Syllabus
Assets/**Q-Kinds**

Read ~ 35 pages: Chs 1, 2-2.4 from 9e – or Chs 1,2,3 from 8e.

SWE Introduction

[nox]
Pgm Sizes – [non-std – old-school typewritten "page" = 50-55 lines/page & 12 5-letter words/line]
XXS = 25 LOC, half page (avg typical algorithm size)
XS = 100 LOC, 2 pgs (AKA Tiny)
SM = 500 LOC, 10 pgs
MD = 2,500 LOC, 50 pgs
LG = 10,000 LOC, 200 pgs
XL = 50,000 LOC, 1,000 pgs
XXL = 250,000 LOC, 5,000 pgs
XXXL = 1,000,000+ LOC, 20,000 pgs

Pgmg Lang power == fewer LOC to get the job done (in an easily understandable way)

Why we need SWE?

[Jones][Chaos]

o- 20% **to 80%** of all non-small (> 20 pgs) projects **Fail** – **% depends on who you talk to**
o-- or experience a **massive overrun** (> 33%) in cost/effort or timeline, usually both
→ paid for by grudging customer (gov't agency), who probably won't buy from you again
o-- **We will go over** these reports & some exciting massive failures in lecture

o- **2 Kinds of Failure**: 1) Never delivered, and 2) Delivered but deemed **unsuitable** to use
o-- **Unsuitable** – 1) too slow, 2) too complicated to operate,
3) too many errors/inaccuracies, and/or 4) breaks down too often (low MTBF)

Why do projects Fail? **4 major reasons**:

o- **Complexity** (scales exponentially ceteris paribus)

o- **Mgrs** (mostly lack of knowledge)
o-- Fail to manage devr (developer) **morale** → much lower productivity == much more effort
o-- Fail to arrange adequate/effective **comm w users**
o-- Fail to see **looming risk** events in time
o-- Fail to **gel group** into a team, or maintain the team

o- **Comm poor** with "users" (built ill, unsuitable product)
o-- (Extra hard) Not every project has an individually identified set of users

o- **Bad Prediction** of "The Plan" (Features/Effort/Timeline – AKA the "Project Triangle")

SWE Status (eg, today, how does SWE "look")

The Good
o- Pgmr Need is still growing faster than all other occupations

The Bad
o- "SWE" coined 1968 (at NATO conf), after a decade-plus of failed big-gov projects
o- 20% to 80% of all **M-L-XL+** SWE projects fail today, [Standish/Chaos][Jones]
o- SW Defects (= shipped bugs) cost US alone ~$60B [Tassey 2002] – likely an under-estimate

The Ugly
(*)** No one knows how to build SW (>= medium size, 50pgs) reliably
o- Parnas/Lawford: 30+ years SWE research hasn't helped [Parnas 2003]
(*)* Key: "Quality S/W" == Bug-Free – never achieved, even for the "good" products
o-- 10s of $$M spent on SWE **research** to date (**pbb >> $1B**)
o-- NB, Dave Parnas buddy w Fred Brooks (IBM & taught at UNC Chapel Hill, good CS dept)

o- **"No Silver Bullet", Brooks**, 1985 -- still true today – **we will read** this paper
o- SWE taught for BSCS for >40 years

o- Numerous **SWE "Quality" Groups** now exist, some >30 years old
o-- SWE today is **barely** better understood than 4+ decades ago
– the "barely" award goes to the **Agile Manifesto** crowd – **we will read** & memorize it

o- Sample of major SWE Quality Groups
ABET -- for Univ CS Dept curriculum accreditation – undergraduate degrees
ACM -- for model CS curriculum – undergraduate degrees
SEI -- **CMMI** pseudo-metrics + a large pile of docs on how to do SWE
Computer.org – Certs (Certifications)
SWEBOK -- a mishmash, some useful (SWE Body of Kno.wledge) [Bourque 2014)
SWECOM -- ditto (SWE Competency Model)
ISO 9xxx group (Euro) – SWE standards docs

Assets/**Mini-SWE** Prelims & Rules

Pressman & Maxim: 8th ed vs 9th ed – line by line correspondence of section headings

Ch 1. The Nature of SW 1 1 . 1 The Nature of Software 3 1. 1. 1 Defining Software 4 1. 1.2 Software Application Domains 6 1. 1.3 Legacy Software 7 2. 1 Defining the Discipline 15 2. 2 The Software Process 16 2.2.1 The Process Framework 17 2. 2.2 Umbrella Activities 18 2. 2.3 Process Adaptation 18 2.3 Software Engineering Practice 19 2.3.1 The Essence of Practice 19 2.3.2 General Principles 2 1 2.5 How It All Starts 26	Ch 1 Software And Software Engineering 1 1.1 The Nature of Software 4 1.1.1 Defining Software 5 1.1.2 Software Application Domains 7 1.1.3 Legacy Software 8 1.2 Defining the Discipline 8 1.3 The Software Process 9 1.3.1 The Process Framework 10 1.3.2 Umbrella Activities 11 1.3.3 Process Adaptation 11 1.4 Software Engineering Practice 12 1.4.1 The Essence of Practice 12 1.4.2 General Principles 14 1.5 How It All Starts 15
3. SW Process Structure + 4. Process Models 3. 1 A Generic Process Model 31 3.2 Defining a Framework Activity 32 3.3 Identifying a Task Set 34 3.5 Process Assessment and Improvement 37 4.l Prescriptive Process Models 41 4.1.1 The Waterfall Model 41 4.1.2 Incremental Process Models 43 4.1.3 Evolutionary Process Models 45 4.3 The Unified Process 55 4.6 Product and Process 62	Ch 2 Process Models 20 2.1 A Generic Process Model 21 2.2 Defining a Framework Activity 23 2.3 Identifying a Task Set 23 2.4 Process Assessment and Improvement 24 2.5 Prescriptive Process Models 25 2.5.1 The Waterfall Model 25 2.5.2 Prototyping Process Model 26 2.5.3 Evolutionary Process Model 29 2.5.4 Unified Process Model 31 2.6 Product and Process 33

More – **SWE Status (eg, today, how does SWE "look")**

Fixing SWE – How to Fix Complexity?

Agents + "Separation of Concerns"

o- They help other agents (or the user directly)
o- Do one thing, conceptually (**SRP from SOLID/D, High Cohesion**)
o- Own their data (**Encapsulation, or Data Hiding**)
  o-- Do things related to their data
o- Don't know much about other agents except helper **API**s they need (**Low Coupling**)
Don't Optimize (till proven it is needed) – (**Predictions are Hard**)
Don't "Build for a Future Product"– build only for "today's project" – (Predictions are Hard)
Don't "Build for Reuse" until at least building the 3-rd similar project – (Predictions are Hard)
  o-- Don't generalize from one data point
Don't "Build tall class hierarchies" (**OCP, LSP from SOLID/D**)
(*) Build **a little** (a "feature set slice") and deliver it, and get user feedback

How to Fix Mgrs?

Recognize Which Kind – **4 Categories**: the Good, the Bad, the Ugly
o- **Good**, top 10%, **supt**/protect their people
  o-- They create "teams" from "groups" ("gung ho" – work together – given to US Marines, China)
  o-- Clue: They ensure everyone knows who is doing what – gently
o- **Ugly**, bot 10%, have a reputation to build, at expense of their people
o- **Bad, Clueless**, top 40%, can't tell when they "annoy" their people
  o-- Can still deliver products
  o-- They sometimes create "teams" from "groups"
o- **Bad, Callous**, bot 40%, don't care when they "annoy" their people
  o-- "We pay you, don't we? Well, we need this done by Friday. Get it done."
  o-- They convert (almost) any "teams" to "groups"
No mgr is always in the same Category all the time – but mostly so
Peter Principle: people rise to their **level of incompetence** (Lawrence J. Peter)
Mgrs might grow in the job, for better or for worse
Work for the top half, if you can
Work for (real) Agile, if you can
Warning: there is no such thing as "**a team-building exercise**" in reality

How to Fix Comm (with the Users)?

o- Try to Talk in User's "**Problem Domain Language**" – the Problem they need help for
  o-- Helps uncover mistakes in word/phrase meaning – avoids misinterpretations
o- **Don't talk** to users about **CS** concepts – they need results – they won't understand
o- **Don't make users** fill out anything – too much work for them
  o-- In any conversation, be their secretary & write down what they say – make it easy for them
o- Ask user's **how they do it now**, or how they would do it
  o-- Users think in **actions** and **steps** and **sequencing** and **conceptual data** and **results**
  o-- What would they tell a human personal assistant (PA) to do? That PA is your pgm
o- Use **CRC Hand-Simulation**, and listen to users view on your CRC model sim
o- Give users tiny feature slices **every few weeks** to play with and **give you feedback (Agile)**
  o-- Each time, let **users pick the features** that are most urgent – (users assign priorities)
  o-- You bundle the top few of those features into a "slice" you think you can deliver next time

How to Fix Bad Predictions?

- o- Try to **avoid Predictions (Agile avoids BIG predictions)**
  - o-- Most mgmt can't allow this
  - o-- Remember: all businesses make predictions about future needs and cash flow
    - (*) but S/W projects are "special" – insanely exponential (in complexity)
- o- Give **Error Bars** to all Predictions (say it "should work out" 4/5 or 80% of the time)
  - o-- Finer predictions/estimates are **probably not accurate**
  - o-- Say the other 20% of the time, is likely to take **twice as long** (but who really knows?)
- o- Gain **vast experience** in making small-scale predictions – (use the **Half-Day Rule**)
  - o-- Get a feel for how long Small (10 pg) & Tiny (2 pg) & XXS (half pg) tasks usually take
  - o-- XXS is your average simple algorithm – eg, quicksort, priority queue, minimum spanning tree

o1.1 The Nature of Software p3

S/W makes a universal computation machine into a specialized Personal Assistant (PA)

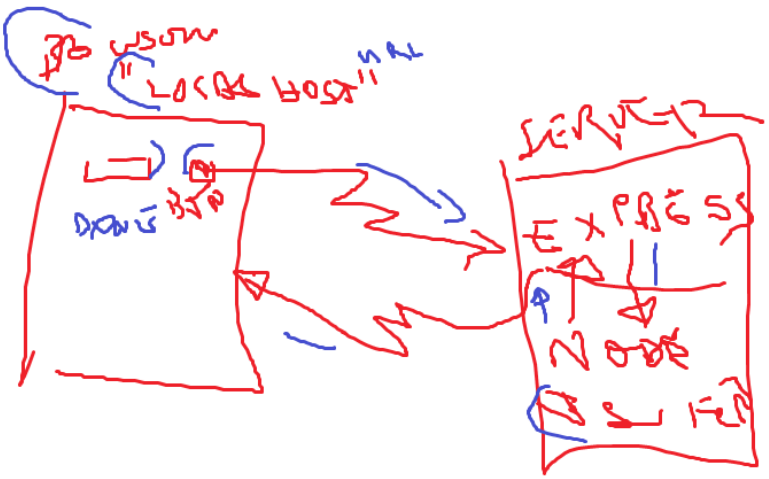
S/W Nature Qs:

- **Why does it take so long to build?**
  - → Unlike engr'g things like buildings or ckts, S/W complexity typically has 1M to 1T bit dynamic state data
  - → Almost always, dynamic info is used to control execution pathways (AKA behavior changes)
  - → Easily a factor of 1K..1B times more complex behavior than other kind of engr'g
  - **(*)** Key to fix:**
    - o- Reduce # of pathways (CF McCabe's Cyclomatic Complexity metric)
    - o- Reduce complexity of data flowing between boxes (AKA Coupling) along those pathways
    - o- Make some/most/all components "functional" (AKA don't depend on state, only on input args)
    - o- Reuse Reliable parts (eg, 3-rd party libraries, frameworks, or maybe your own SPL cores/frameworks)
      - o-- You want to Make new-dev code size be <= Med size (like 10 pages or less)
    - o- Include 1+ very strong pgmrs on your team (very hard to do)
- **Why are development costs so high?**
  - → Hard to predict/estim effort due to complexity, hence very risky
  - → Labor is expensive (more so for specialized knowledge: low-use languages, tools, algorithms)
  - → Most of effort is spent in find/fix run-time bugs: **emergence == unforeseen & significant**
  - → Big % of projects fail – Bigger the size, more likely to fail – their costs is spread over successful projects
- **Why are defects in the completed pgm?** (**Defect** == shipped bug, (usually) found by users)
  - → Complexity: (# of pathways) coupled with (# state changes) too big to test, ever
- **Why do we maintain existing programs (AKA Legacy) for so long?**
  - → You would too if new-dev was so risky: of fail & of shipping-bad-stuff
- **Why is it hard to measure project progress for new-dev & maint/upgrade?**
  - → Mostly, **poor metrics** (things meas'd) vs predictive accuracy – (AKA giant error bars/uncertainty)

o1.1.2 S/W App Domains (AKA Pbm Areas) p7

- **System** – tools & mech foundations – OS, Network, browser, GUI, cloud, ML, IDEs, languages,...
- **App** – highly focused biz/user-areas
- **Engr/Sci** – answers to engr'g or sci questions, w accuracy & performance
- **Embedded** – no kbd, no screen – S/W added to specialty CPU box, cust buys the box to do a job
  EX: cars (have maybe 10's of CPU chips), older cell phones, refridgerators (for IoT)
- **S/W Product-Line** == **SPL** – Devrs stamp out many similar sol'ns for many competitors in a Biz area
  EX: supermarkets, factory small-box assembly (eg hand-held electronics), ...
  o-- Make new-dev code size be <= Med size
  → Key: cheap to make 1 reusable core/framework & then new-dev just adds Biz-specific feature variations
- **Web/Mobile App** – just another new App sub-area, maybe more GUI focus
- **AI** – Plan B, when no algo gives a fast, exact answer
- o**1.1.3 Legacy** (the old "Cash Cow" product still gives milk (the money))
  → Old stuff needs kept running: fixes, small upgrades
  o- Very expensive to rewrite – and also high risk of failure (as we've seen)
  o- Susceptible to Lehman's "Mummy" Laws – factors that degrade legacy S/W over time

Lab:
assets/standup-status-sample
pic, poor – Project Infrastructure – HTML + JS + Node + Express (half of the M**E**R**N** stack)



Pressman & Maxim: 8th ed vs 9th ed – line by line correspondence of section headings

5. Agile Dev 66	Ch 3 Agility And Process 37
5.1 What Is Agility2 68	3.1 What Is Agility? 38
5.2 Agility and the Cost of Change 68	3.2 Agility and the Cost of Change 39
5.3 What Is an Agile Process? 69	3.3 What Is an Agile Process? 40
5.3.1 Agility Principles 70	3.3.1 Agility Principles 40
5.3.2 The Politics of Agile Development 71	3.3.2 The Politics of Agile Development 41
5.5.1 Scrum 78	3.4 Scrum 42
-	3.4.1 Scrum Teams and Artifacts 43
-	3.4.2 Sprint Planning Meeting 44
-	3.4.3 Daily Scrum Meeting 44
-	3.4.4 Sprint Review Meeting 45
-	3.4.5 Sprint Retrospective 45
5.5 Other Agile Process Models 77	3.5 Other Agile Frameworks 46
5.4 Extreme Pgmg 72 (AKA “XP”)	3.5.1 The XP Framework 46
-	3.5.2 Kanban 48
-	3.5.3 DevOps 50

Sidebar: SWEBOK (started 1998, output v1 2004 – **ISO**/IEC TR 19759:2005)
What SWEBOK covers: (**SWE Body of Knowledge**) [Bourque 2014, Guide to v3, the latest)
o-- Covers a lot of stuff, some pretty well ** No Ship/***Deploy*** at top-level
SWEBOK -- **15 Practical KAs** (Knowledge Areas) – (each prefixed with "SW" or "SWE")
o- ***Reqts*** (AKA **Comm**)
o- ***Design*** (AKA Arch/**Model**/Analz) [Analz = Analyze User's Pbm] + some Detailed Design, maybe
o-- *"during software design, software engineers produce various **models** that form a kind of blueprint of the solution to be implemented"*
o- **Construction** (AKA ***Build***) = Detailed Design, Code, Unit Test
o- **Test** = I&T, V&V o-- **I&T = Integration & Test** – includes bolting "units" together
o-- **V&V= Verification** (works per spec) **& Validation** (users like it)
o- **Maint = Maintenance**, fixups & upgrades after first deployment/delivery
o- **SCM**/CMS/VCS – Version Ctrl/ Change Ctrl/ Doc Archiving
o- **Mgmt** = Feasibility, ***Plan***, Assign+Track+Ctrl, Tools, Metrics, Reports, Post-mortem – all per MO
(MO == Modus Operandi (Latin) == Mode/Method of Operation == Methodology)
o- **Process** (& Meta-Process) – Dev MO's, SDLC, & How to Improve (eg SEI's CMMI) using an MO
o- **Models & Mds** = (AKA Arch/**Model**/Analz) – incl tools for arch & autogen arch code (UML-ish)
o- **Quality** = V&V + dev **QA** + **SDLC** dev/maint culture (a bit of social + ethics)
o-- *"**Verification** is an attempt to ensure that the product is built correctly, in the sense that the output products of an activity **meet the specifications** imposed on them in previous activities."* – IE, **works per Spec**
o-- *"Validation is an attempt to ensure that the right product is built—that is, the product **fulfills its specific intended purpose**."* – IE, **users like it** – (users are always assumed fair judges)
o-- SWEBOK v3 also perpetuates confusing V1="*the product is built right*" & V2="*the right product is built*" – A very cute phrasing, but useless without the proper explanation of what it means.
o- **Pro Practice** = **#1 social & ethical** devrs behavior (what an individual should do)
#2 to "encourage" SW devrs to get **licenses/certifications** to practice SW development.
– there are a bunch of SW-related certs with BOKs:
CSQE for QA, CISSP for Security, PMBOK for Project Mgmt, DMBOK for Data Mgmt,
CSDP for Certified SW Development Professional, and others
(*) Doesn't emphasize Team Building issues – a big drawback
o- **Econ** = Staying in business – finance, acctg, cash flow, **time-value-money**, SDLC, SPL, Risk, EVM, ROI, Break-Even, Outsourcing (and more details)
o- **CS** Foundations = BSCS curriculum
o- **Math** Foundations = Sets, Fcns, Logics, Proofs, Graphs, Pbb, FSMs, Gmrs, Precision vs Accuracy, Number systems NZFRC, Primes/GCD, Groups/Rings/Fields
o- **Engrg** Foundations = Experiments, Stats, Meas, **Model+Sim+Proto**

Example SWEBOK weirdness:
*"For example, software **requirements validation** is a process used to determine whether the requirements will provide **an adequate basis** for software development; it is a sub-process of the software requirements process."* – we want **reqts to be testable** in the end product.
o- Reqts Validation (SWEBOK) means **Can we build** per Reqts? Not, users like it
o- Reqts Validation per user is unaddressed – Will users like it? We don't care!
SWEBOK naively assumes Reqts correctly predict user's future deployment-time likes.

Example SWEBOK weirdness (ditto):
*"Although **some** detailed design **may be** performed **prior** to construction [IE during SWE **Design** KA 'phase'], much design work is performed during the construction [SWE **Build**] activity."*

o1.3 The SW Process p9 (AKA **Dev MO**, **Dev Framework**)

5 Phases (can **overlap**, can **recur** later)

o- Made famous by Waterfall Dev MO – 1-st "Big Bang" style Dev MO (AKA BUFD = Big Up-Front Design)

(*) All Projs use these "big 5" – (Eg, you can point to stuff in each Dev MO that looks like each of these)

o- **Comm**/Reqts (Talk w Cust/Users/SMEs, write Reqts) – (SME = Subject Matter Expert)

→ High-level Reqts → **BC Before-Contract**

→ Detailed Reqts → **AC After-Contract**

o- **Plan**/Estim = Estimation

→ BC: **Feasibility** Estim (& Prelim WBS "whibiss") (AKA "**Go/No-Go**" Decision)

o-- **WBS = Work Breakdown Structure** = hierarchical decomposition of tasks & sub-tasks down to "units"

→ AC: WBS+Dependencies+Assignments → Gantt Chart + Deps (sometimes Deps are in a PERT chart)

o- **Model**/Analysis/Arch/Design

→ BC: Todo High-level Reqts (and maybe high-level CRC arch which has been hand-simulated, with users)

→ AC: more user-level (ie, Top Story) arch, box-linkages, and major data parts/flows/pkg'g

→ Brooks's **Silver Bullet**: test/**verify** the model can do the reqts job & (most especially) validate that users will like it

o-- (approximately) Never Done – maybe a **quick awkward prototype** might be built, **maybe** users see it

o-- Architecture can be slightly different that Modelling/Analz cuz it can include stuff below the level of the Problem Domain but above the level of Detailed Design (DD – non-std acronym)

– Arch and DD sort of blended together

o- **Build** – major ramping up in staff (and hence in expense, cash flow – >= 90% of project cost)

→ Detailed part guts Design (DD) – often not down to the unit task level, yet

→ **Unit Design** (sometimes called DD), Build, Unit Test

o-- Lots of RT bug-find-fix – (Agile's TDD is a heavy-weight way to avoid RT-bugs)

→ **Integration** Test (joining units) – often treated as part of I&T (these are fuzzy terms)

o-- Lots of RT bug-find-fix (good APIs are never enough)

→ **I&T = Integration & Test** (of major parts, & with H/W if needed)

o-- Lots of RT bug-find-fix (good APIs are never enough)

→ **Prelim V&V** – at least Verify works per Spec – can reveal need for a bunch of rework, eg for performance

o-- Performance Test, etc. as spec indicates

o- Ship/**Deploy**

→ Test in Tgt H/W Envir maybe before ship (eg, works in ALL the major browsers?)

More – Ship/**Deploy**

→ Test deployed configuration of features

→ Test Install procedure

→ Make UG, Install Gde

→ Final V&V, Verify works per Spec w Cust, Validate users like it

→ Maybe Accept Test (usually on cust/user site) – this could be just Verif, or include Validation – fuzzy

Lab:

Project Infrastructure – HTML + JS + Node + Express (half of the M**ERN** stack)

assets/js-node/nodejs-up-run.zip

"Hello World!" app with Node.js and Express – by Adnan Rahić – (one of many up-and-run webpages)

https://medium.com/@adnanrahic/hello-world-app-with-node-js-and-express-c1eb7cfa8a30

Sidebar – Asserting Code Invariants to Catch RT Bugs
Helper for Rule #2 Bugs (Kill Bug Hunts)
- o- To avoid RT bugs
- o- During development
- o- At a specific point in your code
- o- When you're sure a **variable value combo** has a **simple checkable property – and should always be true**
  EX: **((1 <= xx) && (xx <= 10))** // xx in 1..10
- o- Because you planned your code to be that way
- o- Add a Helper, AKA an **assert-or-fail**
  EX: ***assert(((1 <= xx) && (xx <= 10)), "Err 17, bad xx")*** – test cond failure causes program to stop
- o- So your pgm **will stop right there** with a code **line-number** & source **filename** & your msg
  EX: "Assertion Failed at line 23, in file myfile.cpp: Err 17, bad xx"

(*) Rather than stop 2 minutes later 20 lines away – or 20 mins later a thousand lines away
Why would the pgm stop later?
- o- **Fault** = an internal bug event that doesn't disable part of the program (immediately)
  - o-- Users don't see it, the bug event results
  - o-- One bug event can cause a "cascade" of other bug events 'downstream"
- o- **Failure** = a bug event that the users see -- some UI or output doesn't work

(*) Assertions (AKA Invariants, == "Always True Things")
- o- They are also used in Formal Methods, to PROVE correctness of (small) programs

Alt "assert" is a conditional print [AKA console.log("Err at ...")]
- o- But you have to also fill in the filename and line number

EX: if ((1 <= xx) && (xx <= 10)) { print("Err 17, bad xx" + "file: Foo.xxx Line: 21")

(*)* Pbb Disable Asserts in deployed code & (a frill) maybe replace with graceful shutdown (save user data)

Sidebar – WhereAmI Prints
Putting prints (eg, **console.log**(–)) every few stmts is a handy way to see run progress
- o- Helps boost morale

Sidebar –Stubs/Mocks
Q: How do you build one unit and **test it without the rest** of them – your other-unit helpers?
  o-- Equiv: one object (& its methods) without its helpers/callers?

A: 1) Defining the API you think you need from the other CRC agents – a "rough API"
2) And "stubbing" (creating **fake) agents** which get fixed (ie, constant) args (the EI in EIO) and return the correct result (the EO) by **EIO table lookup**.
3) Of course, you only need to stub out the units that have yet to be built
4) Using Rule #0 (Fast), you may have to revise some APIs as you develop your agents, but that is normal as you gain further insight into the agent-agent interactions.

o1.3.2 Umbrella Activities p11 – (AKA Administrative Tasks)
[P&M]: 8 examples:
o- Tracking Mgmt, Risk Mgmt, QA, Tech Rvu, Metrics, SCM/CMS/VCS, Reuse Mgmt,
Create "Work Products" = non-code docs req'd by contract or by upper mgmt or by your mgr
o- In general, Admin Tasks are stuff that doesn't end up being shipped,
o- Except the program's source code, and the design docs and the arch docs and the reqts doc (AKA the spec)
o-- On the theory that the spec drives the arch/model, drives the design, drives the source, drives the exe
o-- And also include the EIOs cuz they drive the arch & design
o- But (maybe) don't include the test code, cuz they're in a fuzzy in-between world
o-- but maybe include the test code cuz it's close to the EIOs, and it's a "key" crutch to getting code working

Ch 2 S/W Dev Methodologies AKA "Process Models" (sometimes AKA SDLCs == SW Dev Life Cycles)
SKIP 2.1 A Generic Process Model 21 – (but Fig 2.2 diagrams are sort of related)
o2.2 Defining a Framework Activity 23
o- "5 Framework Activities" = the 5 phases: **Comm, Plan, Model, Build, Deploy**
o- Ignore "inception, elicitation, elaboration, negotiation, specification, and validation" – it's from "RUP"
o2.3 Identifying a Task Set 23
o- Note the boxed "Task Set" (no Fig number) – but it's just a sample checklist – you'd pick what you need
o- In real dev, we don't create "task sets" as such – we build a WBS
o2.4 Process Assessment and Improvement 24
o- SEI's CMMI is all about improving your SW Dev MO – a good idea, but biased toward big slow companies
o2.5 Prescriptive Process Models 25 – Here are some actual SW Dev MOs
o- [P&M] describe – The Waterfall, Prototyping, Evolutionary, Unified (AKA "RUP" – Rational Corp.'s)

SWE Dev MOs – (not in P&M)
2 Kinds: **Plan-Driven** & **Agile-like**
o- **Plan-Driven** (AKA Big Bang, Heavyweight, Traditional)
o-- Key: BUFD = Big Up-Front Design/Arch/Model (and Reqts & Plan)
Waterfall (pix are from Software Engineering Practice, Hilburn 2021)
o- Note phases: A) Model is folded into Comm/Reqts, and B) I&T is its own phase (cuz it takes so long)
and C) Plan is MIA (it's assumed) and D) "design" == Arch + Detailed Design

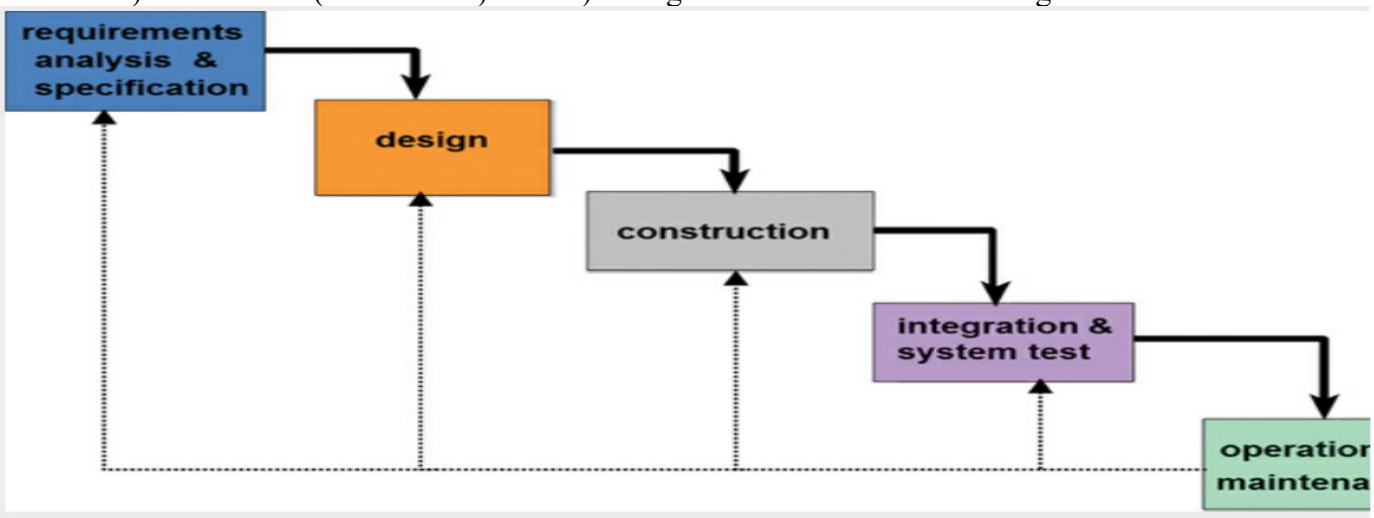


Figure 2.1 Waterfall model.

Spiral

(Barry Boehm, a modification of the Waterfall – mini-Waterfalls each in one of the "loops")

o- **First** loop Starts with **Prelim Reqts** – Prototype shows project Model is doable – ends with a Dev Plan

o-- NB, cust (and maybe users) (Gov agency) sign off on each successive Budget

o--Cust (and maybe users) **look at results** each time, but **no hand-on** & thin feedback

o-- Comm & Model phases smeared over First & Second loops

o- **Second** loop Prototype drives **final Reqts** AND a Verification (**System Test**) plan (AKA **what to test for**)

o- **Third** loop Prototype drives **Arch + Detailed Design** (DD) – Still not "**ramped up**" to full staffing

o- **Fourth** Partial loop Prototype used to show DD looks okay – then Build and I&T phases, (and Deploy)

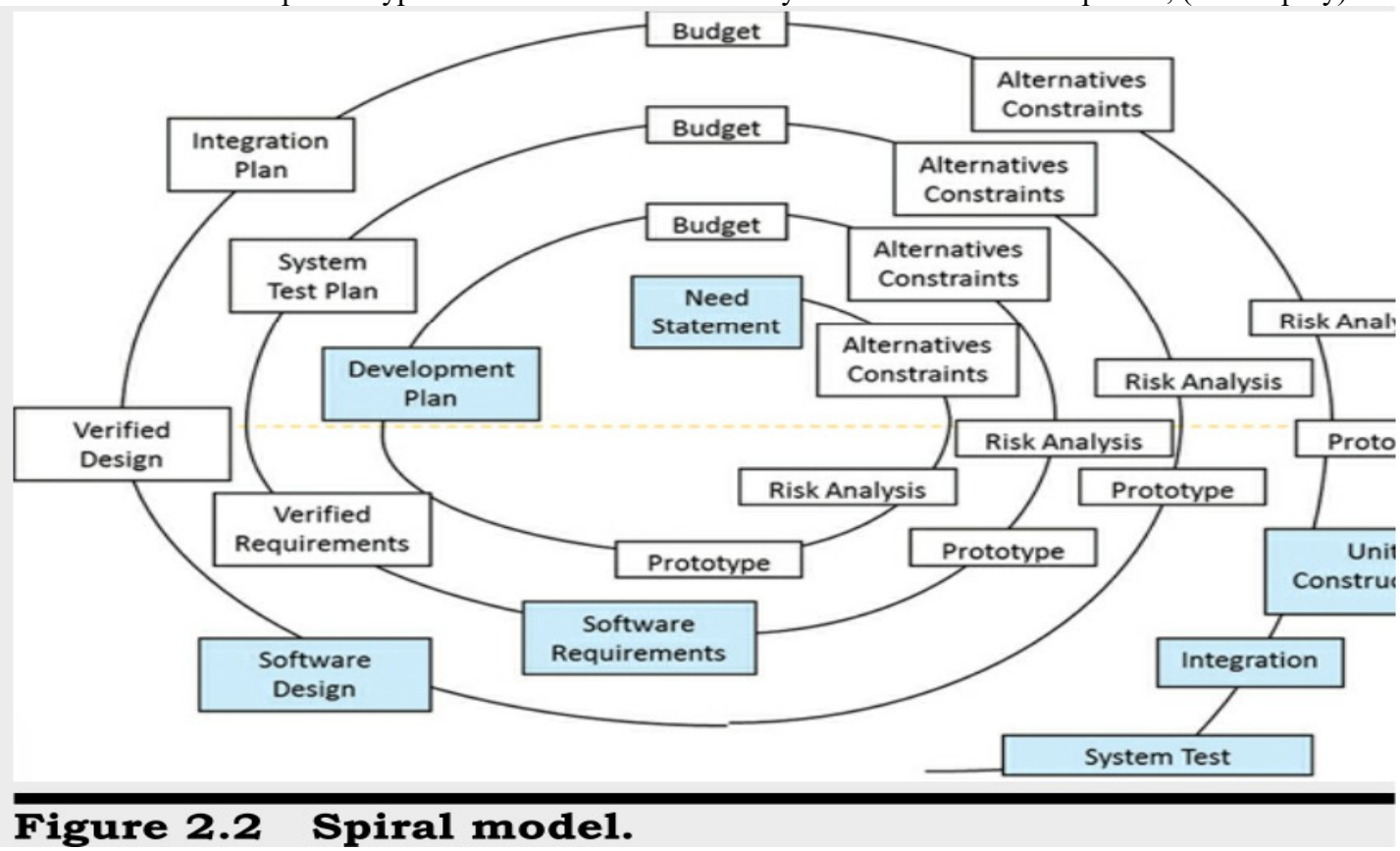


Figure 2.2 Spiral model.

V-Model (used mostly in Europe)
- o- Comm == Need Stmt & Spec are **sometimes distinct docs** in big gov contracts
- o- Model == Arch (a synonym, when dealing with the user-domain)
- o- Build == Module (Detailed) Design + Coding + Unit Test

** Big Idea is that each LHS item has its own RHS Test
And the Dashes show Levels – where the RHS Test is spec'd/built before moving down to the next level
- o-- In real life, it's complicated – and still no user feedback till Deployment

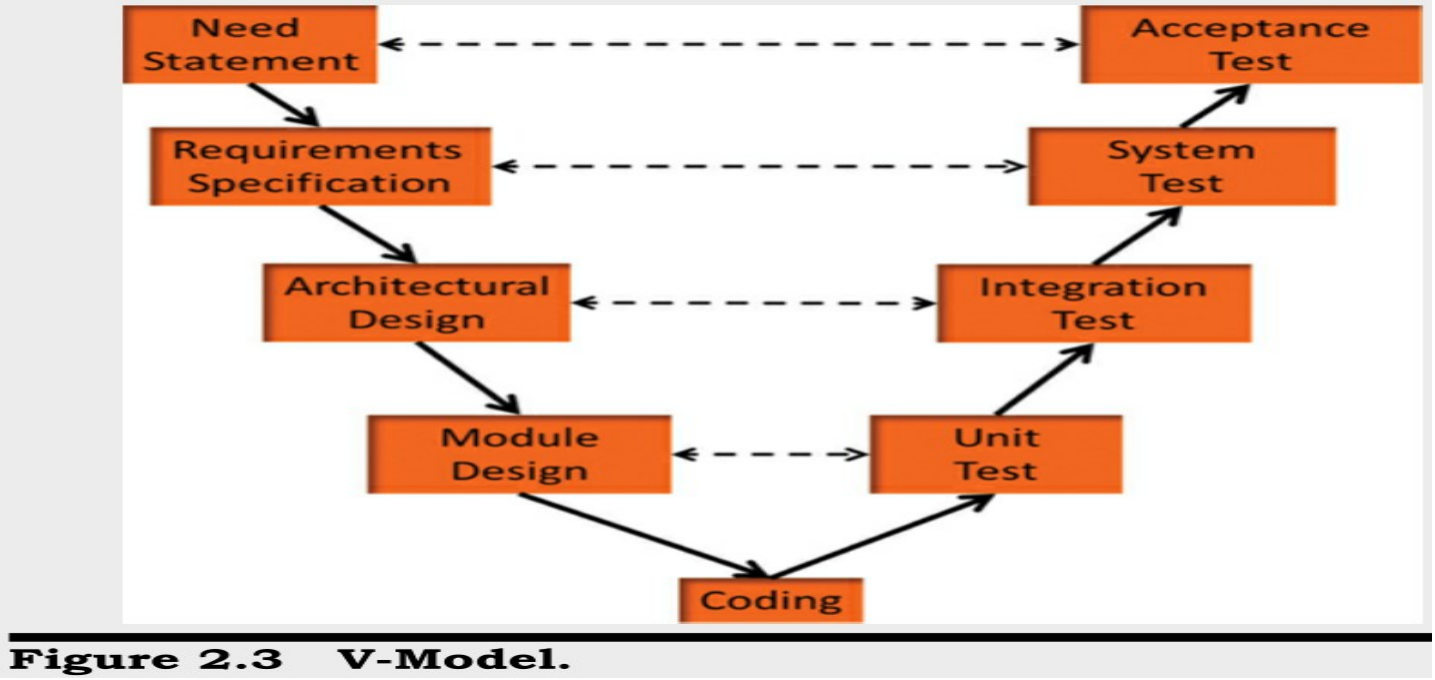


Figure 2.3 V-Model.

Big Bang (Plan-Driven) Cons:
(*)(*) Key Pbm: ***long time*** *till hands-on* **user feedback** on suitability – just looking at a proto doesn't cut it
==> 1 of the 4 major causes of project failure
Recall: Complexity big, Mgrs poor, Predictions bad, Comm poor (user feedback)

Agile-like (AKA Scrum,XP,Kanban,...)
(*)** Warning: "Agile" now used as a near-useless "buzzword"
(*) Key to Agile: Sml size deliverable (working features) to cust avg each month or less
- o-- Get real user feedback very early, very often
- o-- Keep added complexity Sml (to Med)
- o-- Keep predictions Sml (much safer to estimate for 2-4 weeks of work)
- o-- For real Agile – teams self-org ("poisoned pill"), so fewer mgmt issues, plus better morale (on avg)

Other "major" SWE Dev MOs –sort of emulate Agile

Incremental MO (AKA "Stovepiping")
- o- Loop till Done: Build 1 feature-set then I&T then Post-mortem
- o- **No delivery**, but sometimes the users get a peek & do feedback
- o- **First** Increment usually builds a **substrate** Plus a **Sml slice** of the Features (ie, they work)
  - o-- Substrate is stuff the Feature slice needs to work (eg, add the DB & GUI & networking etc.)
  - o-- **Full** system substrate is **not built** (not shown) – just enough to get the Feature slice up and running
- o- **Each Feature slice** built on top of the substrate is called a "**Stovepipe**" coming vertically off the "stove"
- o- Each **increment** is (in theory) **usable** by users – but typically only at the devrs' site (not very thorough)
  - o-- So, (in theory) you can get hands-on user feedback after each increment
- o- Each increment takes **a few months**

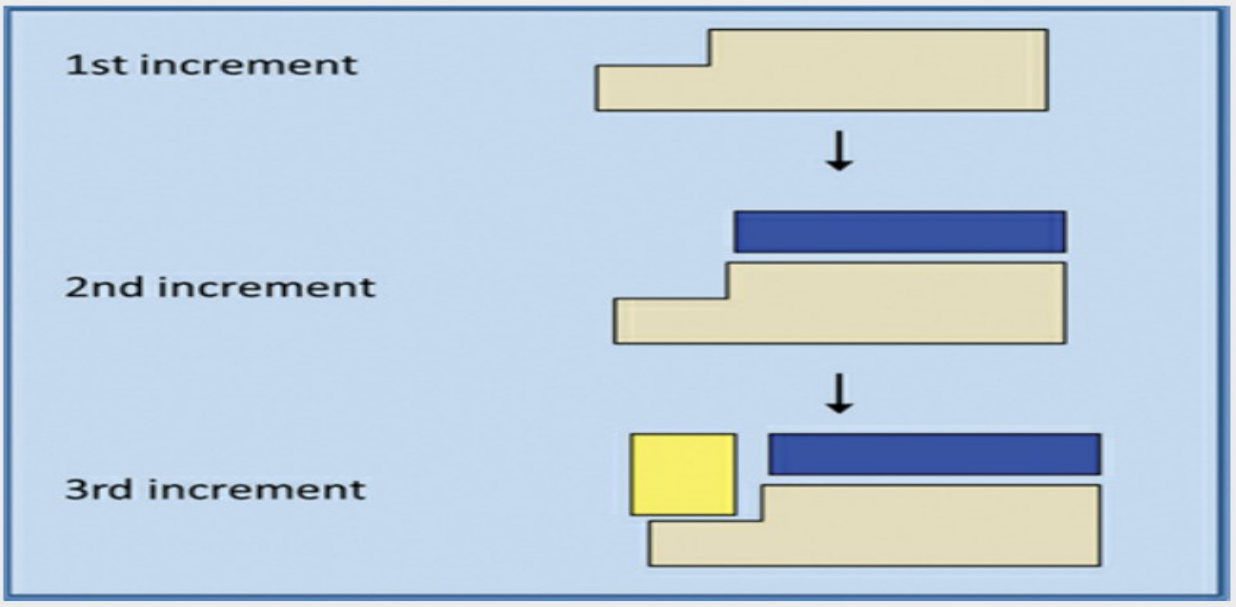


Figure 2.4 Incremental model.

Unified Process – (and Rational corp's Unified Process (RUP)
- o-- UML + UP is a merger of the Triplets' OOP modeling mechs (1999 Jacobson, Booch, & Rumbaugh)
- o- UP picked new names for the phases – (some people like being different)
  - Comm == **Inception**
  - Plan + Model == **Elaboration**
  - Build == Build (**Construction**)
  - Deploy == **Transition**
- o- UP is an **Incremental** MO – but they deliver usable increments – usually every few months (not weeks)
  - Hence, it gets early-ish hands-on user feedback
  - o-- But it is still somewhat heavy-weight (tools & docs) – especially using a UML tool heavily

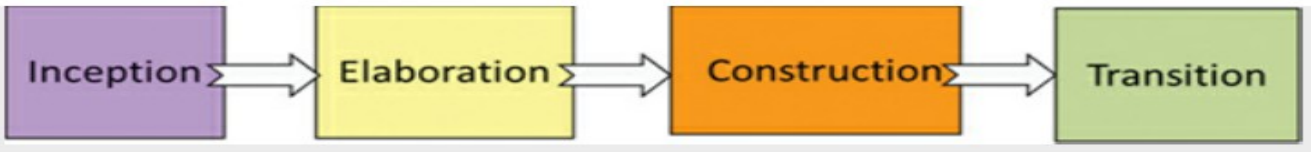


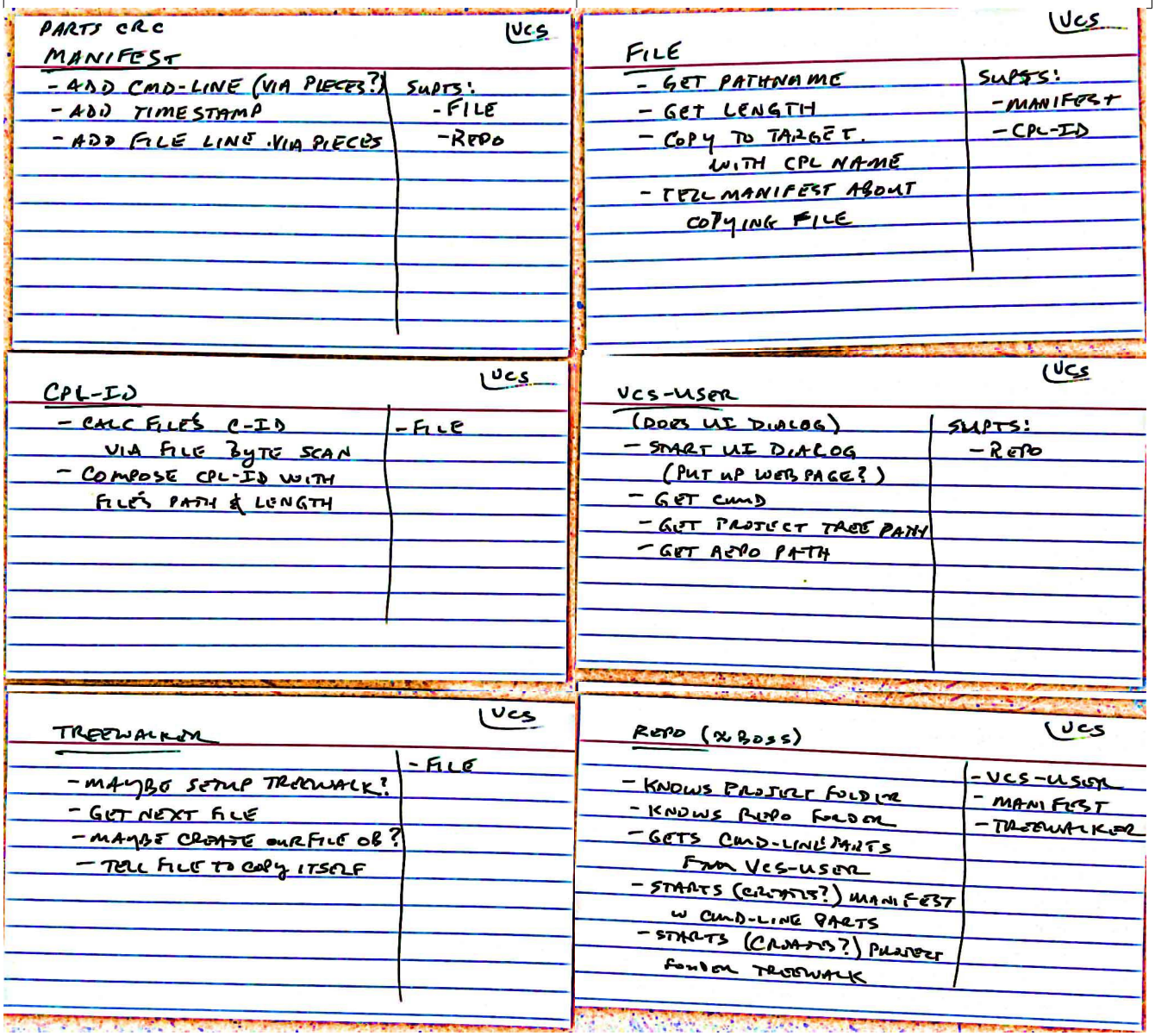
Figure 2.6 UP process.

Lab:
- x- Project VCS User Scenarios doc
- x- P1 assignment PDF

CECS-343 SWE Intro — CRC Cards Arch Example

Parts CRCs Example
o- Input: The UCs built earlier
UC: VCS – 1. **Create Repository**
2. Create a new Repository in a folder for a Project Tree of folders/files.
3. **Summary**: See PDF assignment
4. **Role**: VCS User
6. **Main-Scen**:
o- Get source & target;
o- Start new manifest file with cmd-line & timestamp
o- Traverse source tree: for each file: {
o- Create file's **Art ID** ["**CPL**" in pix, below]
o- Add file line to Manifest
o- Copy file into repo w Art ID filename }
5. **Agents (Mgrs/Handlers)**: Repo, Manifest, File, Art-ID, Treewalker, VCS-User

Key: Try to limit couplings in the Arch graph.
Note: How UC(s) **mismatches** Parts CRCs, also note awkwardness in **picking Agents** & **placing responsibilities**. (Cards are cheap. Try ideas.)
Q: Who should be **boss** of (or collab with) who?



Agile Manifesto – Preferred Values – (This over That)
- o- 2001 – https://agilemanifesto.org

4 Preferred Values
"We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value:"
- o- Individuals and interactions over processes and tools [II > PT – mnemonics]
  [Processes like heavy-weight big-bang doc-review gates; Tools like heavy-weight UML layout pgms]
- o- Working software over comprehensive documentation [WS > CD]
  [Working Features over heavy-weight big-bang docs]
- o- Customer collaboration over contract negotiation [CC > CN]
  [Creating customer **goodwill** and **feedback** over creating protection from possible future **legal battles**]
- o- Responding to change over following a plan [RTC > FTP]
  [Letting customer/user feedback **drive** the project over big-bang comprehensive "5-year" plan]

"That is, while there is value in the items on the right, we value the items on the left more."
(Next time – motivations & the **12 Principles** behind the Agile Manifesto, see it at same website)

Goal: UScens -> UCs -> CRCs -> CRC hand-sim -> validated (users like it, you understand it, as a first cut)
- o- First cut, because as users get to play with features, they will correct (themselves and you) and innovate (think of new stuff)

How to Begin [CS-Style]
UScen-UC-Model Pipeline [No Planning, and Modeling is for Reqts Understanding & User Feedback]
User comm to Determine Usage
→ **Raw UScens** + (**EIOs** v0 – see if users can give you some example input & it's expected output)
→ Group **UScens by role** – in case it wasn't already done during user Reqts (AKA What they want) mtgs
→ Extract **UCs' Tops** (by Role+Task) & Pry em – Top == missing #6 Main Scen (AKA the How To Do It)
  EX: VCS Use Case (UC) – **#1 Create Repository** (OneVerb+Object(s) format, from the UC Tag-line #2)
    #2 Tag-line (from the UC Summary #3): Create a repository in an empty folder from the given project source tree (including a "snapshot" of "all" its files, and a manifest listing them).
    #3 Summary (one task from one **UScen**+one role): The users need to keep track of various snapshots of their communal project, 1) in case they have to "rollback" to a previous good working copy, or 2) in case they want to preserve several experimental feature versions (branches of mainline) of the same project, or 3) in case they want to work on a bug fix without touching the "mainline" project code (branches of mainline) until the fix is well-tested, or 4) in case they want to ship the product with a different mix of features to different customers (a diff mainline for each feature "cluster"), or 5) you want to ship your product for different operating systems (a diff mainline for each).
    [#4 User Role][#5 Major Data Parts/Agents][#6 Main Scenario, AKA No-Frills Procedure to follow]
→ Sketch **UC Steps** (AKA the #6 Main Scen) + EIOs v1 (You Fit/Adjust EIOs v0 to each UC)
// We now have at least one UC filled out (except for any "#7 Extended Scenarios", much less important at first)
// NB, the Main Scen doesn't talk about agents (who will do the work) – but we need some agents
→ Create **CRC Card** for each closely-related group of actions for a given role == the role/user's helper agent]
  o- Typically, an agent/Card will "own" one of the UC Major Data Parts – possibly helping multiple UCs
→ **Check that linkages** between CRC Card/agent collaborators make sense – revise Cards as needed

→ Hand-simulate each UC (its Main Scenario) by given each player a CRC Card/agent
  o- "User" kicks off UC request by calling on main PA/agent for help (usually a UI CRC)
  o- PA follows UC Main Scen (3-8) steps, calling on other agents for help & getting results
  o- Are their gaps during hand-sim processing? Do the Cards make sense? – revise Cards as needed
→ Note UC Extras + EIOs v2 for them – #7 Extended Scenarios == Err Handling, Nice to Haves
→ Extract Qualifiers Reqts if any + Pry em – Esp Perf needs, Distrib H/W Hosts, and Legacy Fit
→ Eval Model v Quals – Does the model still (seem to) work with all these qualifiers/constraints

Lab
prjs/343-Parts-CRC-Cards.pdf – based on UC in P1's VCS Create Repo PDF

More Project Infrastructure – HTML + JS + Node + Express (half of the M**E**R**N** stack)
assets/js-node/nodejs-up-run.zip
  o- Show localhost:3000 webpage running – bring up node cli box & start app.js first
  o- Show older version – js-1.html and app.js
  o- Show newer version, with edit box + button – js-1-edt.html and app.js mod to catch box & reply to it
  o- Discuss JS code for HTML webpage
  "Hello World!" app with Node.js and Express – by Adnan Rahić – (one of many up-and-run webpages)
  https://medium.com/@adnanrahic/hello-world-app-with-node-js-and-express-c1eb7cfa8a30

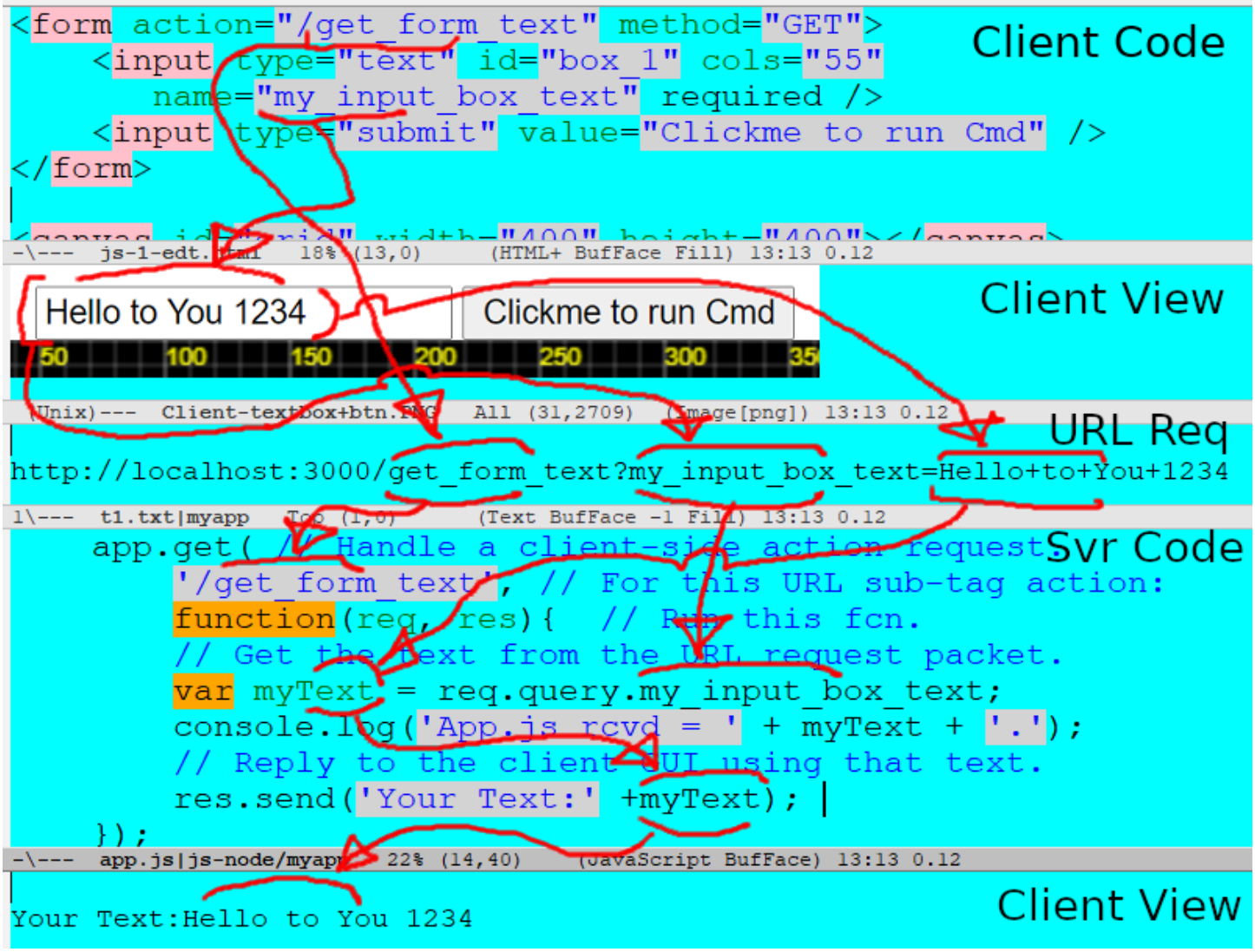
Lab:
Git VCS Terms
"repo" -- git has one per user – P1 has a single centralized repo (simpler)
  o- git: it's in .git sub-folder of your projtree folder -- no central repo – P1, it's at a fixed folder loc
  o- "git init' -- create an empty repo in curr folder – P1, "create repo" creates the repo from user's projtree
"Working Folder" -- root folder of projtree, inside is .git – P1 your projtree is your "Working Folder"
"commit" == P1, a check-in/snapshot of your projtree, not just a selection of your files
"commit object" == P1, manifest
"clone" == P1, a check-out
"master" == P1, 1st snapshot/tail of a branch – either create repo result, or check-out result
"blob" == P1, artifact (version of a file), OR a subfolder name
"branch object" == P1, a leaf-label on a branch-leaf
"tag" == P1, label on (alias for) a snapshot
"git hash-object ..." == P1, return the hash/artifact ID of file
"stage" == P1, files to check in on next commit – we check in them all (except dot-files)

Lab:

How the Client and Server Work Together:

Client passes data to the Server & how the Server does work & passes a result webpage back.
Client Code has a FORM section – 1+ named input boxes, an "action" URL "folder", and a "submit" button.
User fills in the input and clicks the submit button – forming a URL REQuest with "query" parameters+values.
Server (app.js) "catches" the URL REQuest by "folder" name and calls its handler function.
Server handler fcn extracts the query parameter values by parameter name from the URL REQuest.
Server handler fcn does any server-side work needed.
Server handler fcn composes a result webpage (as needed) and sends it as the RESult back to the Client.
Client-side User sees the result webpage – which may allow the user to create another request.

Picture: key-name linkages in the Client-to-Server-and-back-to-Client flow.



Agile – motivations & the **12 Principles** behind the Agile Manifesto

4 Motivations for Agile – to avoid the worst excesses of projects

- o- **Avoid death march** syndrome – (AKA when the project is behind, make em work longer hours)
  - High pressure, long hours, extra stress, lower productivity, burnout
  - Leads to high staff turnover, low retained product knowledge among dev group
- o- **Avoid user surprises** (AKA users don't like it when project is finally completed)
  - Show users incremental working features, get their feedback, uncover misinterpretations early
  - Rapid "Prototypes" (fast incremental "add-a-feature")
- o- **Avoid overly rigid design** (AKA don't poor batches of concrete on design until the users like it)
  - Design with low loose coupling – among self-contained helper agents (easy to plug-replace)
  - Design top-down with domain layering – this big agent calls on these smaller-agent helpers
- o- **Avoid Gold-Plating** syndrome (eg Rule #1 Optim)
  - Built only the minimal needed – no optim, no reuse, no unvalidated users-will-need-it stuff
  - Ensure entire team knows what's going on – no duplication or misdirection built by team members
  - Refactor only to clear local confusion – (Refactoring == "Making **local** code simpler")
    - o-- **JIT** == "Just In Time", when you need it
  - **YAGNI**: "You Ain't Gonna Need It", so don't build it

Agile – **12 Principles** – the better half

#1. Our highest priority is to **satisfy the customer** through **early and continuous delivery** of valuable software. (Always very short development/deployment schedule to get user feedback, repeatedly)

#2. **Welcome changing requirements**, even late in development. Agile processes harness change for the customer's competitive advantage. (At a minimum, we can write 'em down for the next "Sprint")

#3. **Deliver working software frequently**, from a couple of weeks to a couple of months, with a preference to the shorter timescale. (Scrum has fixed repeating time-boxes, others deliver when Feature(s) ready.)

#6. The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**. (Hence, everyone in short walking distance, <= 12 members, & Standups)

#7. **Working software is the primary measure** of progress. (Working Features is intended)

#11. (Poisoned Pill) The best architectures, requirements, and designs emerge from **self-organizing teams**.

Agile – **12 Principles** – the other half

#4. **Business people and developers must work together daily** throughout the project.

#5. **Build projects around motivated individuals**. Give them the environment and support they need, and trust them to get the job done.

#8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to **maintain a constant pace indefinitely**. (AKA avoid Death Marches)

#9. **Continuous attention to technical excellence** and good design enhances agility. (KISS, Rule #0 Fast)

#10. Simplicity--the art of **maximizing the amount of work not done**--is essential. (Rule #1, Optim)

#12. At regular intervals, the **team reflects on how to become more effective**, then tunes and adjusts its behavior accordingly. (AKA Retrospective, Lessons Learned)

o3.4 Scrum 42

o3.4.1 Scrum Teams and Artifacts 43

Scrum Artifacts (AKA key things involved in the Scrum process)

- o- **Sprint** == Regular time-box to comm-plan-model-build-deploy next **feature (set) slice** (1 to 8 weeks)
  - o-- Typically, same duration for each Sprint – once the team figures themselves out
- o- **Product Backlog** == Stuff cust/users want ("Features" is typically used)
  - o-- **Top 6-ish are prioritized** – rest are too far in the future to waste time **pry**'ing (Rule #1, Optim)
  - o-- At start of each Sprint, **pry**'s may be changed
- o- **Sprint Backlog** == Product Backlog stuff team thinks can **fit into next Sprint**
  - o-- Selected in pry order (ordered by cust/users, or their proxy, the **Product Owner**)
  - o-- Each feature eval'd by team to determine its effort (in **Story Points**)
  - o-- Team determine how many Story Points in all can fit into next Sprint (which may include safety slack)
- o- **Story Points** == Effort measurement of each User Story (but some teams have features in a **UStory**??)
  - o-- Usually several **UStories** per Feature
  - o-- Story points are in funny sequence (cuz of uncertainty) – eg Fib seq (1, 2, 3, 5, 8, 13, ??21)
  - o-- Story points can be translated into staff hours, but **never done** – to avoid mgmt "gaming the system"
  - o-- Points assigned during team planning session – eg, via **Planning Poker**, or Affinity Planning
  - o-- **UStory** has a format style (For **UNeed**, do Action/Behavior, to get **UValue**)
- o- **Daily Standup** Meeting
  - o-- At start of day (eg, 9am)
  - o-- Whole team present (& including as many other stakeholders as can make it)
  - o-- 15 minutes long (approx.) – too short to sit
  - o-- Clustered around the Progress Board (eg, cork board or whiteboard) (not as good – electronic "board")
  - o-- Each team member answers THE 3 questions
    - Q1: What did you **complete** yesterday? (Bad alt: What did you **do** yesterday?)
    - Q2: What do you plan to **complete** today? (ditto) (Rule Half-Day)
    - Q3: Any **blockers**? – AKA Are there any **obstacles** holding you up? (You've done "due diligence")
- o- **Progress Board** (AKA **Kanban** board)
  - o-- Columns represent left-to-right progress from Sprint Backlog features/UStories to checked tasks
  - o-- **Cols**: **Ready** > **Working** (grabbed by a team member) > **Done** > **Checked/QA** (by another teammate)
  - o-- May append a Col for out-of-sprint Issues or **Tech Debt** (== code or arch to clean up/rewrite later)
  - o-- May have a max **WIP** (== Work-In-Progress) limit to avoid someone working two tasks at a time
  - o-- (*) **Story Points** are "credited" when all Feature sub-tasks are finished, hence "Working Feature" – similar to **EVM** project tracking
- o- "**Scrum Master**" == team member is referee for what is and isn't Agile
- o- **Product Owner** (PO) == representative/proxy of cust/users w.r.t. Assigning pry's to Features
- o- Cross-Training – each team member should learn another's specialty area (to know more than one area)

Planning Poker – AKA Crowd-sourcing an answer

- o- Each team member has a card deck – one card for each Story Point value (eg, 1, 2, 4, 7, 11, 16)
- o- UStory is read/shown to everyone
- o- Each team member picks a card and places it face down (till all are done) – guess on Story Pts it'll take
- o- Big Reveal – cards are turned face up
- o- Discussion if not all in agreement – argue in favor of your choice if you want to
- o- Agree/Consensus on Story Points for this UStory

Lab:
Q: How can the **app.get fcn create a repo** with all the source projtree files?
A: The app.get fcn will examine the source-path projtree and copy all its files to the target-path empty folder, ...
Q: How can the **app.get fcn produce a new** (ie, the next) **webpage**?
A: Once the repo is created, simply return a webpage-as-a-string, spliced together from fragments, static or computed, as needed – via the res.send(my_string) call, for example
Q: What does the **target webpage-as-a-string** contain as **fragments**, roughly?
- o- **Header** stuff
- o- Previous page's user **query**/command **answer**
- o- Another Form for the **next** user **query**/command
- o- **Footer** stuff

And **splice these 4 strings** together for the answer. (eg, with JS '+' concatenation operator)

Here is one variant, but there are many other ways to construct the next webpage-as-a-string:
Header stuff – like this

```
<html>
<head>
<title>VCSish Command Central</title>
</head>
<body>
<h1>VCSish Command Central</h1>
```

Footer stuff – like this

```
</body>
</html>
```

Previous page's user **query**/command **answer** – like this

```
<h4>Create Repo was successful! </h4>
```

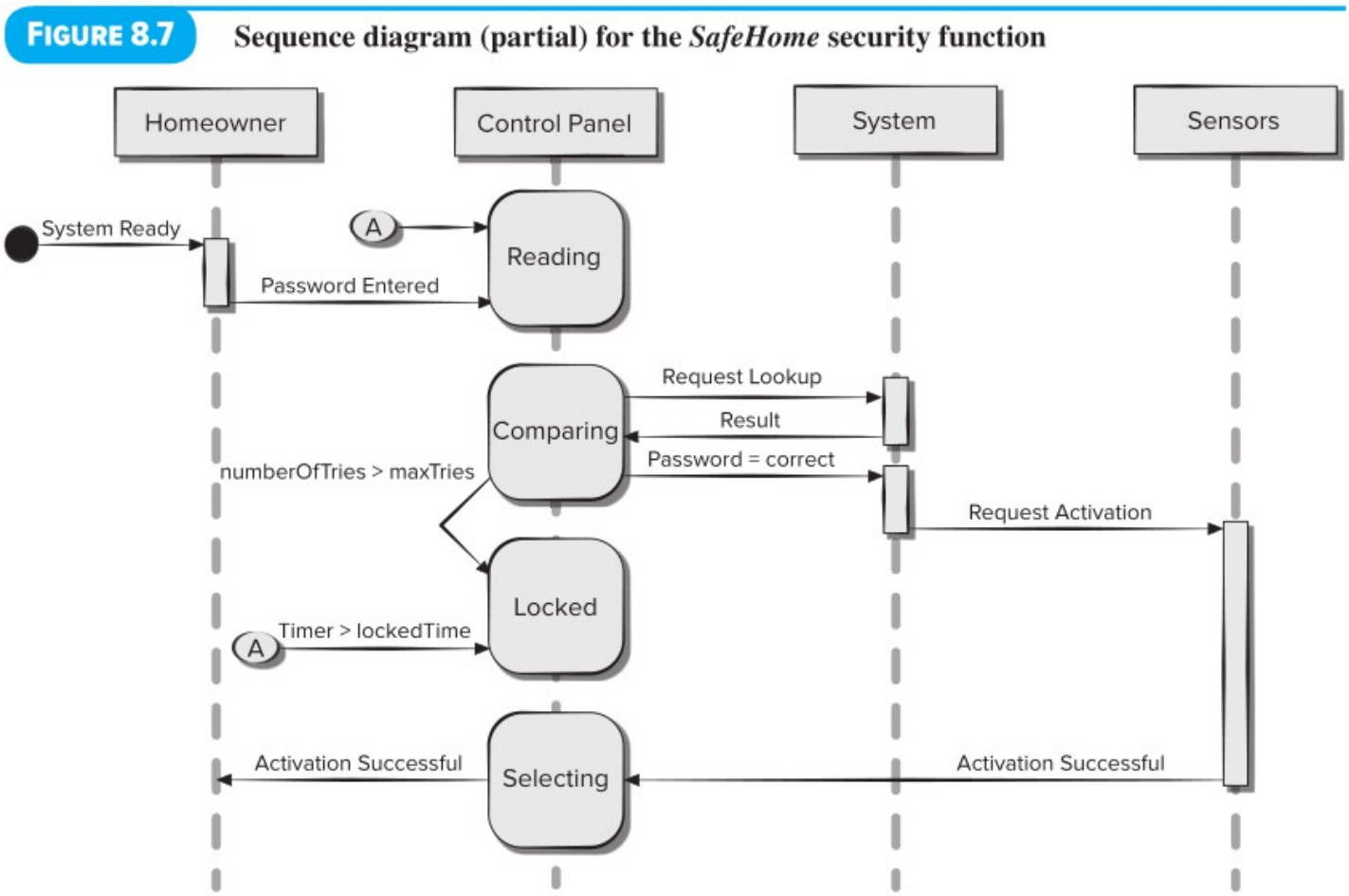
Another Form for the **next** user **query**/command – maybe like this
- o- Same as the first form – but either escape the internal quotes, or use single-quotes outside

EX:

```
'<form action="/create_repo" method="GET">
  <label> Source Path:
      <input type="text" id="box_2" cols="55"
             value="(Replace me) C:\\Time\\for\\Squiddie\"
       name="source_path" required /> </label>
    <label> Target Path: <input type="text" id="box_3" cols="55"
       name="target_path" required /> </label>
    <input type="submit" value="Create Repo" />
</form>'
```

Lab:
UML (Pole) **Sequence Diagram**
- o- For seeing the interaction sequences between multiple agents
- o- Agent (AKA Mgr) atop each pole
- o- **Time** flows **downward**
- o- Box in the pole indicates processing activity
- o- **Arrow** between poles indicates a **message or call**
- o- Can be fancy or simple – main thing is to make interactions clear
- o- Originally for distributed asynchronous systems

(*) This is what a CRC Hand-sim would look like if diagrammed



Lab (for next time, a reminder to me): – how to do multi-file development in Node.js:
- o- Bar.js wants to use a function or var from Foo.js (both in same dir) – how does it work?

In foo.js: – use "exports." on vars & fcn names you want accessible to Bar.js

```
exports.myvar = "wow";
exports.myfn = function () { console.log( exports.myvar ); }
```

In Bar.js, in same dir:

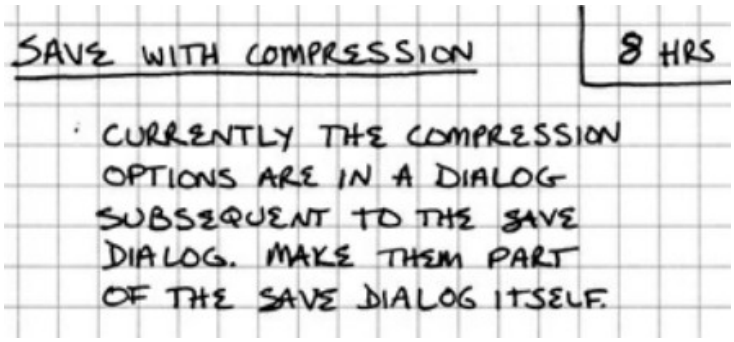
```
let fool = require( './foo.js' )  // eval's foo.js creating obj w slots
// Access each exported Foo var/fcn with its name via the var 'fool'
console.log( fool.myvar );
fool.myfn( );
```

o3.4.4 Sprint Review Meeting p45 – (after code is done-ish)
- o- **Demo** (to team, PO, & stakeholders) of the **completed Feature Slice** to be Deployed to User
  - o-- Completed Features may be a subset of planned Features – planning estimates aren't always accurate

(*) Important as a **Morale boost** for the team (and all others present)
- o- NB, during sprint, as each Feature is completed, it should be demoed to the PO (cust/user proxy) for (proxy) feedback – don't surprise the PO/cust/users at the end of the Sprint

o3.4.5 Sprint Retrospective p45 (AKA "Lessons Learned")
- o- Short mtg – half-day or less
- o- (Politely) Looking for improvements in how to do S/W for next sprint (with the SM as referee)
- o- Updating performance "**velocity**" (AKA **story points per time**) based on Features Done this last sprint

o3.5.1 The XP (Extreme Pgmg) Framework p46 (+ CI == Continuous Integration)
- o- Like Scrum, but Predates Scrum
- o- **Pair Programmers** – two staff at one computer – both involved in unit EIO, design, code, testing
  - o-- Keep each other on task
  - o-- Brainstorm refinements to the system & Clarify ideas
  - o-- Take initiative when their partner is stuck, thus lowering frustration (improving Morale)
  - o-- Help each other to adhere to the team's practices (more transparency gives better S/W)
  - o-- (Another Poisoned Pill) Mgrs say – Why pay two people to do what one person could do
- o- Sample XP Story (from Extreme Programming Explained: Embrace Change, 2ed, 2004, Kent Beck)
  - o-- Title, Summary, Time estimate



- o- **2-3 Month Planning cycle**
  - o-- Lessons Learned – esp. bottlenecks from outside the team
  - o-- Plan major areas (AKA "**themes**") to work on – stuff cust/users & biz needs
  - o-- Pick/Create "a quarter's worth of (User) stories" for those themes [They will change with feedback]
- o- **Weekly dev cycle** (used to be slower-paced longer cycle – eg, 3-4 weeks)
  - o1. Review progress to date (actuals vs predicted) [with an eye to improving predictions/estimates]
  - o2. Have cust/user (or maybe proxy) pick a "week's worth" (rough guess) of stories for this next week
    - o-- Stories queued in story pry order (with pry picked by the user/proxy)
    - o-- Team estimates each story's effort (in hrs per pair – eg, 8 hrs)
  - o3. Break stories into tasks (units)
    - o-- Tasks (cuz they are from Stories that can be bigger than 1 pair can do) queued in story pry order
  - o4. Tasks taken (usually) in first-come first-grabbed order (FIFO – AKA a queue)
    - o-- One at a time task per pair
    - o-- Pair estimates task (eg, 1-4 hrs)
    - o-- FIFO helps provide variety for pairs, and helps avoid cherry-picking
  - o5. Start week by writing automated tests (Rule #3 EIO) to run when a Story is complete

o6. Get the stories to pass their tests – and look to ensure users will like the results
o7. Deploy completed stories at end of the week – (here, Stories are equivalent to Features)
o- **Build every couple of hours** – CI = Continuous Integration
** "The **longer you wait to integrate**, the **more it costs** (RT bugs) and the more unpredictable the cost (of fixing them) becomes."
o-- Check in next completed code segment each couple hours, also add its (EIO) tests to the Regression suite
o-- Run automated Build & Regression tests (all should take 10 minutes or less – else fix the process) with your segment tests included

o3.5.2 Kanban p48 (Deliver each Feature as soon as it is ready)
o- "Kanban" in factory (& S/W Dev) means "Progress Board" – (more generally "signboard" in Japanese)
o-- Progress board helps everyone understand what's happening – helps morale
o-- Progress boards have been used for over a century, tho
o- Like Scrum, but
o- No fixed dev time-frame – instead, as a Feature is completed, test-package-deliver it
o-- Reqs solid configuration control (with a branch per Feature), I&T, and Regression testing
** Forerunner of "DevOps"

o3.5.3 DevOps p50 (Development team plus Operations team)
o- **CMS** (Configuration Mgmt System – for all artifacts, eg code, tests, docs)
o- **CI** = Continuous Integration – you check in code & its tests, and they will automatically get run
o- **CD** = Continuous Delivery-ish – you flag a completed feature, and a deployment package is auto-built
o- **CD** = Continuous Deployment – a deployment package is automatically delivered/installed to the customer
o-- Typically, a customer/user **would want control** of this to avoid new bugs during a **critical biz time**
** DevOps means different things to different groups – but **automating as much as possible** is primary
(*)* Key – Handling multiple conflicting "branches" being folded into Mainline automatically w/o errors

Lab

How to do **multi-file development** in Node.js:

o- **Bar.js** wants to use a function or var from **Foo.js** (both JS files in same dir) – how does it work?

In **Foo.js**: – use "**exports**." on vars & fcn names you want accessible to Bar.js

```
exports.myvar = "wow";
exports.myfn = function () { console.log( exports.myvar ); }
```

In **Bar.js**, in same dir (cuz the relative path to Foo.js shown below means "**same directory**":

```
let fool = require( './foo.js' )  // eval's foo.js creating obj w slots
// This means 'fool' contains an object whose slots were exported names,
//   and the value of each slot is what foo.js put into them.
// Access each exported Foo var/fcn with its name via the var 'fool'
console.log( fool.myvar );
fool.myfn( );
```

Sample Manifest and its "source" in a reasonable format

// Here is what the **source Project Tree** looks like:

```
c:/projs/mypt/bot/a/b/fred.txt
c:/projs/mypt/bot/alice.js
c:/projs/mypt/main-pgm.js
c:/projs/mypt/bot/d/fred.txt   // 2nd "fred.txt"
```

// Assume cmd line (GUI equivalent) was this:

```
C:> myvcs create c:/projs/mypt c:/p1/repo
```

// where "myvcs" is the program name
// arg #1 = "create" // vcs command
// arg #2 = "c:/projs/mypt" // source projtree
// arg #3 = "c:/p1/repo" // target repo folder, empty

// **Sample Manifest created** -- **with** line numbers and **comments** you don't add:

1. `create c:/projs/mypt c:/repo/p1` // **vcs cmd line args**
2. `2021-02-08 11:27:46` // **date-timestamp** of command

// Each **file in snapshot**, format: <art-id.ext @ relative-path>

3. `5F-0027-612E-fred.txt @ bot/a/b/` // 1st file //Fake ArtID
4. `A4-0123-C812-alice.js @ bot/` // 2nd file //Fake ArtID
5. `21-5193-6939-main-pgm.js @ /` // 3rd file //Fake ArtID
6. `01-633E-0991-fred.txt @ bot/d/` // last file //Fake ArtID

<EOF> // AKA "End of File" mark -- you don't add this

// **Same Manifest, without** the added comments – in file `c:/repo/p1/.man-1.txt`

```
create c:/projs/mypt c:/repo/p1
2020-09-08 11:27:46
5F-0027-612E-fred.txt @ bot/a/b/
A4-0123-C812-alice.js @ bot/
21-5193-6939-main-pgm.js @ /
01-633E-0991-fred.txt @ bot/d/
```

And the **Repo** would look like this:
```
c:/repo/p1/.man-1.txt
c:/repo/p1/5F-0027-612E-fred.txt
c:/repo/p1/A4-0123-C812-alice.js
c:/repo/p1/21-5193-6939-main-pgm.js
c:/repo/p1/01-633E-0991-fred.txt
```

From the PDF assignment:
> "and a copy of this manifest file should be placed both in the repo and in the source project tree root folder of the Create-Repo command."

// Here is what the **source Project Tree** looks like after the Create-Repo command:
```
c:/projs/mypt/.man-1.txt
c:/projs/mypt/bot/a/b/fred.txt
c:/projs/mypt/bot/alice.js
c:/projs/mypt/main-pgm.js
c:/projs/mypt/bot/d/fred.txt   // 2nd "fred.txt"
```

We will assume every team developer works on (sharing) the **same single computer** (laptop) – no frills dev.
There is at least **one source Project tree** (AKA "working directory") **per team developer**.
- o- Each of Alice, Bob, Dave, Elise, Fred have their own Project Tree
- o- Alice created the Repo from her Projtree
- o- The others each checked out a particular snapshot to initialize their own Projtree
- o- Everyone makes mods and checks in their latest snapshots
- o- If anyone needs to "rollback" to an earlier snapshot – they check it out into an empty folder

There is only **one Repo per Project** – mirrors the Centralized Repo style (vs Distributed Repo Style of GIT).
(We won't but) If there are two Projects (eg, A330 Flight Control S/W, and also Canal Bridge Control S/W) then each will have its own Repo.

Ch 5 Human Aspects of SWE 74 (Ch 6 in old version)
Effective SW Engr (the person)
Individual responsibility
(*)(*) Support/Ensure your own Morale
Your Word (What you say to others) – AKA "Commitments"
o- Do what you (even casually say) – Deliver on your word
o- Hence: Guard your word, it is your reputation – pay attention to what you say, in detail
o- For future activities – Say "**plan to do**", not "will do"; IE, it is your goal
Helping the "team" (IE, your neighbors, an ad hoc group)
o- Be neighborly – look out for your local people
o- Help others (at least when they ask for help)
o- Be polite – in all your phrases
o- Complement the work of others (when you can) – helps build team morale
Heads Up Truthfulness – (Trying not to mislead people)
Not "Brutally Honest": nobody likes harsh (impolite) criticism
o- Exactly backwards
o- Remember – Everyone has their own view, and their view of themselves is usually quite strong
o Egos are always involved**
So keep your ego quiet, control it
You must Sell your ideas
o Ask a **Question,** don't make a **Statement**
Key: even if you know the answer
Use the word "issue", not "problem"
Use "not a problem" – when you plan to solve it
o- **Keep goals in mind**
o- **Find Stds that give (or support) your viewpoint**, and Ask if they apply
IE, Let the "experts" make your case for you
o- **Show interest in others' ideas** and comments
Show More-Than-Fairness
Always give more than you get in Trade
Kindness and Generosity, and Overdo It
Cut others more slack than they give you
Key: Don't give advice, just answer their questions
If they don't ask a question, you could be stuck – live with it
Don't bad-mouth others – try to avoid gossip
When you give negative remarks, always be unsure and do so as a Question, **Gently**
o- People are sensitive to being led by the nose with questions, so don't ask too many
Be open to possible change
Watch out for your own engineer's single-minded pbm-solving mind set
o- Remember the parable of the **Engineer and the Guillotine**
Attention to Detail
(*)** Always test your code
Always come up with a way to test – very preferably before you design your code
Work Fast
Uncover issues early
Preserve and Run Every Sample Test: Script them as your regression test suite.

Pressman & Maxim: 8th ed vs 9th ed – line by line correspondence of section headings

6. Human Aspects of SWE	Ch 5 Human Aspects of SWE 74
6.1 Characteristics of a SWE 88	5.1 Characteristics of a SWE 75
6.2 The Psychology of SWE 89	5.2 The Psychology of SWE 75
6.3 The Software Team 90	5.3 The Software Team 76
6.4 Teen Structures 92	5.4 Team Structures 78
6.6 The Impact of Social Media 95	5.5 The Impact of Social Media 79
6.9 Global Teams 99	5.6 Global Teams 80

o5.3 The Software Team p76
Toxic Poisons – Jackman 1998 – ("Team Toxicity")

1. **Death march** == "frenzied work atmosphere"
2. **High frustration, via team friction**
   (stress, bad comments, unneighborly, gossip)
3. **Poorly coordinated SW process (follow MO)**
   Who needs what SW parts when?
   How should they be "checked in"?
   Where is the "mainline" copy kept?
4. **Team member roles unclear**
   Who decides what should be worked on, and by whom?
   Who to talk to if a problem arises?
5. **Continuous/Repeated (micro-)Failures**
   Things keep breaking: tools, code, tests
   Unclear when breakage will finally be overcome
   Demotivating

o5.4 Team Structures p78
Cockburn & Highsmith – 2001

o- Talking about a "**strong team**"
o-- Upshot: Process < People < Politics
"People trump Process" – the MO doesn't matter much
But "Politics trump People" – ie, mgmt sabotage
What it means:
People trump process
"If the people on the project are good enough, they can use almost any process and accomplish their assignment. If not, no process will repair their inadequacy"
Politics trump people
"Lack of mgmt/user supt can kill a project"

Project "Triangle" (== "Iron Triangle")
o- (Better) Scope / Features
o- (Faster) Timeline / Deadline / Schedule
o- (Cheaper) Effort / Cost

People make up the Darndest Triangles –

Jerry Ogdin – on Super-programmers
o- Best SW Engr is **10x** better than an Avg SW Engr,
o- Avg SW Engr is **10x** better than a Newbie SW Engr
– from Jerry Ogdin's paper "**The Mongolian Hordes Versus Super-programmer**."
Infosystems (December 1972/1973): pp20-23.

Chief Pgmr Team – S/W Dev MO
o- created 1971 by **Harlan Mills**, a "super-programmer", IBM research fellow
o-- Chairman of the First National Conference on Software Engineering
o-- Chief Editor for IEEE Transactions on Software Engineering
o-- IEEE created the Mills Award for S/W Engrg (20 yrs ago)
1 Chief Pgmr – very senior architect/designer/coder
1 Backup Pgmr to assist chief & know all details – almost as senior
+ a few supt members for assistance – Docs Editor, CMS Clerk, Toolsmith, Testsmith, Language guru, Admin

The Problem with Projects – Exposed

** **Capers Jones – 2004 Study**

250 large SW projects from **1998-2004**

o- **10%** = 25 **within plan "successful" (AKA original Estimate: Time, Budget, Features)**

o- 20% = **within 35% overrun** of the plan "barely successful" (AKA **overran** by up to 1/3; Time-Budget)

o-- Projects needed extra time and/or money to finish & deliver

o-- Hard to know if they "worked well" for the users (IE, Validation)

o- **70% = failed, or nearly (AKA > 35% overrun, but got massively more money, or were cancelled)**

--------> So 30% success or "modest" one-third overrun (of cost & timeline/deadline)

o Bit more Subjective, but in line with Capers Jones reports.

Standish Group's Chaos Report

Avg findings: Win=30% Poor=50% Fail=20%

Standish Group's Chaos Report (at a glance); www.projectsmart.co.uk

Measure	1994	1996	1998	2000	2002	2004	2006	2009
Successful	16%	27%	26%	28%	34%	29%	35%	32%
Challenged	53%	33%	46%	49%	51%	53%	46%	44%
Failed	31%	40%	28%	23%	15%	18%	19%	24%

[from www.infoq.com/articles/standish-chaos-2015/]

Measure of success: (Old Success || plus New Extra Success criteria)

on Time, on Budget, on Target || on Goal, User Value and User Satisfaction

UPSHOT (CS): Avg findings: Win=30% Poor=50% Fail=20%

MODERN RESOLUTION FOR ALL PROJECTS

	2011	2012	2013	2014	2015
SUCCESSFUL	29%	27%	31%	28%	29%
CHALLENGED	49%	56%	50%	55%	52%
FAILED	22%	17%	19%	17%	19%

The Modern Resolution (OnTime, OnBudget, with a satisfactory result) of all software projects from FY2011–2015 within the new CHAOS database. Please note that for the rest of this report CHAOS Resolution will refer to the Modern Resolution definition not the Traditional Resolution definition.

PMI == Project Management Institute

Success == met Project Triangle Estimates

onTime, onBudget, onTarget/Features (**nox** "Triple Constraints" == Iron Triangle == Project Triangle)

Best Chance of a Successful Project: o0. **Small** Project Size = Low cplxty
o1. Use Mostly **COTS – "Common/Commercial Off-The-Shelf" S/W**
o2. "Modernize an existing system" – AKA **Port** to new "foundation"

CHAOS RESOLUTION BY PROJECT SIZE

	SUCCESSFUL	CHALLENGED	FAILED
Grand	2%	7%	17%
Large	6%	17%	24%
Medium	9%	26%	31%
Moderate	21%	32%	17%
Small	62%	16%	11%
TOTAL	100%	100%	100%

The resolution of all software projects by size from FY2011–2015 within the new CHAOS database.

Upshot (CS): #1: Grand-row=10%,30%,60%; Small-row=70%,20%,10% #2: Win Ratio S/G=7x

CHAOS RESOLUTION BY AGILE VERSUS WATERFALL

SIZE	METHOD	SUCCESSFUL	CHALLENGED	FAILED
All Size Projects	Agile	39%	52%	9%
	Waterfall	11%	60%	29%
Large Size Projects	Agile	18%	59%	23%
	Waterfall	3%	55%	42%
Medium Size Projects	Agile	27%	62%	11%
	Waterfall	7%	68%	25%
Small Size Projects	Agile	58%	38%	4%
	Waterfall	44%	45%	11%

The resolution of all software projects from FY2011–2015 within the new CHAOS database, segmented by the agile process and waterfall method. The total number of software projects is over 10,000

More from Chaos Rpt 2011-2015

Top 5 Factors of Project Success (AKA Things that seem to "Cause" Failure)

o- 10 Items; First 5 Items are 70% of "estimated responses"

15% Executive Support: [C-Level believes project success is valuable to Biz] when an executive or group of executives agrees to provide both financial and emotional backing. The executive(s) will encourage and assist in the successful completion of the project.

15% Emotional Maturity: [Team vs Group – are they friendly/helpful toward each other] collection of basic behaviors of **how people work together**. In any group, organization, or company it is both the sum of their skills and the "weakest link that determine the level" of emotional maturity.
For mgrs: "managing expectations", consensus building, and collaboration.

15% User Involvement: [Early User Feedback is essential to correct course] users are involved in the project decision-making and information-gathering process. This also includes user feedback, requirements review, basic research, prototyping, and other consensus-building tools **(eg incremental delivery)**.

15% Optimization: [Alignment = Project "is Aligned" with Biz Values/Goals] a structured means of improving business effectiveness and optimizing a collection of many small projects or major requirements. Optimization starts with managing scope based on relative business value.

10% Skilled staff: [competent (non-"strong") staff] understand both the business and the technology. Highly proficient in execution of the project req'ts and delivery of the project. (Better to **invest in people**, even tho it takes time, than to invest **in tools** – Project Mgmt tools, or complicated dev tools required by policy. – EX: force team to use UML tools to build the design.)

Also-rans – 5 More Reasons – Not as important [NOX]

8% SAME (Standard Architectural Mgmt Environment): consistent group of integrated practices, services, and products for developing, implementing, and operating software applications. (helps cuz same tools & procedures)

7% Agile proficiency: (to avoid Agile project failure) the agile team and the product owner are skilled in the agile process. Agile proficiency is the difference between good agile outcomes and bad agile outcomes.

6% Modest execution: (SWE Dev M.O.) having a (**simpler**) process with **few moving parts**, and those parts are automated and streamlined. Modest execution also means using *project management **tools sparingly*** and only a very few features.

5% Project management expertise: application of knowledge, skills, and techniques to project activities in order to meet/exceed stakeholder expectations and produce organization value. (AKA **Manage Expectations**)

4% Clear Business Objectives: understanding of all stakeholders and participants in the business purpose for executing the project. Or the project is aligned with the organization's goals and strategy.

How Does Agile Help?

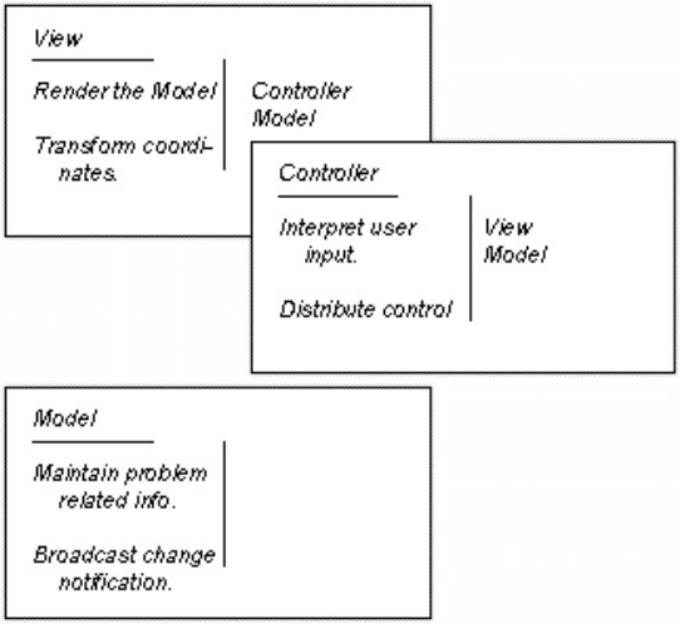
o- **Fail earlier** ==> less "**sunk cost**" (money spent & you can't get it back)

o- **Restart sooner**; cuz Still have most of budget unspent

Read Ch 8 Requirements Modeling p 126 [in 8-th ed. Chs 9 & 10 & 11]

CRC Cards created by Ward Cunningham, 1989 – (Also author of the open-source Wiki software)
(CRC collaboration group also including Kent Beck, Rebecca Wirfs-Brock & Brian Wilkerson)
CF "A Laboratory For Teaching Object-Oriented Thinking", by Beck & Cunningham, OOPSLA'89
CF "Object-Oriented Design: A Responsibility-Driven Approach" by Wirfs-Brock & Wilkerson, OOPSLA'89

EX: MVC top-level architecture in CRC Cards – pg 2 in "A Lab...", Beck & Cunningham, OOPSLA'89



(*) Big Key, tho → Do this with User-level Language to avoid problem/model misunderstandings earliest
(*) Smaller key → Not "Classes", but "Agents" (AKA "Objects") (or PAs == "Personal Assistants")
o- the CRC group was focused on the latest new thing – OOP, OOAD – hence the "Class" in the CRC name

CRC Agent Kinds (very basic)

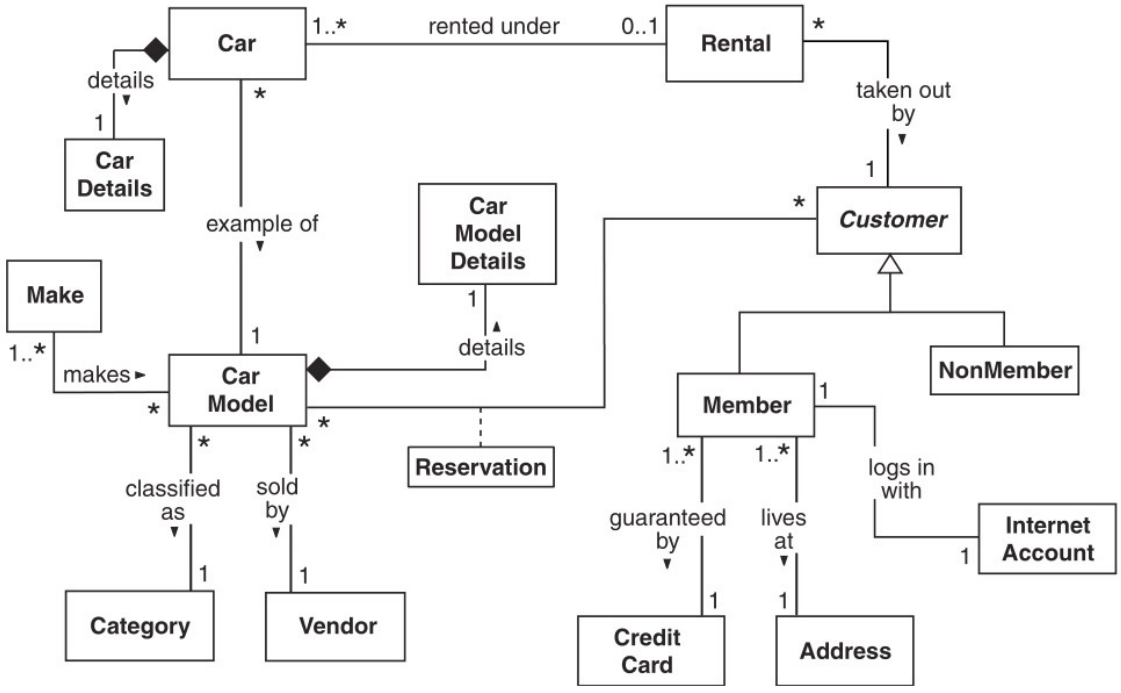
o- **Boundary**/Proxy/Wrapper
o-- Hides complexity of **Outsider** behind a simple API → Lower Coupling to the outside
o--- (*)** Lower Coupling→ Simpler Arch→ Better Understanding→ Higher Productivity & less RT Bugs
o-- If Outsider changes its API (owned by another group/biz), only one agent need be changed)
o-- "Proxy" is a GoF design pattern – " **GoF**" == "Gang of Four", authors of "Design Patterns", 1995
o--- Design Patterns == semi-std multiple-class/object interactions == mini-architecture mechs
o- **Ctlr**/Mgr/Mediator
o-- Hides coordination of helper agents – EG, for a multi-step process
o-- **Helps avoid** helpers knowing each other → Lower Coupling between helper agents
o-- **Coordination** often involves maintaining some **controlling state data**
o-- Mediator is a GoF design pattern; (and "Mgr" used to be a bad name in OOAD)
o- **Entity**/Other
o-- All other kinds of agents dumped here for simplicity of understanding

UML Diags – Principle Kinds

(*) Why useful? Visualization will above the code level
(*) Drawbacks – temptation to cram too much info into a single diagram
o-- Big temptation to force devrs to use UML tools – which are very awkward to control well
Main Kinds – **Inheritance, Association, Sequence/Pole, State/FSM**
o- Others – "Activity"/Flowchart (EX: Fig 8.9), and "Swimlane"/Pole+Flowcharts (Fig 8.10)
o-- Flowcharts usually not helpful (too close to code)
o-- Swimlane diags (AKA Pole+Flowcharts) can be helpful **if not too cluttered**
Also, a **Association/Relationship** diagram

CRC Cards → UML Association/Relationship/Linkage Diagrams (sometimes with Inheritance)

(*)* CRC Cards map directly to simple UML Association diagrams
o-- But most uses of UML Assoc diags are **far too complex**, mixing **many diff levels** of model abstraction
o- It links agents (and sometimes Class to sub-Class, via the hollow triangle – points to parent class)
o- Link == Client-Helper, or Owner, or "Inheritance"
o- Number on link next to box shows multiple of those boxes – but is overly cluttering
o- Filled Diamond on link next to box shows that box owns a bunch of instances of the other box's kind
o-- Often, this means eg, a Car has 4 wheels – but is usually "class overkill" → treat them all as agents
o- Label on a link usually means some sort of relationship – but obscures agents
o- Big pbm with links is in **exposing** an agent's **private data** (eg, Car → Car Details)
o- Hollow triangle on a link – points to parent class, from sub-class → a bad choice when agents are involved
(*)* The principle problem with most UML Association diagrams – **mashup of multiple concept levels**
EX: Fig 7.1 from "Object-Oriented Analysis and Design", by O'Docherty, 2005 – a normal "simple" example



UML State/FSM Diag – (FSM == Finite State Machine)

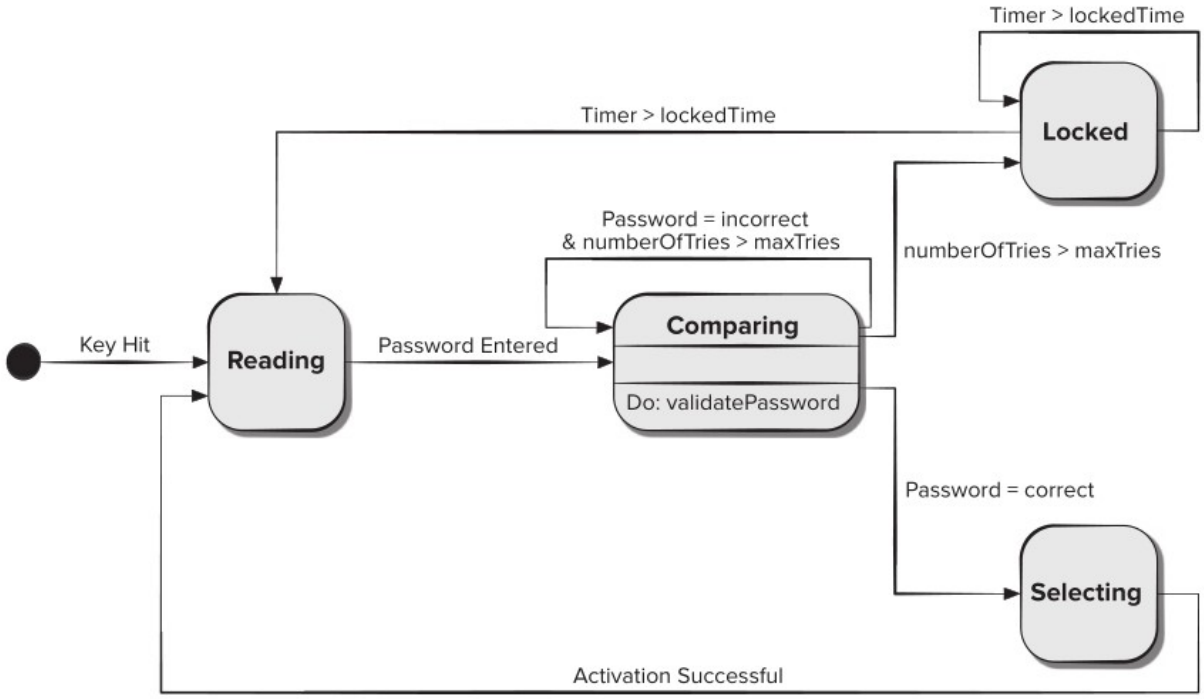
** Can be very useful

o- State nodes – and Transition links = event (optionally plus an "action" to run)

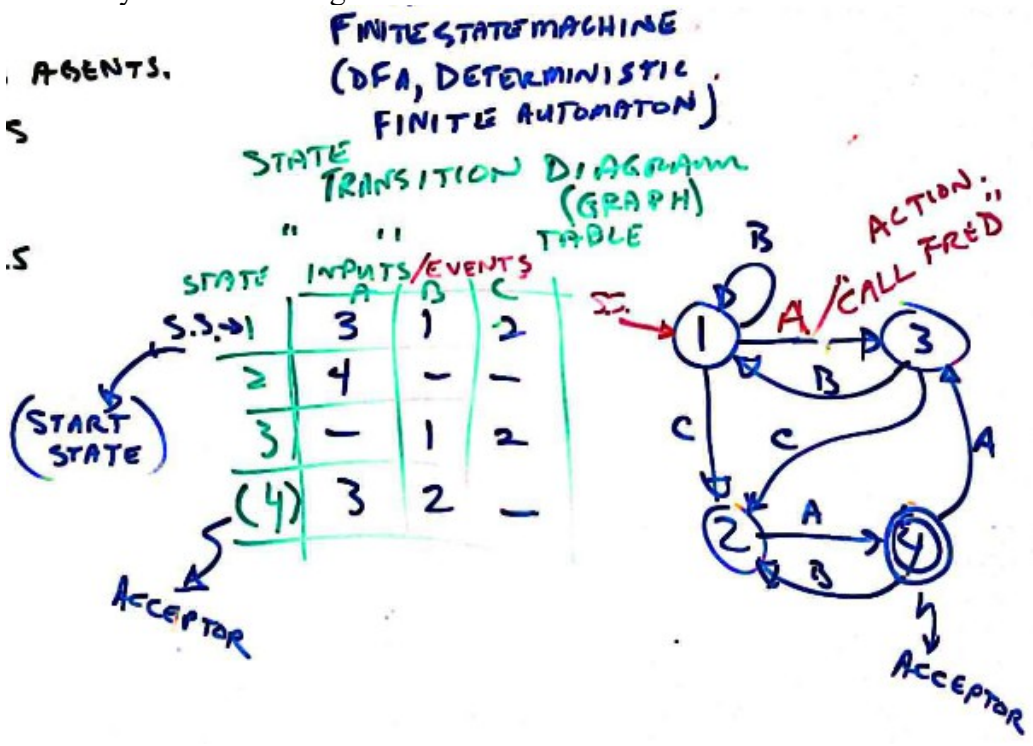
o- Usually has an initial or "start" state (called "SS") or event

o- Usually has a final "acceptance" or "acceptor" state – that is resettable to the start state

EX: Fig 8.8



EX: The usual FSM style – a State Diagram on RHS and its State Transition Table on LHS



UML Sequence/Pole Diag – See Lect 200204

Read o11.2.1 Basic Design Principles p212 [in 8-th ed. 14.2.1]

o11.2.1 Basic Design Principles p212

Component == collection of agents/classes, treated as a part – hopefully moderately related to each other
- o-- Closely-related because we want High Cohesion for any part/box/object/agent

(*) Best wrapped in a **Ctlr**/Mgr/Mediator agent – so you only see a simple API for a "black box"

SOLID/D (OOP) – Class-Based stuff
- o- (SRP) **Single Responsibility** Principle -- Does one thing → 1 verb to 1 or more objects
- o- (OCP) **Open/Closed** P -- Can inherit from it but can't change it's guts
  - o-- Cuz if you change a class's guts – can affect dozens of Client classes using the class → subtle RT bugs
  - o-- One of the issues in class-based OOP
  - o-- (Not too bad during initial development – but often deadly after first version is shipped)
  - o-- So, **you sub-class the original class** and then override its methods as needed (and maybe add data slots)
- o- (LSP) **Liskov Subst** P -- "if Kid wears Mom-hat, then it must behave like Mom"
  - o-- If you override Mom's method, but don't ensure Mom's old behavior also runs→ **very subtle** RT bugs
    - o--- RT bugs show up in old well-tested code
  - o-- Because dozens of older classes rely on any Mom-like object to behave like Mom
    and your new sub-class object can be passed as a Mom argument (AKA kid is "**up-cast**" to a mom)
    and then some other object can call Mom's "method" expecting to get Mom-behavior
    and if your override of Mom's method doesn't provide this behavior then the other object can crash
    → very obscure kind of bug indeed
  - o-- One of the issues in class-based OOP, overriding methods, & polymorphism
  - o-- Plus – **Barbara Liskov, 43-rd Turing award, 2008**
    Foundations of programming languages, system design, data abstraction, fault tolerance, and distributed computing.
- o- (ISP) **Interface Segregation** P -- Wrapper to hide Outsider internals, or (originally) unused API items
- o- D1 == (DIP) **Dependency Inversion** P – Client knows Helper's API, & has a handle/link to actual Helper
  - o-- Client does NOT KNOW the Helper's guts (except possibly to Construct the Helper) – a Loophole
  - o-- If Client knows Helper directly – means Client "includes" Helper's Class Defn (showing Helper's guts)
  - ** But there is an extra pbm – How does Helper agent get created → Creator needs to include Helper's Class
- o- D2 == **Dependency Injection** P – some other agent (eg, Mgr/Mediator) installs Helper link into Client
  - o-- So that Client only needs to know Helper's API
  - o-- Means the Client has an "install helper reference/pointer" method
  - o-- There are GoF design patterns to do this for several mini-solutions

Pressman & Maxim: 8th ed vs 9th ed – line by line correspondence of section headings

9. Reqts Modeling: Scenario-Based Methods106 9. 1 Requirements Analysis 167 9. 1.1 Overall Objectives and Philosophy 168 9. 1.2 Analysis Rules of Thumb 169 9. 1.4 Requirements Modeling Approaches 171 9.2 Scenario-Based Modeling 173 - - 9.2.1 Creating a Preliminary Use Case 173 9.2.3 Writing a Formal Use Case 177 10. Reqts Modeling: Class-Based Methods 184 10.1 Identifying Analysis Classes 185 10.2 Specifying Attributes 10.3 Defining Ops 9.3 UML Models That Supplement the Use Case 10.4 Class-Responsibility-Collaborator Modeling - - - 11.1 Creating a Behavioral Model 203 11.2 Identifying Events with the Use Case 203 11.3 State Representations 204 9.3.1 Developing an Activity Diagram 180	Ch 8 Reqts Modeling — A Recommended Appr 126 8.1 Requirements Analysis 127 8.1.1 Overall Objectives and Philosophy 128 8.1.2 Analysis Rules of Thumb 128 8.1.3 Requirements Modeling Principles 129 8.2 Scenario-Based Modeling 130 8.2.1 Actors and User Profiles 131 8.2.2 Creating Use Cases 131 8.2.3 Documenting Use Cases 135 8.3 Class-Based Modeling 137 8.3.1 Identifying Analysis Classes 137 8.3.2 Defining Attributes and Operations 140 8.3.3 UML Class Models 141 8.3.4 Class-Responsibility-Collaborator Modeling 8.4 Functional Modeling 146 8.4.1 A Procedural View 146 8.4.2 UML Sequence Diagrams 148 8.5 Behavioral Modeling 149 8.5.1 Identifying Events with the Use Case 149 8.5.2 UML State Diagrams 150 8.5.3 UML Activity Diagrams 151