(enum eLogSubSystem sys, enum eLogLevel level)
- void LogGlobalOn();
- void LogGlobalOff();
- void LogVersion(struct sFirmwareVersion *v) // always a good practice to maintain
  // versions of modules or systems

- Interesting example of logging design in BSD syslog protocol – RFC 3164
  - https://tools.ietf.org/html/rfc3164

Summary

- Diagrams for design: Architecture Block Diagrams, Hierarchy of Control Diagrams, Software Layers
- UML Design Diagrams: Class, Sequence, Use Case, Activity, State
- From Diagrams to Architecture
  - Encapsulate Modules
  - Delegation of Tasks
  - Develop Interfaces
  - Using Common Design Patterns

Next Steps

- 2nd Quiz up now for Wednesday, new Quiz up Friday/Saturday
- Project 2 is starting
  - Initial WBS due Tuesday 2/11
  - Remainder of project 2 due Tuesday 2/18
- Today's architecture material comes from Chapter 2 of the White book if you'd like to read for more details
- For Thursday
  - Bring your PC and KL25Z with cable to class, with MCUXpresso installed
  - We'll walk through a quick tour and a test project together
- See me or the SAs for any help you need!