Global Variables Are Evil!

"Global variables are responsible for much undebuggable code, reentrancy problems, global warming, and male pattern baldness. Avoid them!"

— *Jack Ganssle*

Global Variables Are Evil!

- Anti-Patterns:
  - More than a few read/write globals
  - Globals shared between tasks/threads
  - Variables have larger scope than needed

- Global variables are visible everywhere:
  - Use of globals indicates poor modularity
    - Globals are prone to tricky bugs and race conditions
  - Local static variables are best if you need persistence
    - File static variables can be OK if used properly
    - Don't make procedures globally visible if not needed

Global vs. Static Variables

- Globals:

```
uint32_t gVar = 0;
void gProc(…)   { … }
```

- Global risks
  - Written from anywhere
    - Debugging: who wrote it?
  - Read from anywhere
    - Changes break everything
  - Multithreaded race conditions
  - Increased complexity
    - Data flow “spaghetti”

- **File Static:**

```
static uint32_t fsVar = 0;
static void fsProc(…)   { … }
```

  - Only inside .c file
  - Use with small .c files
  - Like C++ “private”

- **Local Static:**

```
void gProc(…)
{ static uint32_t sVar = 0;
… }
```

  - Persistent variable value
  - Can’t be seen outside procedure

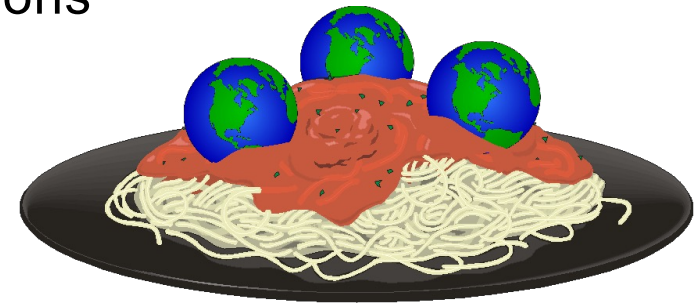
Avoiding And Removing Globals

- Define smallest scope possible (variables and procedures)
  - Change global to file static; file static to local static
- Arrange .c files based on access to data
  - Example: time of day updated by ISR
    - File static time of day variable in TimeOfDay.c
    - Put timer tick ISR in TimeOfDay.c
    - Put procedure to disable interrupts & read time of day in TimeOfDay.c
- Configuration values & constants
  - Use const keyword – prevents multiple writers
  - Read-only access to global configuration data structure
  - Limit visibility to need-to-know within relevant .h file

Best Practices For Avoiding Globals

- Use smallest practical scope for variables & procedures
  - Ideally, zero global variables
  - Use file static if you must; local static if you can
  - A good compiler will generate efficient code

- Reorganize code to reduce scope
  - Write anything except locking variables only in one place
  - File static variables for small groups of functions
    - More or less the idea of C++ private keyword
    - Take care of data locking when reading



- Global Variable Pitfalls
  - Lots of global variables is a sign of bad code

Example

https://betterembsw.blogspot.com/2013/09/getting-rid-of-global-variables.html

You have a "globals.c" file that defines a mess of globals, including:

```
int g_ErrCount;
```

which might be used to tally the number of run-time errors seen by the system. I've used a "g_" naming convention to emphasize that is a global, which means that every .c file in the program can read and write this variable with wild abandon.

Let's say you also have the following places this variable is referenced, including globals.c just mentioned:

- **globals.c:** `int g_ErrCount;  // define the variable`
- **globals.h:** `extern int g_ErrCount; // other files include this`
- **init.c:** `g_ErrCount = 0; // init when program starts`
- **moduleX.c:** `g_ErrCount++;  // tally another error`
- **moduleY.c:** `XVar = g_ErrCount; // get current number of errors`
- **moduleZ.c:** `g_ErrCount = 0;  // clear number of reported errors`

Create an Error Counting Module

Create separate "object" for error counting: ErrCount.c

- **globals.c:** // not needed any more for this variable
- **ErrCount.c:** int g_ErrCount; // define the variable
- **ErrCount.h:** extern int g_ErrCount; // other files include this
- **init.c:** g_ErrCount = 0; // init when program starts
- **moduleX.c:** g_ErrCount++; // tally another error
- **moduleY.c:** XVar = g_ErrCount; // get current number of errors
- **moduleZ.c:** g_ErrCount = 0; // clear number of reported errors

Initialize Where Defined

- **ErrCount.c:** int g_ErrCount = 0; // define and init variable
- **ErrCount.h:** extern int g_ErrCount; // other files include this
- **init.c:** // no longer needed
- **moduleX.c:** g_ErrCount++; // tally another error
- **moduleY.c:** XVar = g_ErrCount; // get current number of errors
- **moduleZ.c:** g_ErrCount = 0; // clear number of reported errors

Convert to File Static

- **ErrCount.c:** static int ErrCount = 0; // only visible in this file
- **ErrCount.h:** // static variables are invisible outside .c file
- **moduleX.c:** g_ErrCount++; // tally another error
- **moduleY.c:** XVar = g_ErrCount; // get current number of errors
- **moduleZ.c:** g_ErrCount = 0; // clear number of reported errors

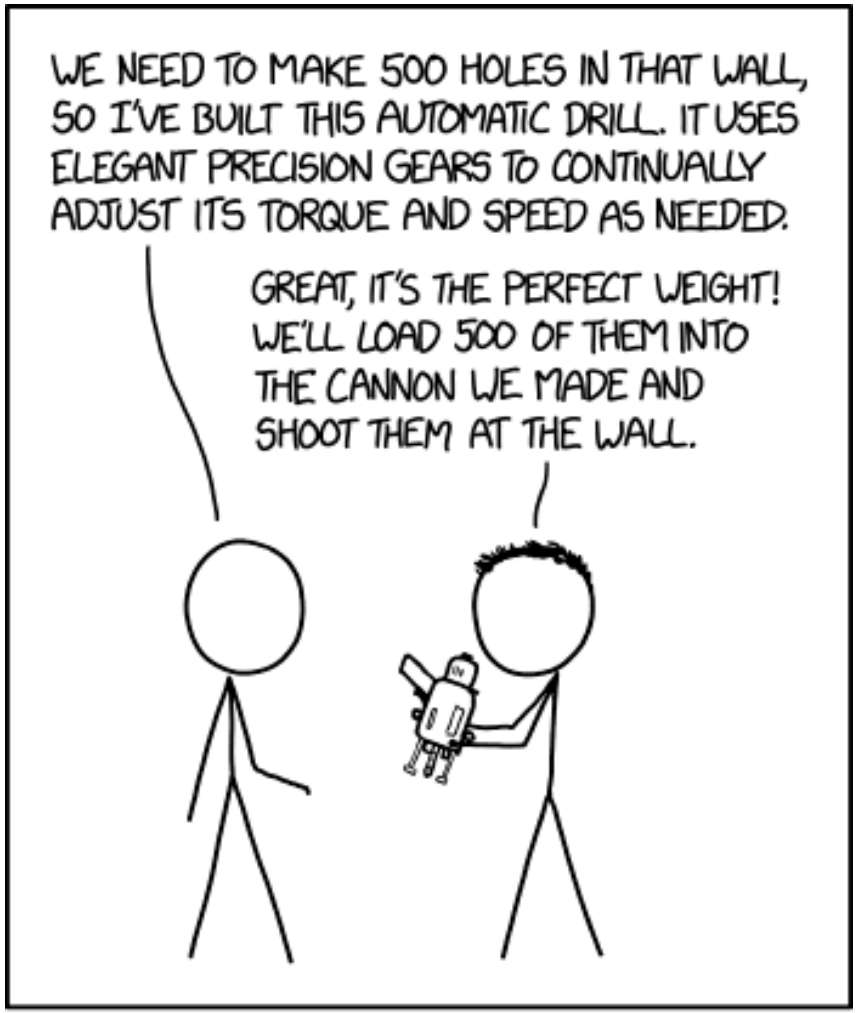
Add Accessor Function

- **ErrCount.c:** static int ErrCount = 0; // only visible in this file
  - inline void ErrCount_Incr() { ErrCount++; }
  - inline int ErrCount_Get() { return(ErrCount); }
  - inline void ErrCount_Reset() { ErrCount = 0; }
- **ErrCount.h:**
  - inline int ErrCount_Get(); // get current count value
  - inline void ErrCount_Reset(); // reset count
  - inline void ErrCount_Incr(); // increment the count

  // Note that there is NO access to ErrCount directly
- **moduleX.c:** ErrCount_Incr(); // tally another error
- **moduleY.c:** XVar = ErrCount_Get(); // get current number of errors
- **moduleZ.c:** ErrCount_Reset(); // clear number of reported errors

Advantages of this Approach

- Software authors can only perform intended functions specific to an error counter: increment, read, and reset. Setting to an arbitrary value isn't allowed. If you don't want the value changed other than via incrementing, you can just delete the reset function. This prevents some types of bugs from ever happening.
- If you need to change the data type or representation of the counter used that all happens inside ErrCount.c with no effect on the rest of the code. For example, if you find a bug with error counts overflowing, it is a lot easier to fix that in one place than every place that increments the counter!
- If you are debugging with a breakpoint debugger it is easier to know when the variable has been modified, because you can get rid of the "inline" keywords and put a breakpoint in the access functions. Otherwise, you need watchpoints, which aren't always available.
- If different tasks in a multitasking system need to access the variable, then it is a lot easier to get the concurrency management right inside a few access functions than to remember to get it right everywhere the variable is read or written (get it right once, use those functions over and over). Don't forget to make the variable volatile and disable interrupts when accessing it if concurrency is an issue.



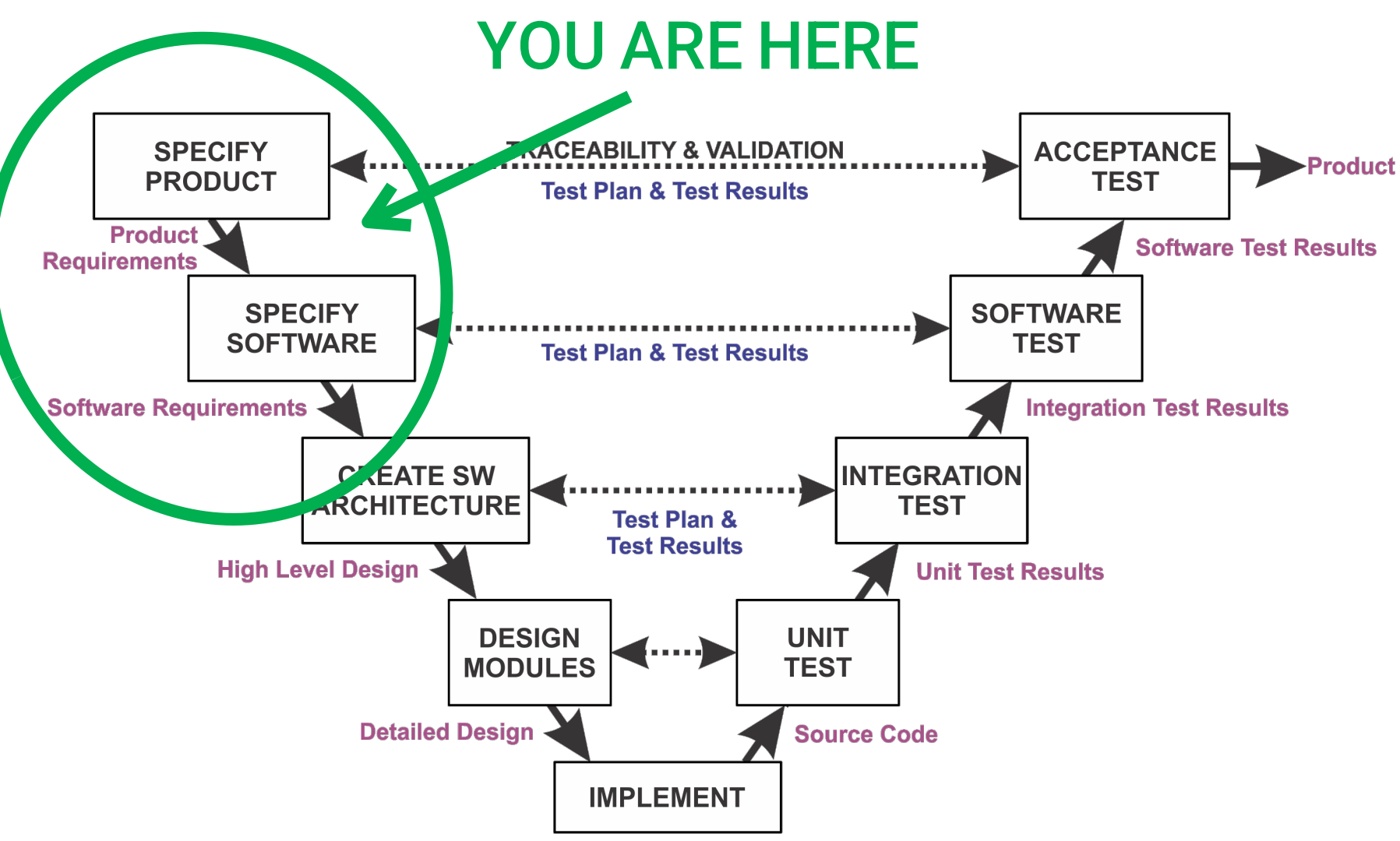
HOW SOFTWARE DEVELOPMENT WORKS

https://xkcd.com/2309/

Embedded Software Requirements

"In spite of appearances, people seldom know what they want until you give them what they ask for. "

– Donald Gause and Gerald Weinberg,
Are Your Lights On?


YOU ARE HERE
SPECIFY PRODUCT
TRACEABILITY & VALIDATION
Test Plan & Test Results
ACCEPTANCE TEST
Product
Product Requirements
Software Test Results
SPECIFY SOFTWARE
Test Plan & Test Results
SOFTWARE TEST
Software Requirements
Integration Test Results
CREATE SW ARCHITECTURE
Test Plan & Test Results
INTEGRATION TEST
High Level Design
Unit Test Results
DESIGN MODULES
UNIT TEST
Detailed Design
Source Code
IMPLEMENT

Requirements Overview

- Anti-Patterns:
  - Requirements aren't written down
  - Requirements incomplete, imprecise
  - "Be like last version, except…"

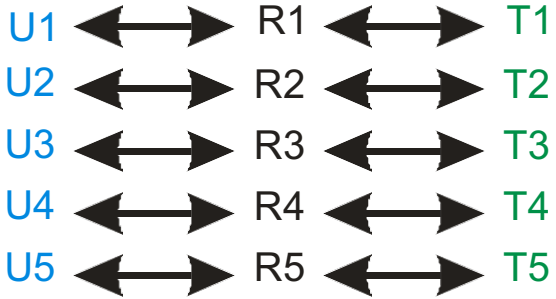
- Requirements
  - Requirements faults can defeat a design before it is even built
  - Describe what system does
    - Also what it's not supposed to do
  - Precise, testable language
    - Each requirement traces to system test

- 2005:
  $170M
  FBI Virtual Case File project terminated

- Requirements issues:
  - Requirements not defined when development contract signed
  - "We will know it when we see it"
  - Repeated requirements changes
  - Scope creep (new requirements added) of 80%

Characteristics of Good Requirements

- Precise and minimally constrained
  - Describes what system should do, not how it does it
  - Uses "shall" to require an action; "should" to state a goal
  - If possible has a numeric target instead of qualitative term
    - Has tolerance (e.g., 500 msec +/- 10%, "less than X")
- Traceable & testable
  - Each requirement has a unique label (e.g., "R-7.3")
  - Each requirement cleanly traces to an acceptance test
  - Requirement satisfaction has a feasible yes/no test
- Supported within context of system
  - Supported by rationale or commentary
  - Uses consistent terminology
  - Any conflicting requirements resolved or prioritized

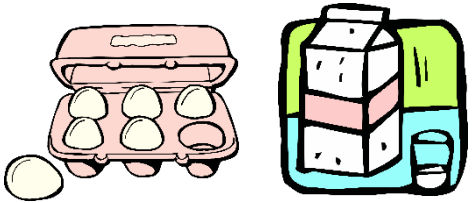


Problematic Requirements

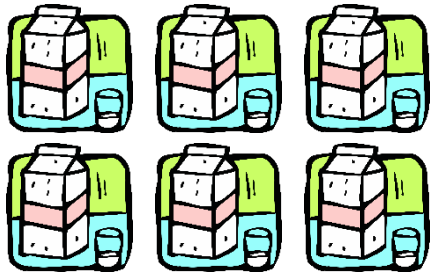
- Untraceable (no label)
  - System shall shut down when E-STOP is activated.
- Untestable
  - R-1.1: System shall never crash
- Imprecise
  - R-1.7: The system provides quick feedback to the user.
- No measurement tolerance
  - R-2.3: LED shall flash with a period of 500 msec
- Overly complex
  - R-7.3: Pressing the red button shall activate Widget X, while pressing the blue button should cause LED Z to blink instead of LED Y illuminating steadily, which would be accomplished via the yellow button.
- Describes implementation
  - R-8.3: Pressing button W shall cause two 16-bit integer values to be added, then …

Requirements Ambiguity

- A requirements engineer gets a text message:
  "On the way home, please pick up one carton of milk.
  And if they have eggs, get six."



- The requirements engineer comes home with:
  6 cartons of milk and no eggs.

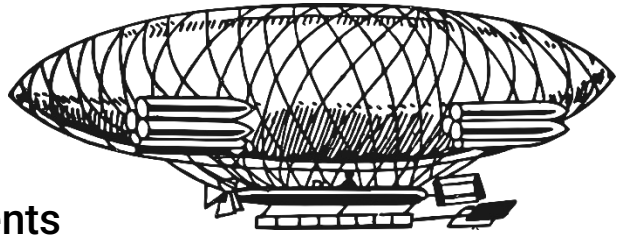


- Spouse: "Why did you buy six cartons of milk?!"
- Requirements Engineer: "They had eggs."

Adapted from: www.ganssle.com/jokes.htm.

Extra-Functional Requirements

- Emergent properties (things hard to attribute to one component)
  - Performance, real-time deadlines
  - Security, Safety, Dependability in general
  - Size, Weight and Power consumption ("SWaP")
    - Often handled with an allocation budget across components
  - Forbidden behaviors ("shall not do X")
    - Often in context of safety requirements
    - "Safety function" is a way to ensure a negative behavior, but some behaviors are emergent



- Design constraints
  - Must meet a particular set of standards
  - Must use a particular technology
  - System cost, project deadline, project staffing

https://goo.gl/hT3nDU

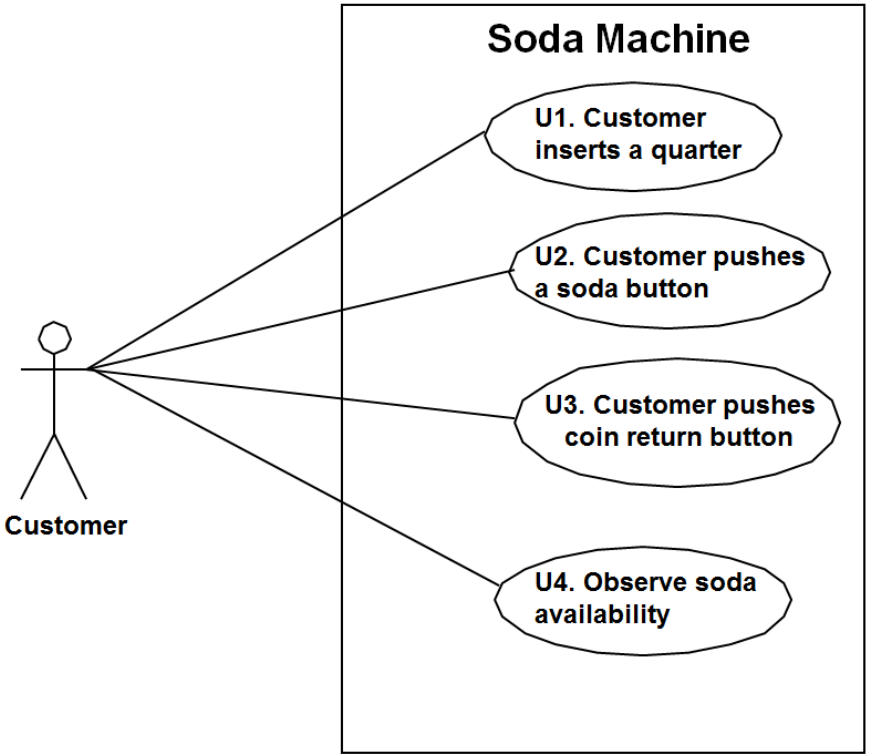
Product vs. Engineering Requirements

- Product level requirements:
  what the product does
  - Example:
    "PR6. The clock shall support a user-settable audible alarm."
  - Gives a feature list of what the product actually does
  - Can be the interface between marketing and engineering

- Detailed functional/engineering requirements:
  how the product actually works
  - Example: "R5. Time set buttons shall change the alarm set time."
  - Embedded systems often have detailed requirements tied to operational modes
    - "R5. In Alarm Set Mode the time set buttons shall change the alarm set time."
    - "R6. Pressing the "+" time set button shall increase time value by one minute per button press according to the current set mode."

Requirements Approaches

- Text document with list of requirements
  - Works best if domain experts already know reqts.
  - Over time, this can converge to OK reqts.
- UML Use Cases
  - Different activities performed by actors
  - Requirements are scenarios attached to each use case
- Agile User Stories
  - Each story describes a system interaction
- Functional decomposition
  - Start with primary system functions
  - Make more and more detailed lists of sub-functions (creates a “functional architecture”)
- Prototyping to elicit requirements
  - Customers know it when they see it
  - Sometimes a paper mock-up is enough



UML Use Case Diagram

Requirements Templates (EARS)

- Easy Approach to Requirements Syntax (Mavin et al.) e.g.: https://bit.ly/2CQSF37
- [While/Where <precondition>] [when/if <trigger> then]
  <system> shall <response>
  - Ubiquitous: The touch screen shall have a response time of less than 250 msec.
  - State-driven: WHILE an external speaker is connected, the internal speaker shall mute.
  - Event-driven: WHEN a card is inserted, the card reader shall verify credentials.
  - Optional feature: WHERE a convertible roof is installed, a park/roof motion interlock function shall be provided.
  - Unwanted: IF an invalid value is entered THEN an error message shall be displayed.
  - Complex: combinations of the above
- Requirements issues to avoid:
  - Ambiguous, vague, complex, omitted, duplicated, wordy, implementation, untestable

Example Software Requirements

National Aeronautics and Space Administration

GRC-CONN-REQ-0084
EFFECTIVE DATE: 02/23/2010

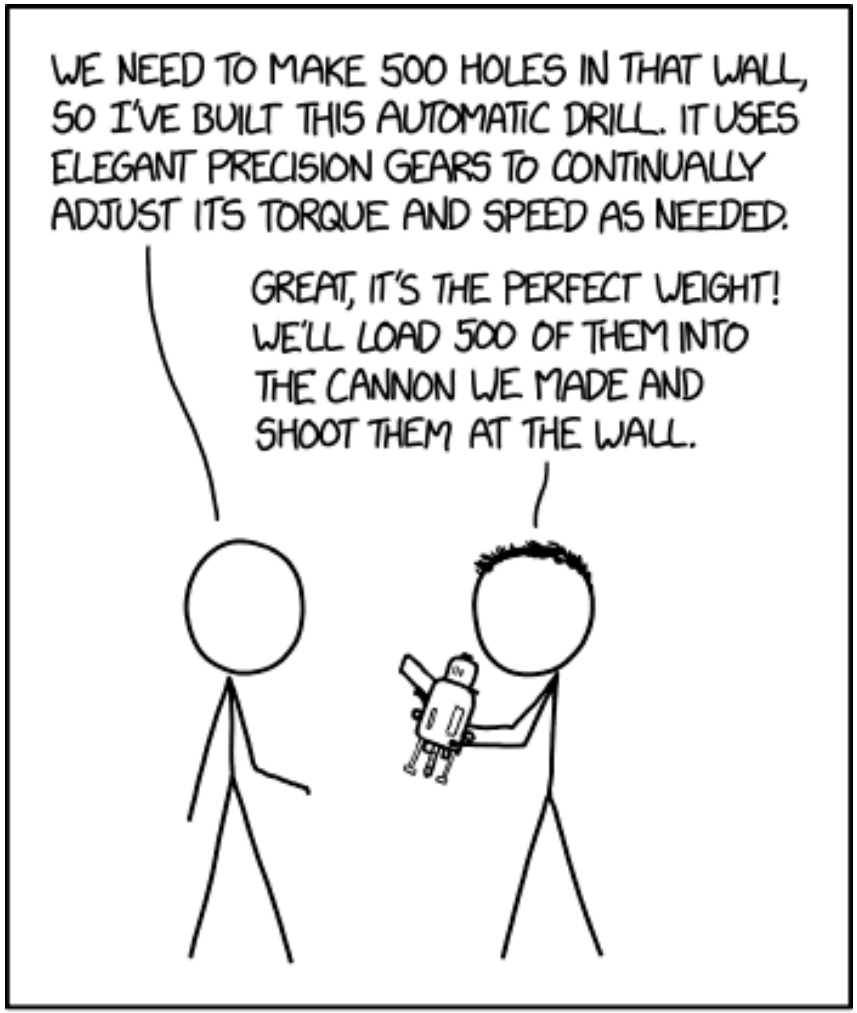
Parent Req	ReqID	Requirement Text and Rationale	Prior-ity	Allocated To
FSRD-3714	SRS-3.2.6.12	The Software **shall** send data at a user data rate from zero up to and including 100 Mbps. *Rationale: The maximum data rate the Payload Avionics Software must send is 100 Mbps. Lower rates must also be handled.*	P2	PAS
FSRD-3133	SRS-3.2.6.13	The software **shall** send and receive data on two SpaceWire channels simultaneously at up to the maximum SDR interface data rate (full duplex) that can be sustained by both SDRs. *Rationale: When communicating with multiple radios, the Software will need to sustain an achievable data rate. In this requirement, it is defined as the minimum data rate of the two (or three, if possible) SDRs involved in the experiment. For instance, this data could be provided in the routing table. If two other radios are involved, then the data rate may change, based on the capability of those two radios (i.e. a new minimum interface data rate). This value should not be hard coded, but should have the capability for change, once on-orbit.*	P2	PAS

https://goo.gl/qct5tL

Best Practices for Requirements

- Six C-terms for Good Requirements
  - Clear, Concise, Correct, Coherent, Complete and Confirmable
- Also:
  - Deal with extra-functional issues
  - Relate requirements to design flow
    - Associate with user stories or use cases
    - Trace to corresponding test
- Requirements pitfalls
  - Avoid unnecessary details and implementation
  - If it's missing from requirements, it won't get done
  - If it's not testable, you won't know if it got done

https://goo.gl/6H3dxi



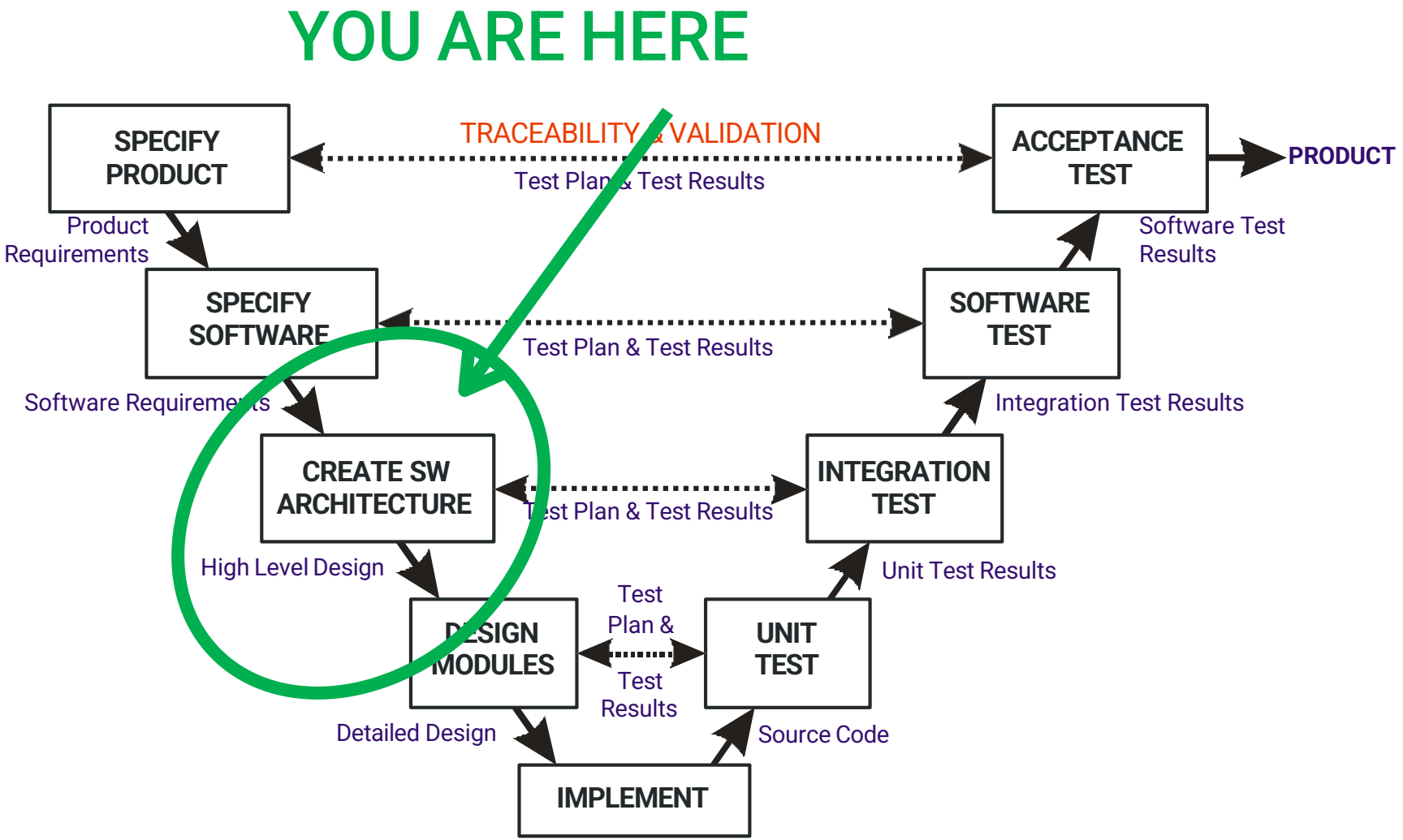
HOW SOFTWARE DEVELOPMENT WORKS

https://xkcd.com/2021/

Software Architecture & High Level Design

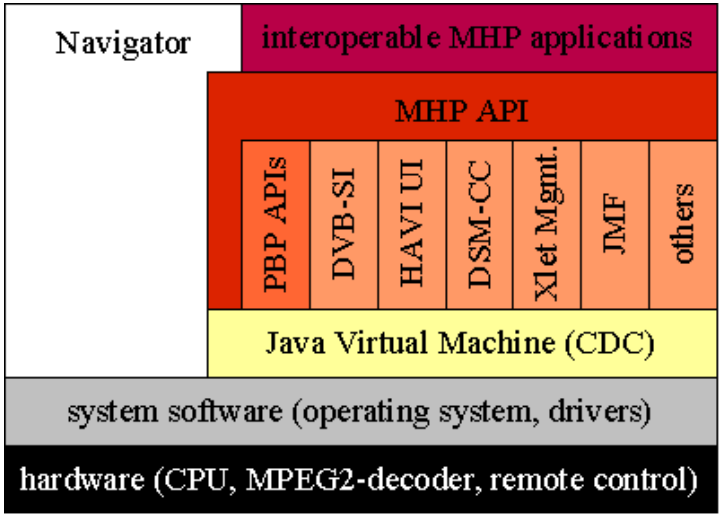
All the really important mistakes are made the first day.

– Eberhardt Rechtin,
System Architecting


YOU ARE HERE
SPECIFY PRODUCT
TRACEABILITY & VALIDATION
Test Plan & Test Results
ACCEPTANCE TEST
PRODUCT
Product Requirements
Software Test Results
SPECIFY SOFTWARE
Test Plan & Test Results
SOFTWARE TEST
Software Requirements
Integration Test Results
CREATE SW ARCHITECTURE
Test Plan & Test Results
INTEGRATION TEST
High Level Design
Unit Test Results
Test Plan &
DESIGN MODULES
Test Results
UNIT TEST
Detailed Design
Source Code
IMPLEMENT

Architecture & High Level Design (HLD)

- Anti-Patterns:
  - Skipping from requirements to code
  - No picture that shows how all the components fit together
  - “Wedding cake” layer diagram that omits interface information

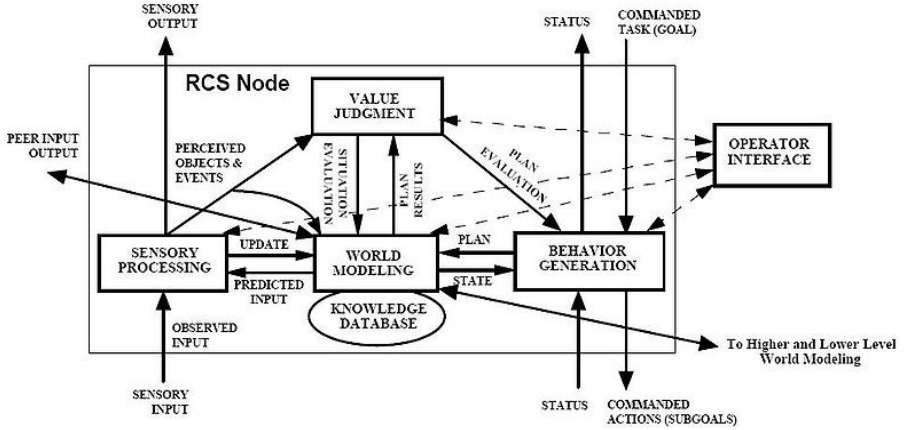


https://goo.gl/J8MAuK

- Elements of High Level Design
  - Architecture: boxes, arrows, interfaces
    - Arrows/interfaces show communication paths between components
    - Recursive: one designer’s system is another designer’s component
  - High Level Design (HLD) = architecture (nouns) + requirements (verbs)
    - Sequence Diagrams (SDs) show interactions

Architecture: Boxes and Arrows

- Software architecture
  shows the big picture
  - Boxes: software modules/objects
  - Arrows: interfaces
  - Box and arrow semantics well-defined
    - Meaning of box/arrow depends on goal
  - Components all on a single page
    - Nesting of diagrams is OK

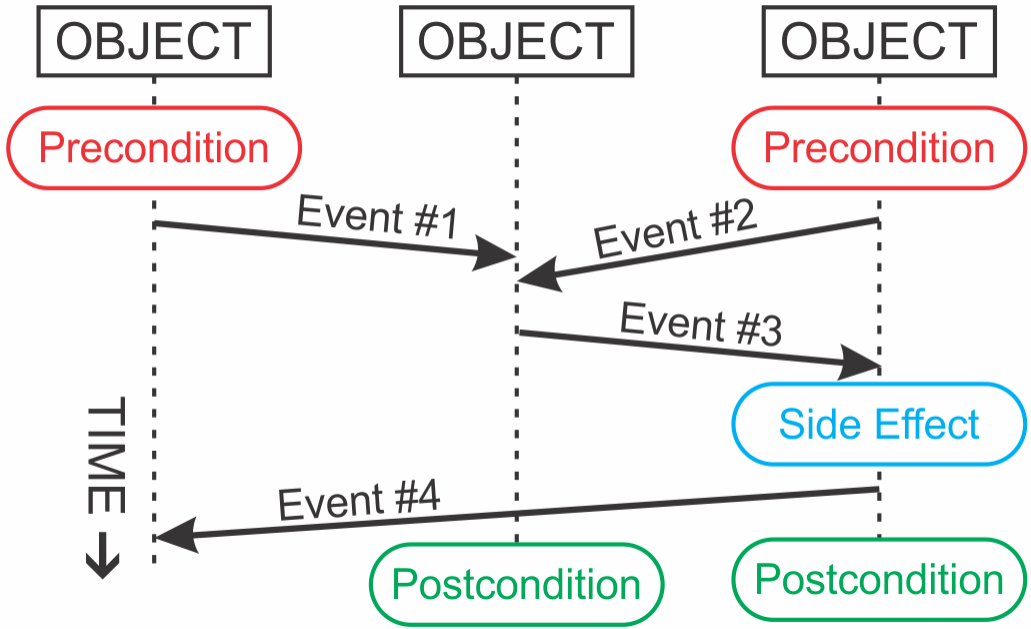


https://goo.gl/WnciF3

- Many different architecture diagrams are possible, such as:
  - Software architecture (components and data flow types)
  - Hardware architecture with software allocation
  - Controls architecture showing hierarchical control
  - Call graph showing run-time hierarchy

Sequence Diagram as HLD Notation

- SD construction:
  - Each object has a time column extending downward
  - Arcs are interactions between objects
- Each SD shows a scenario
  - Top ovals are preconditions
  - Middle ovals are side effects
  - Bottom ovals are postconditions

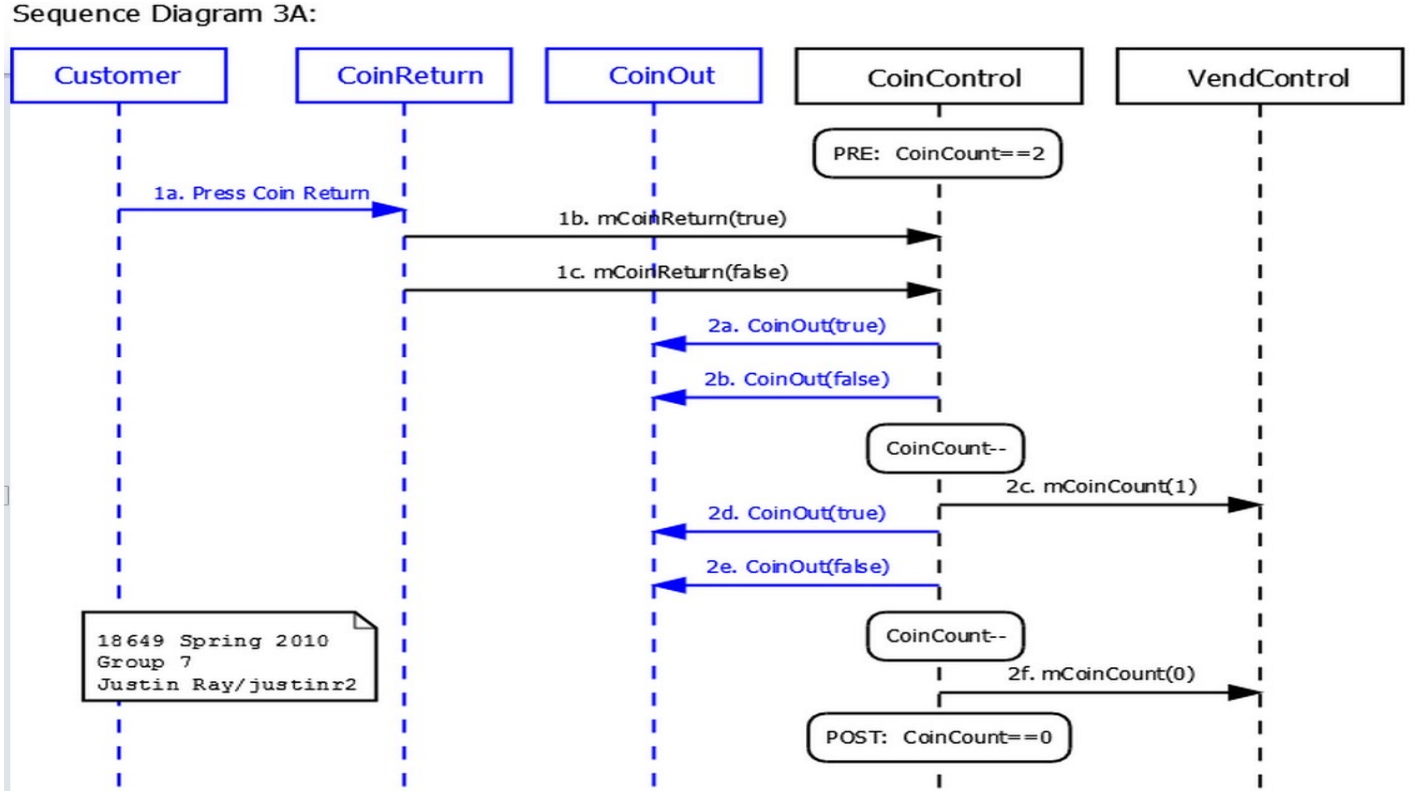


- SD is a partial behavioral description for objects
  - Generally, each object participates in multiple SDs; each SD only has some objects
  - The set of all SDs forms the HLD for all objects in the system

Example Sequence Diagram

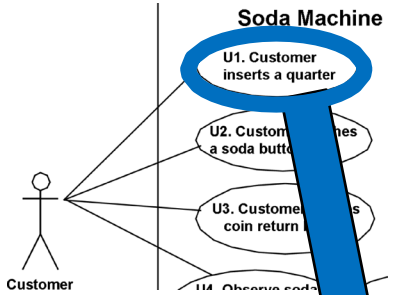
Legend: Blue = physical objects / Black = microcontrollers with software

PRE = precondition / POST = postcondition / other ovals are side effects



Use Cases to Sequence Diagrams

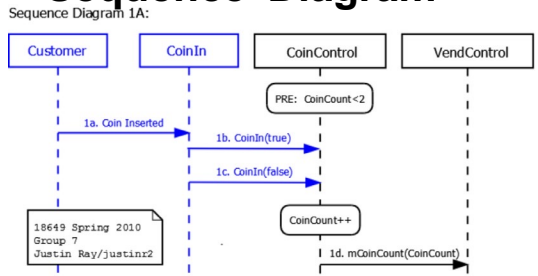
- Use Case diagram – types of interactions
  - System has multiple use cases
  - Example: Use Case #1: Insert a coin
- Scenario – a specific variant of a use case
  - Each use case has one or more scenarios
    - Scenario 1.1: insert coin to add money
    - Scenario 1.2: insert excess coin (too many inserted)
    - Scenario 1.3: … some other situation…
  - Interactions between objects are different for each scenario
- Sequence Diagram – a specific scenario design
  - For our purposes each scenario has one sequence diagram
    - Sequence diagrams 1.1, 1.2, 1.3 show specific interactions
- Statechart – design that incorporates all scenarios
  - One StateChart per object, addressing all scenarios



Pre-Conditions:
The soda machine is not vending.
No button is pressed.
The system has received fewer coins than the cost of a soda since the last vend cycle.
Scenario:
The customer inserts a coin.
Post-Conditions:
The soda machine has one additional coin for this vend cycle.

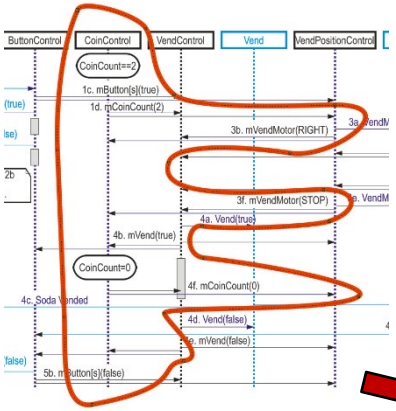
Scenario

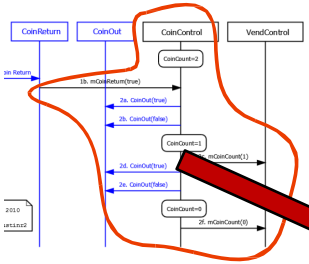
Sequence Diagram



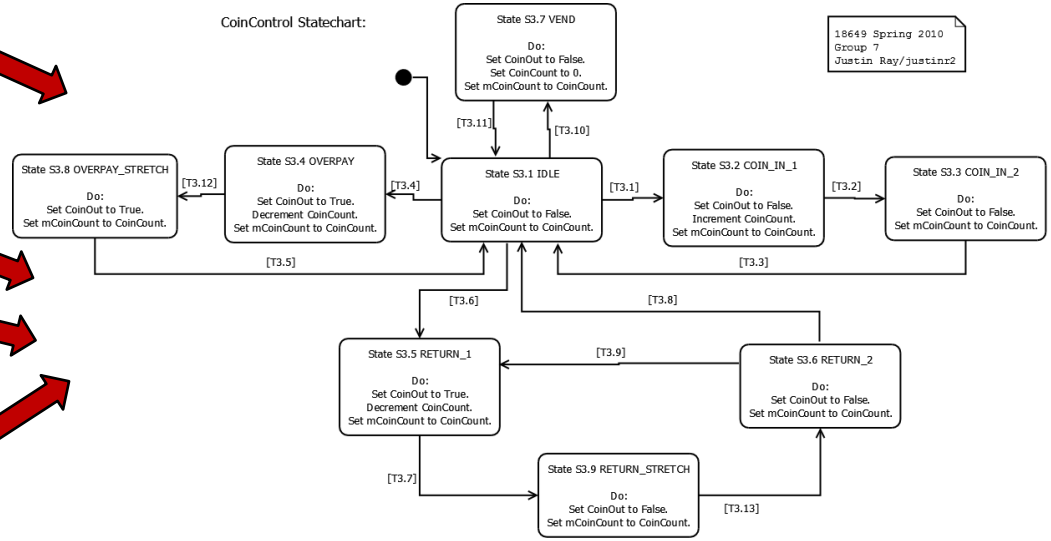
Combining SDs To Make Statecharts

- For each object in each SD: identify input & output arcs
  - Detailed Design: design statechart that accounts for all SD behaviors

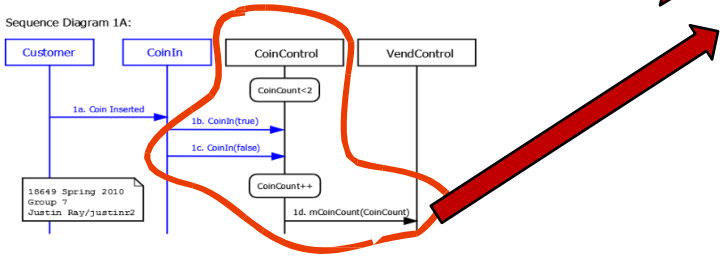




Statechart Must Exhibit All Those Behaviors

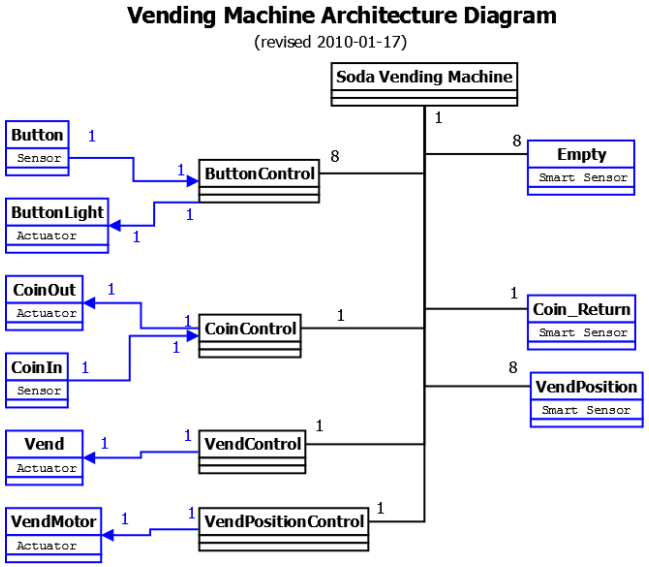


SD set specifies behaviors



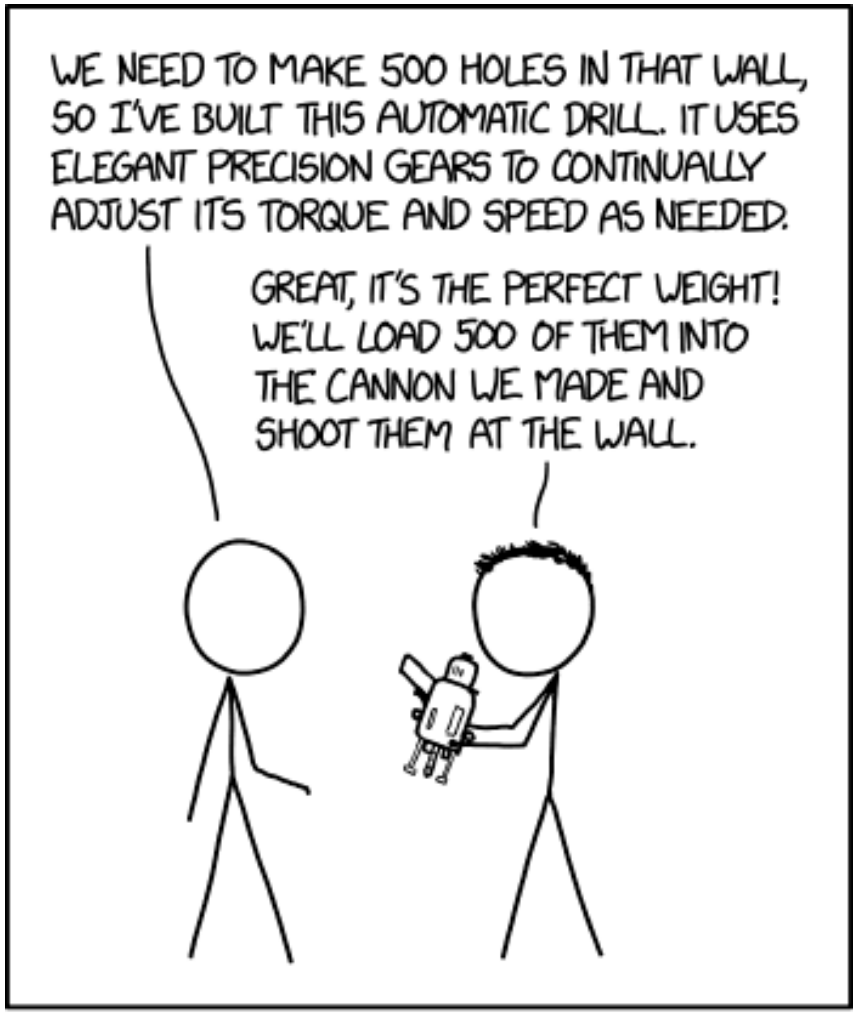
High Level Design Best Practices

- HLD should include:
  - One or more architecture diagrams
    - Defines all components & interfaces
    - HW arch., SW arch., Network arch., …
  - Sequence Diagrams
    - Both nominal and off-nominal interactions
  - HLD must co-evolve with requirements
    - Need both nouns + verbs to define a system!



- High Level Design pitfalls:
  - Diagrams that leave out interactions
  - Boxes and arrows don't have well defined meanings
  - HLD that bleeds into detailed design information
    - Should have separate Detailed Design per component

https://users.ece.cmu.edu/~koopman/ece649/project/sodamachine/index.html



HOW SOFTWARE DEVELOPMENT WORKS

https://xkcd.com/974/

Disclaimer

This lecture contains materials from:

- Philip Koopman - CMU