Popular-Pgmg-Langs-Index_TIOBE

1	Java	16.380%	+1.74%
2	C	14.000%	+7.00%
3	C++	7.668%	+2.92%
4	Python	5.192%	+1.64%
5	C#	4.402%	+0.95%
6	VB .NET	4.124%	+0.73%
7	PHP	3.321%	+0.63%
8	JavaScript	2.923%	-0.15%
9	SQL	1.987%	+1.99%
10	Ruby	1.182%	-1.25%
11	R	1.180%	-1.01%
12	Delphi /Obj-Pascal	1.012%	-1.03%
13	Assembly	0.998%	-1.86%
14	Go	0.970%	-1.11%
15	Objective-C	0.939%	-1.16%
16	MATLAB	0.929%	-1.13%
17	VB	0.915%	-1.43%
18	Perl	0.909%	-1.69%
19	Swift	0.907%	-1.37%
20	Scala	0.900%	+0.18%
21	Apex	0.899%	
22	PL/SQL	0.899%	
23	Transact-SQL	0.884%	
24	Ada	0.867%	
25	SAS	0.859%	
26	Dart	0.859%	
27	Lisp	0.854%	
28	COBOL	0.853%	
29	LabVIEW	0.696%	
30	Bash	0.689%	
31	D	0.679%	
32	Logo	0.577%	
33	Scratch	0.536%	
34	Prolog	0.517%	
35	Erlang	0.506%	
36	Awk	0.503%	
37	Clojure	0.484%	
38	Alice	0.473%	
39	Lua	0.424%	
40	Fortran	0.418%	
41	OpenCL	0.415%	
42	ABAP	0.409%	
43	REXX	0.382%	
44	Scheme	0.381%	
45	ML	0.377%	
46	Julia	0.342%	
47	ActionScript	0.321%	
48	Haskell	0.320%	
49	Kotlin	0.292%	
50	RPG	0.281%	

Cumulative Percentages

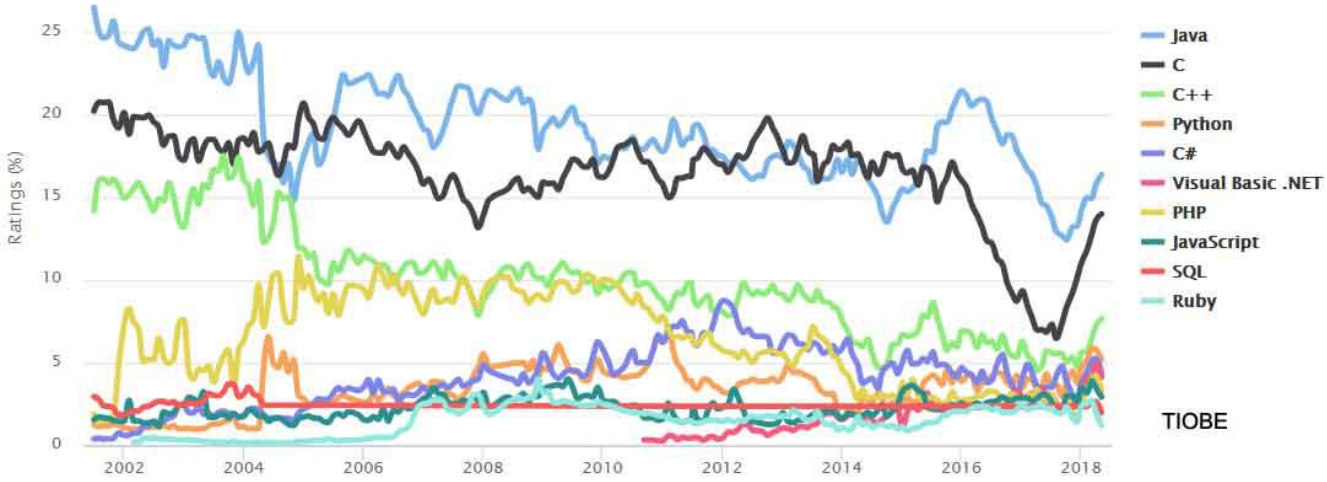
1 Java	**16.380%**	
2 C	**14.000%**	**30.4%**
3 C++	**7.668%**	**38.0%**
4 Python	5.192%	43.2%
5 C#	4.402%	47.6%
6 VB .NET	4.124%	**51.8%**
7 PHP	3.321%	55.1%
8 JavaScript	2.923%	58.0%
9 SQL	1.987%	54.0%
10 Ruby	1.182%	**61.2%**

Lang	**2018**	**2013**	**2008**
Java	1	2	1
C	2	1	2
C++	3	4	3
Python	4	7	6

Lang	**2018**	**2013**	**2008**
C#	5	5	7
VBNET	6	13	-
JavaScript	7	10	8
PHP	8	6	4

Lang	**2018**	**2013**	**2008**
Ruby	9	9	9
Delphi	10	12	10
Perl	11	8	5
Obj-C	18	3	43

Lang	**2018**	**2013**	**2008**
Ada	28	16	17
Lisp	31	11	15
Pascal	145	14	18



The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. It is important to note that the TIOBE index is not about the *best* programming language or the language in which *most lines of code* have been written.

Lab – Creating the Command Result Web Page – Splicing Text Strings to Create a Web Page

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:

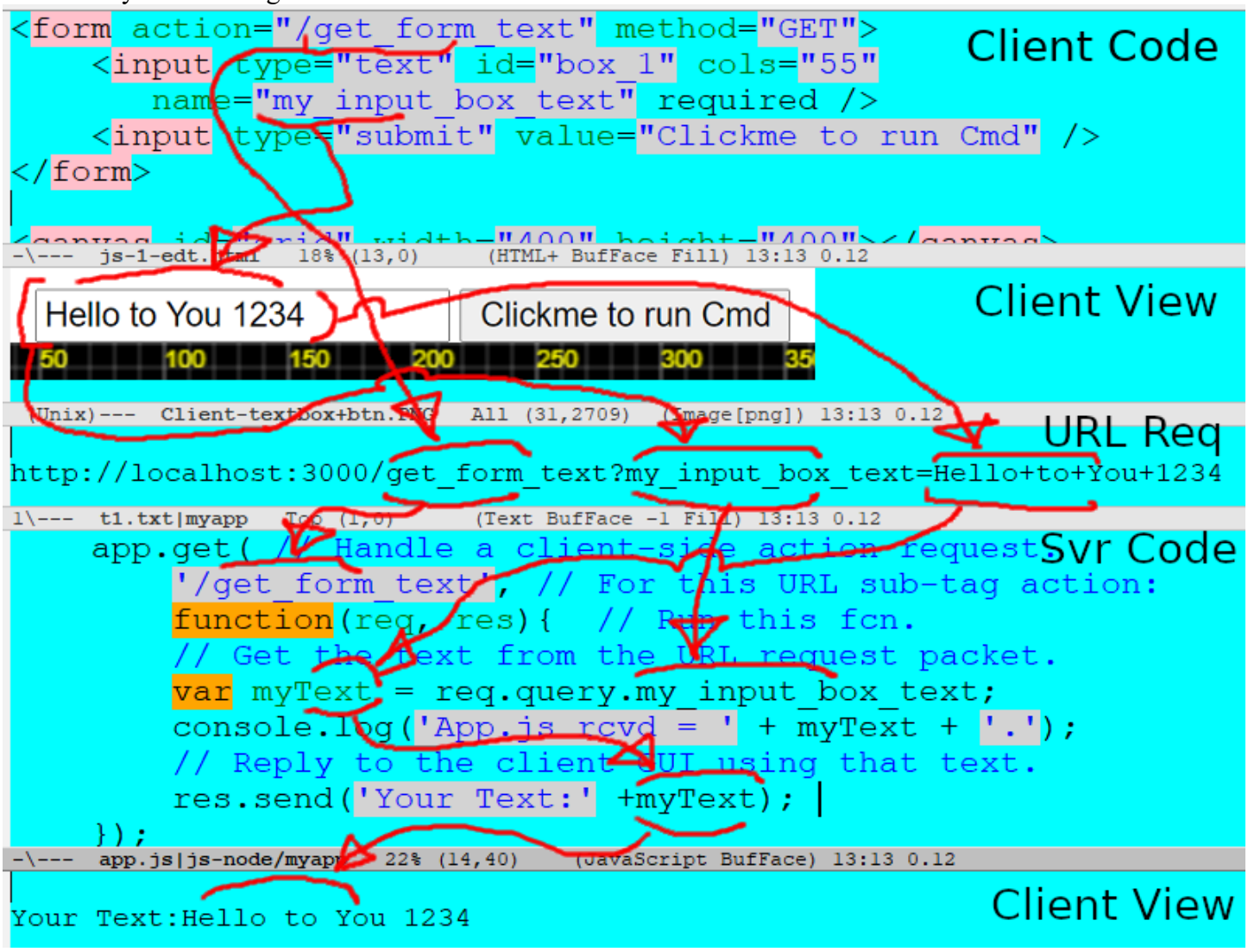
```
'<form action="/create_repo" method="GET">
  <label> Source Path:
      <input type="text" id="box_2" cols="55"
             value="(Replace me) C:\\Time\\for\\Squiddie\"
       name="source_path" required /> </label>
    <label> Target Path: <input type="text" id="box_3" cols="55"
       name="target_path"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

Lab – Creating the Command Result Web Page – Getting Command Data & Composing the Response

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.



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)

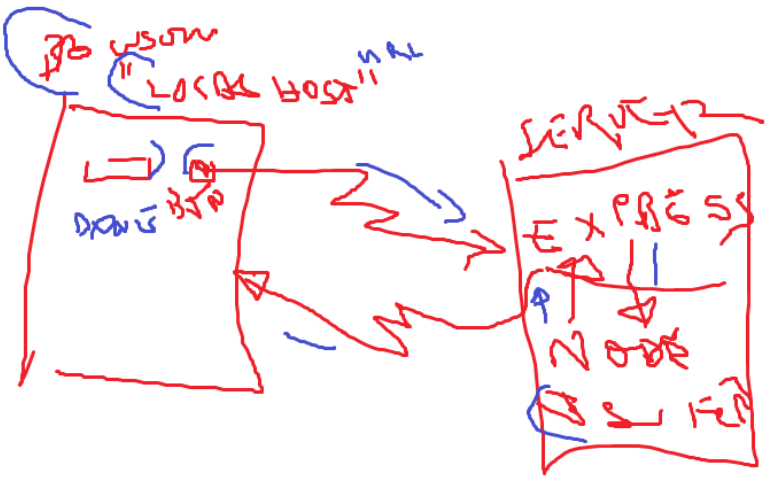
S/W Nature Qs:

- **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

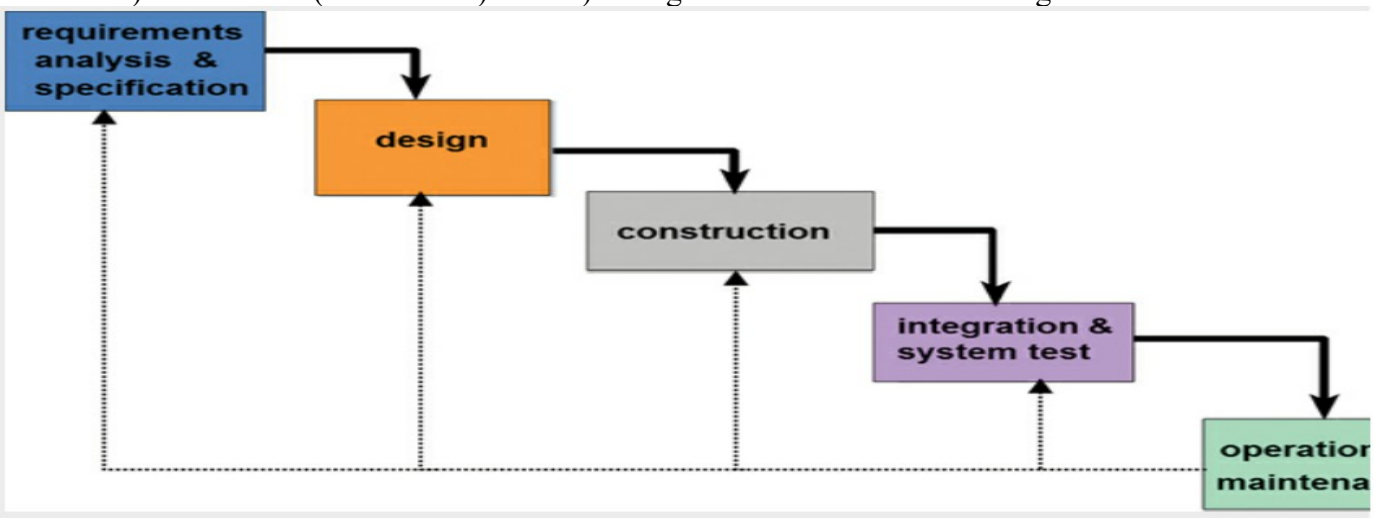


Figure 2.1 Waterfall model.

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)

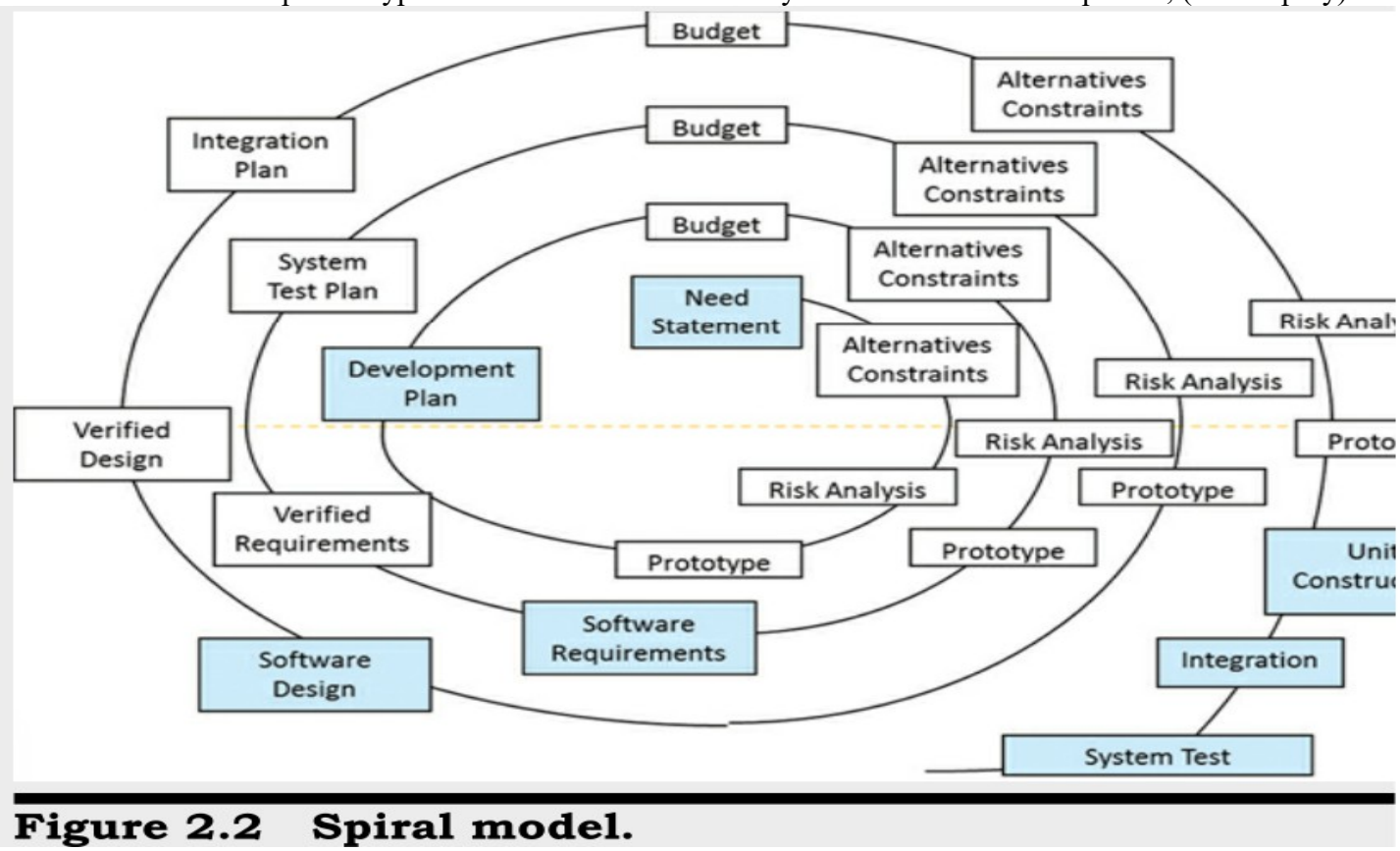


Figure 2.2 Spiral model.

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

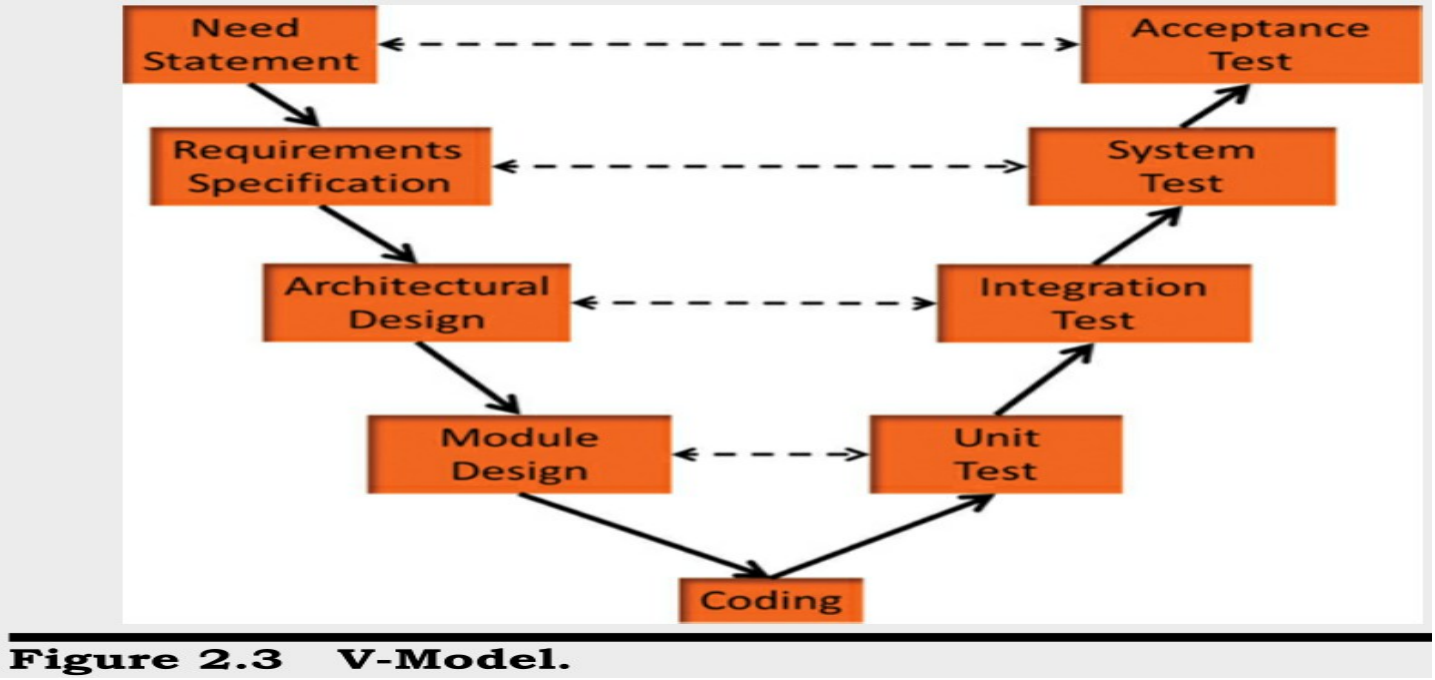


Figure 2.3 V-Model.

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**

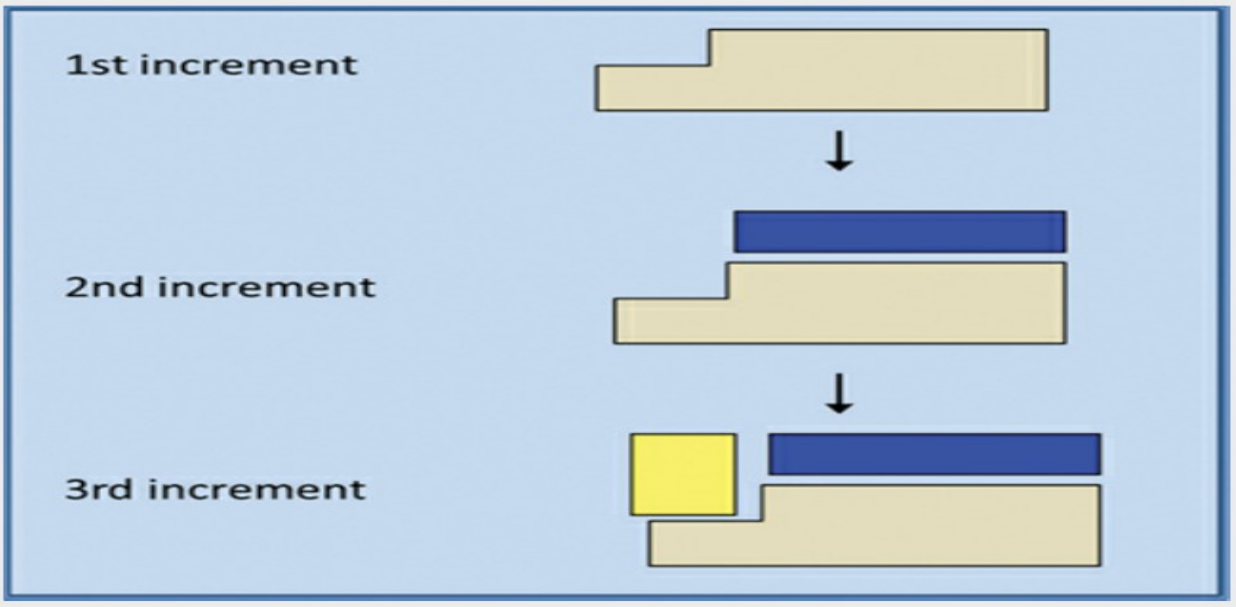


Figure 2.4 Incremental model.

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

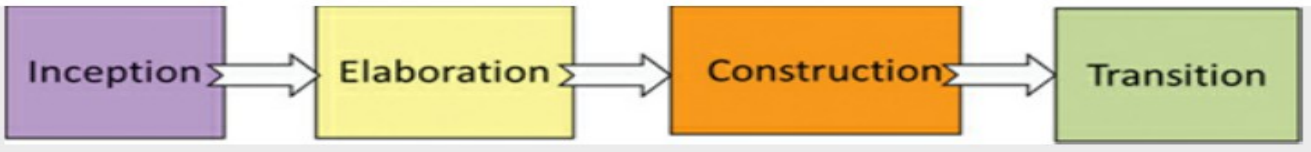


Figure 2.6 UP process.

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

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

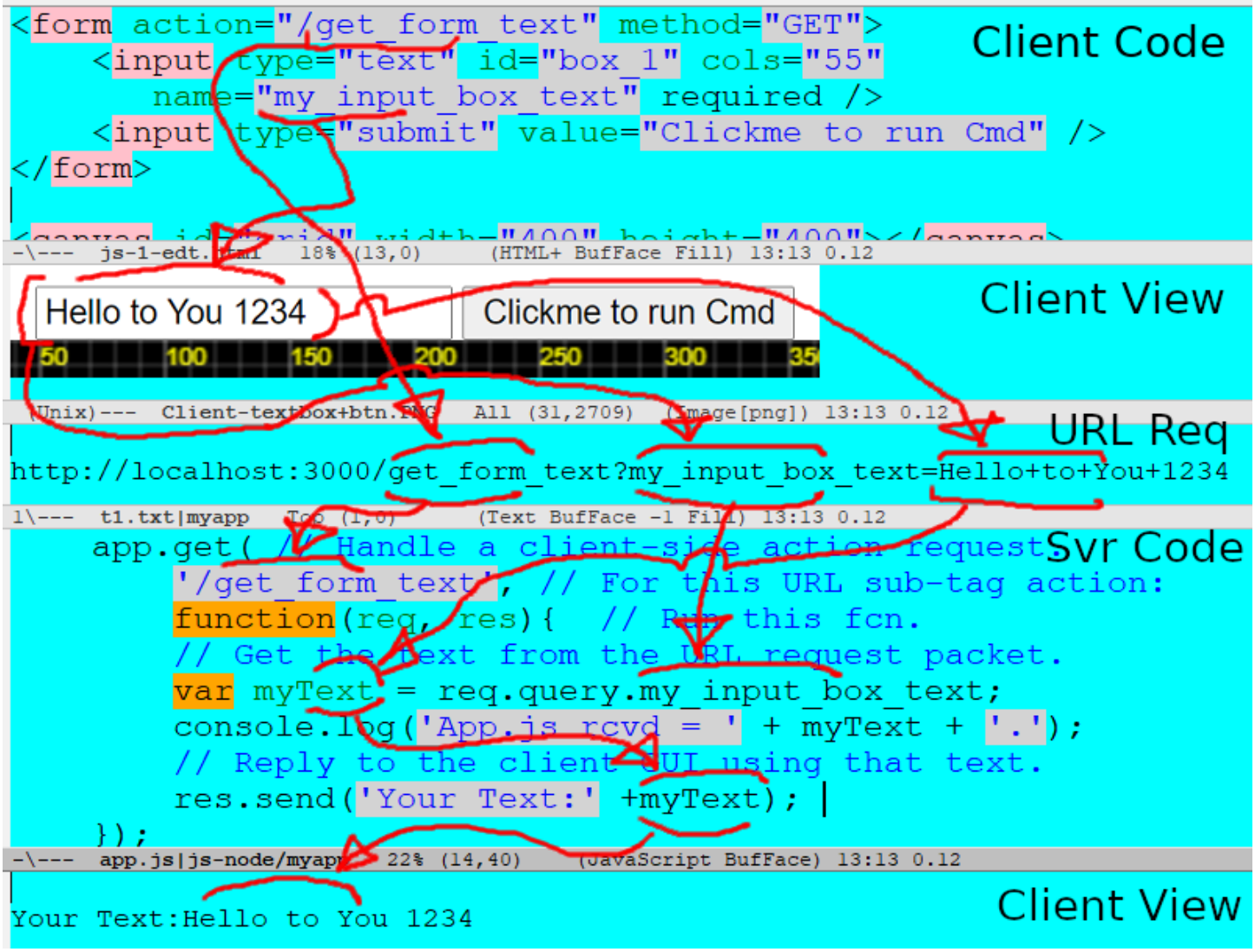
Lab:
Git VCS Terms
"repo" -- git has one per user – P1 has a single centralized repo (simpler)
  o- git: it's in .git sub-folder of your projtree folder -- no central repo – P1, it's at a fixed folder loc
  o- "git init' -- create an empty repo in curr folder – P1, "create repo" creates the repo from 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.

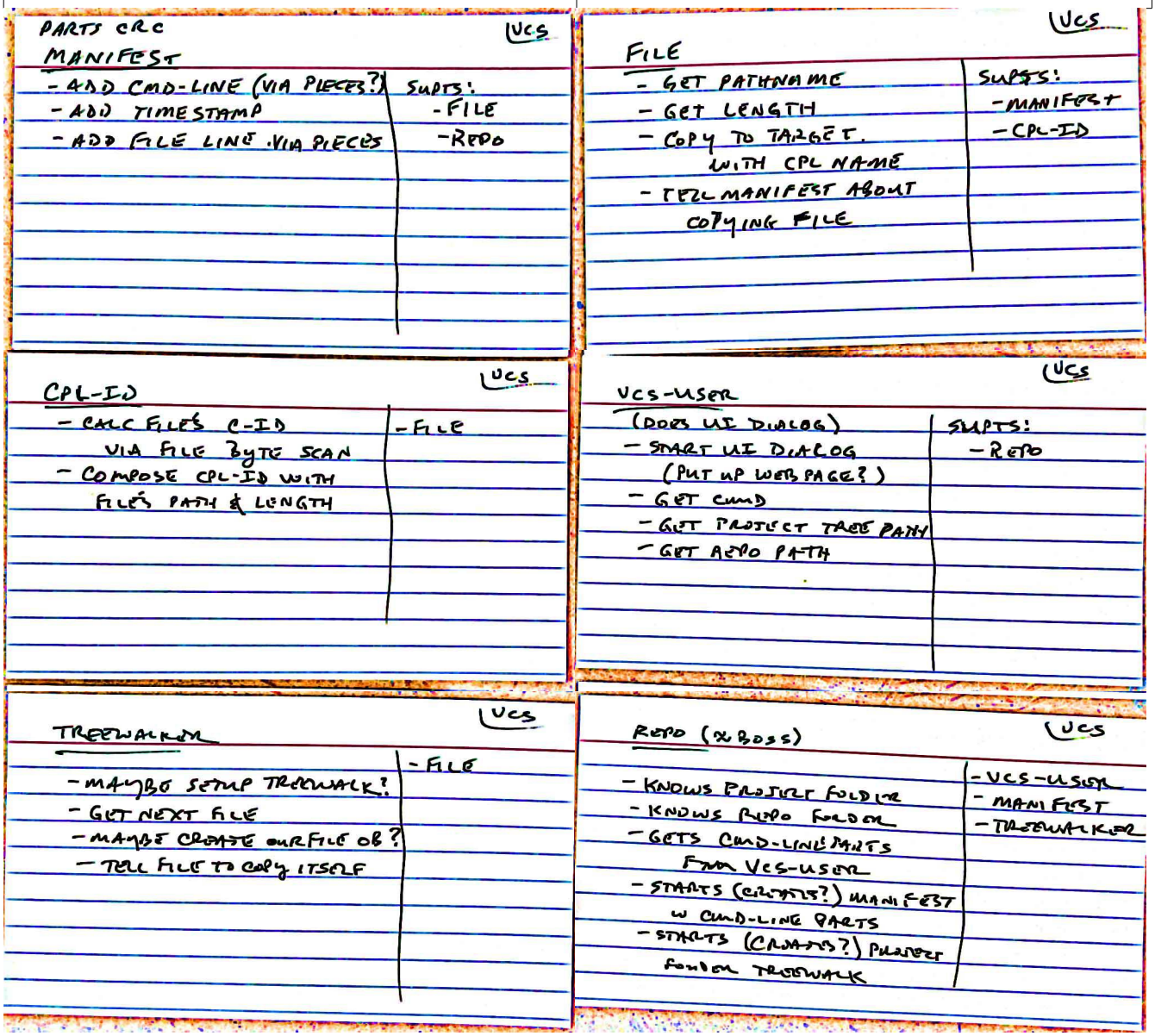
Picture: key-name linkages in the Client-to-Server-and-back-to-Client flow.



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 – 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>
```

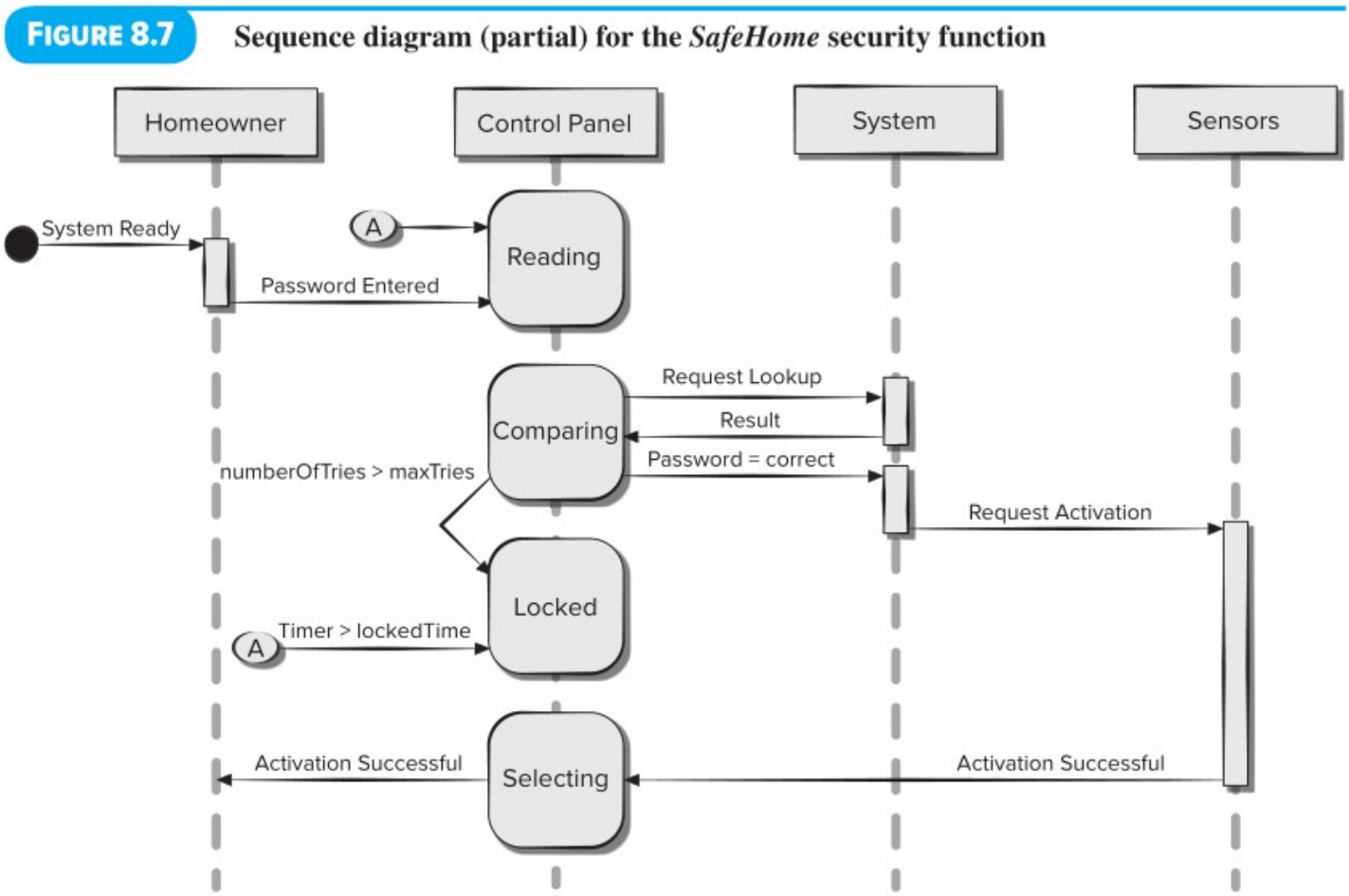
Another Form for the **next** user **query**/command – maybe like this
- o- Same as the first form – but either escape the internal quotes, or use single-quotes outside

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:

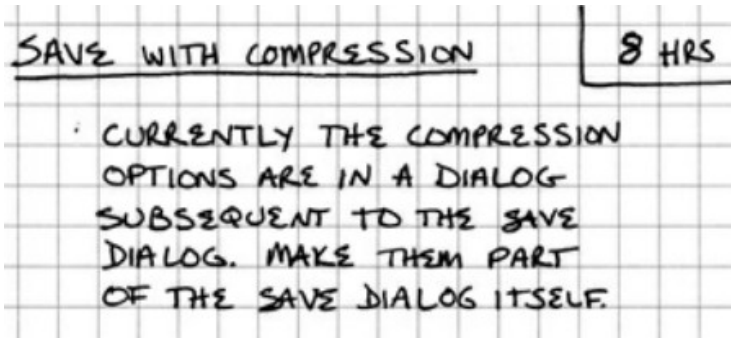
```
let fool = require( './foo.js' )  // eval's foo.js creating obj w slots
// Access each exported Foo var/fcn with its name via the var 'fool'
console.log( fool.myvar );
fool.myfn( );
```

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?

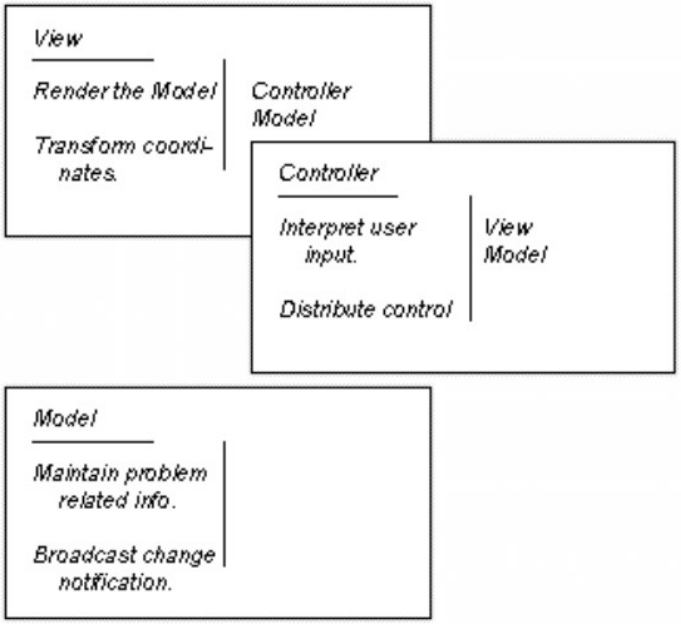
o- **Fail earlier** ==> less "**sunk cost**" (money spent & you can't get it back)

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)

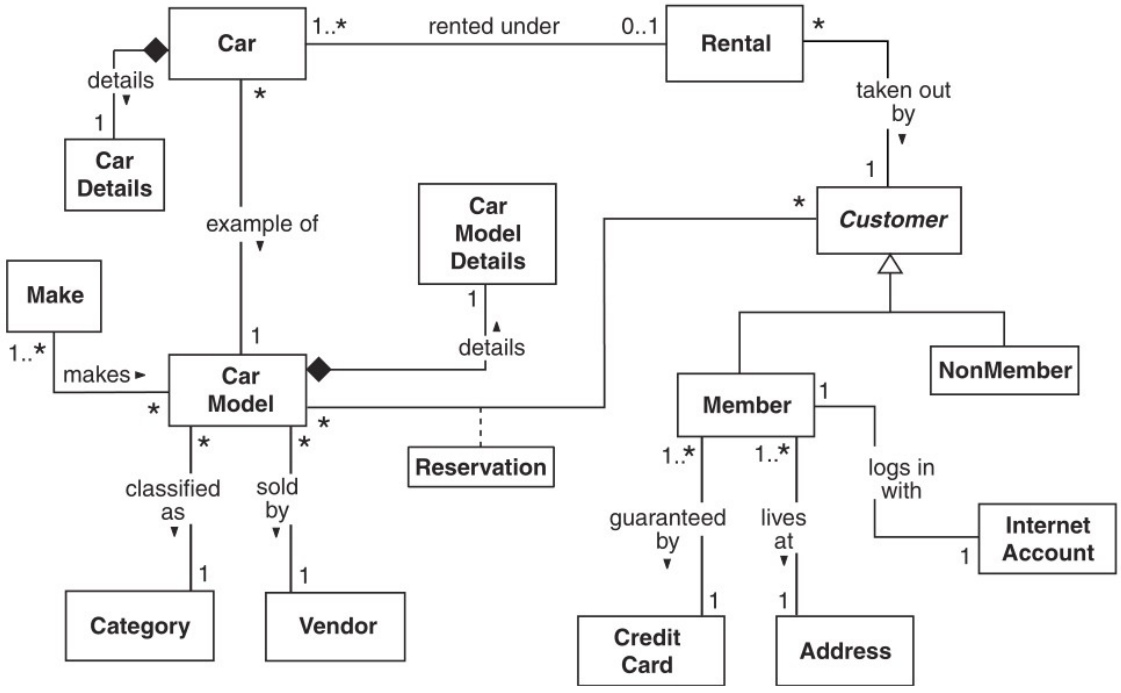
o- **Boundary**/Proxy/Wrapper
o-- Hides complexity of **Outsider** behind a simple API → Lower Coupling to the outside
o--- (*)** Lower Coupling→ Simpler Arch→ Better Understanding→ Higher Productivity & less RT Bugs
o-- If Outsider changes its API (owned by another group/biz), only one agent need be 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)

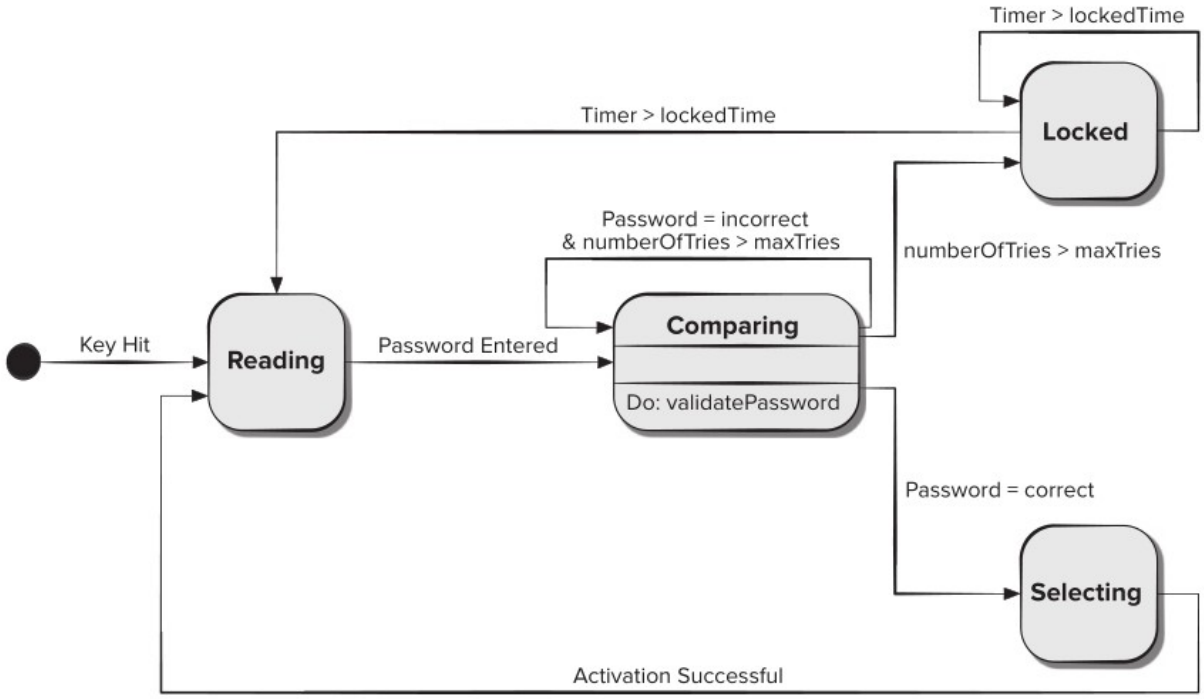
** Can be very useful

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

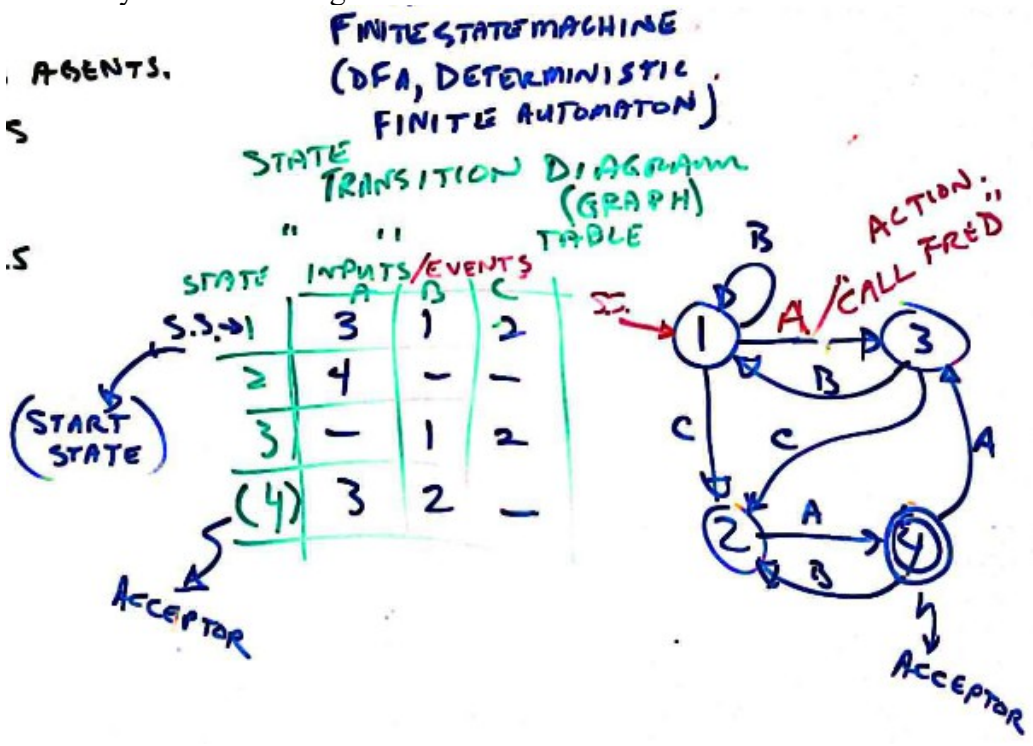
o- Usually has an initial or "start" state (called "SS") or event

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

Read "No Silver Bullet" Fred Brooks
"No Silver Bullet: Essence and Accidents of Software Engineering", by Frederick P. Brooks, Jr.,
TR86-020 Sept 1986, UNC Chapel Hill
CF https://www.cs.unc.edu/techreports/86-020.pdf
NB, it is also included in Brooks' book:
The Mythical Man-Month: Essays on Software Engineering, Brooks, 2ed 1995

MVP – Minimum Viable Product

o- CF eg, The Lean Startup, 2011, by Eric Reis
Goal: Create skinniest (minimum) product that **customers will pay** for (**viable**) ASAP
== Product can "live on its own" == will start making money as soon as its delivered
(*) A Startup biz typically does NOT have a customer paying for development
o-- It has "venture" or "angel" or friends' & relatives' money – and never enough of it
o-- Point is to supplement spending venture capital "seed" money ASAP – before you run out (of business)
Q: Have potential users said they'd be willing to pay for something like it?
Q: No frills (almost), but still usable for real tasks?
Q: **[Big Q] How can it be scaled down? – Ask yourself Repeatedly**
Goal: Get off of the wrong product track ASAP
Pbm -- It's **easy to delude yourself** that the project will sell
Soln – Trust potential users to tell you if they like it – enough to pay money for it
Q: **[Big Q] Are we on the right track? – have you avoided misunderstandings?**
o- Show (or let users drive) something usable to the user, to get early feed
Pros: **Avoid spending time building**
o- the wrong product (you misunderstood them) → **avoid wasting your time+money**
o- stuff (**features/frills**) they don't care about → **avoid wasting your time+money**
o- stuff not needed in the 1st MVP → **avoid wasting your time+money**
o- stuff needlessly optimized → **avoid wasting your time+money**
Fast Keys: Can you show them → and get **very early feedback** on misunderstandings
o- a **paper-pencil version** of the **UI** & **(major) actions**? (above CRC-level)
o- **Mock-ups** (eg, screen shots) of key UI & actions?
o-- Mock-up == not the real thing, but the facade (ie, outside look) looks real-ish
EX: a Mock object has the real-ish API but returns a pre-determined result, not calc'd
o- **Key UI pieces with** some mock-up (hard-coded) **actions** (EIO)
o-- Hard-coded == get answer by look-up, not by computation
EX: for input (key) args ("fred", 42) answer is "Chicago" → via a lookup table or switch stmt

MFS – Minimum Feature Set (Often Smaller than MVP)

Goal: Get something usable to show to the user, to get early feedback
o- Usually not enough features to be viable
o-- Often enough features to perform one useful demo-able action
MFS rough demo:
o- 1st-cut app; **sweetest** (eg, Priority combined with Effort) features
o-- Main-Scen only (of a UC, has no frills, no/little error handling)
o-- Blackbox demo only // users-no-touch
o-- Emphasis on time-to-demo

o9.3.2 Architecture p163

- o- How parts are connected -- both statically and dynamically (AKA at runtime)
  - o-- Static == **Descriptive** == No Timing issues;
  - o-- Dynamic == Procedural == correct Sequencing in Time

Many std Arch patterns – (typically not GoF Design Patterns, which are lower level)

- o- They can be **mixed** o- as siblings or o- as mom-n-kids or o- merged

Sample Arch patterns – Help with Speed, but also Help with Simplicity (→ better understanding, fewer bugs)

- o- **Hierarchical** (Master-Slave) (Monolithic) (Fcn-Md Call-based)
- o- **RPC** (Remote Procedure Calls)
  - o-- Fcn-Md Call-based but fcns/mds/objs reside on diff (remote, usually) PUs
  - o-- "Blocking" == you wait for the answer
- o- **Layered** – upper layer can call lower layer fcn for help (& never call upward) – EX: TCP protocol
- o- **MVC** (Model-View-Controller)
  - o-- Model does pbm calculations & state; Views display model info; Controller adjusts views or model
- o- **Client-Server** (== 2-Tier, 3-Tier, N-Tier) – Horizontal cuz diff (remote, usually) PUs
  - o-- **3-Tier** : (GUI)<==> (Biz-Logic) <==> (RDB)
  - o-- 3-Tier [optional] : (GUI)<==> (Biz-Logic) <==> (ORM) <==> (RDB)
  - o-- ORM = Object-to-Relational Mapper
- o- **Dataflow** (Pipes-n-Filters) (Assembly lines) – linear or DAG or even digraph (directed graph)
- o- **Multi-Agent** (OO)
  - o-- OO Class-Based – class hierarchy fixed at compile-time
  - o-- OO non-Class – dynamic inheritance linkage (eg, JS, Lisp)
- o- **Event-driven** (Reactive, Event-Loop, Event-Bus)
- o- **Broker** (Router) (ORB == Object Request Broker) – middleman connecting Requesters to Services/Helpers
  - o-- Requesters & Services don't know each other – Low-Coupling
- o- **SOA** (Service-Oriented Arch) – Msg-bus + Broker/Orchestrator + Msg-Translator + Clients + Helpers
- o- **Buffered** – put a data-storage "buffer" between two agents for communication
- o- **Parallel** – all PUs do the same instruction at the same time, each on their own variant of the data
- o- **Distributed** – all PUs do their own programs with the data they were given
- o- **Map-Reduce** – split/shard whole pbm data → hand each shard to a PU → PUs calc → send answers back → aggregate together the shard answers (Reduce) into the final answer
- o- **Msg-Based** – Distributed objects communicate via messages (no call-waiting) – typically with buffering
- o- **Pub-Sub** (Publish-Subscribe) (like GoF Observer Pattern)
  - o-- A central (key) Publisher gets Subscribers for the Pubs occasional info notices
  - o-- Reverse is "Polling"
- o- **Polling** – Interested parties ask the Key agent "Is there any news, yet?"
- o- **DB-Centered** == Data-Centered (eg, RDB, Blackboard)
- o- **Blackboard** – Centralized Pbm State (in parts) + Update Event-bus + Pbm-Part-Solvers that do updating
  - o-- Pbm-Part-Solvers look at new data & incrementally update parts of the State toward a full solution
- o- **RBS** (Rule-Based System) – Centralized Pbm State + If-Then-Act Rules + Run-Rule-Controller
  - o-- Controller compare Rule IF-condition to State to find Ready-Rules, then picks one to Run & update State
- o- **VM** = (Virtual Machine) (Interpreter) – build a VM closer to the Pbm Domain level & program the solution in this new higher-level VM "instruction" "language"
- o- **P2P** (Peer-to-Peer) – each PU has a cloned Controller & knows neighbors & they communicate to do stuff
  - o-- Distributed, Decentralized
- o- **FSM** = Finite State Machine
- o- **Process Control** – sensors + balance-controller + actuators --- used to keep processing within safe bounds
  - o-- eg, PID controller (PID = Proportional, Integral, Differential)

- o- **Entity-Component-Systems** – Split Object data up so each slice is handled by a Master
  - o-- Masters (Systems) control diff behavior areas – Each Master has "methods" to handle the area
  - o-- Component is a set of all of the same kind of slice = Component has a bunch of slices
  - o-- Each Master used 1 or a few of the Components to adjust some state for each of the Objects (via its slice)

  EX: MMORPGs – typically handles 1 to 50 Gbytes of new data daily, massive # of objects
- o- **Swarm** – Bunch of simple agents + know dynamically-local neighbors + do stuff w respect to neighbors
  - o-- AI technique; also used for massive # of drones coordination
- o- **Replanner** – dynamically update & fix goal-directed plans in the face of environment changes
- o- **REST** (Memory-less, Stateless) – The needed state from earlier results is included in the messages, so that each agent doesn't need to keep track of what happened in the past – Low-Coupling
  - o-- REST = Representation State Transfer – a style of Web Cli-Svr interaction
- o- **Shared-Nothing** (PU/Agent-clones) – they each get shards (or diff pbms), send back answers
- o- **SBA** (Space-Based PU/Agent-clones + Svc-levels) – Shared-Nothing but must do Svc-Level Agreement
- o- **Microservices** – Tiny part-solver agents + input event/msg bus + output event/msg bus
  - o-- Output bus is connected to the input bus
  - o-- Easy to start/stop/add new agents → Flexibility
  - o-- Can have multiple copies of an agent → Resilience

Buffered (AKA Reader-Writer) Arch Pattern – Flexible Low-Coupling S/W Models Lect #13

o- Separate two Agents/Modules with a Writer-Reader Data Buffer (AKA a "Model Point")

o-- S/W "**Buffer**" == 1) a place to store stuff, and 2) separating direct contact between two agents

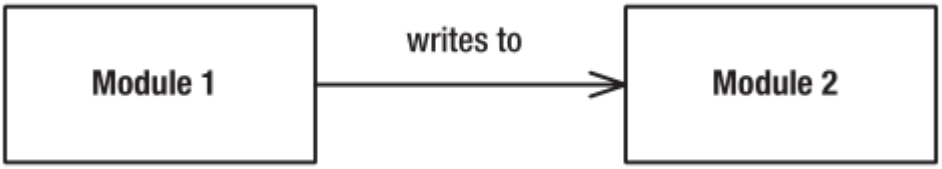
o- Each Agent gets a ptr/ref to the Buffer – use Dep Injection (D2 of SOLID/D)

(*)* Allows you to change either agent &/or its data format API via a Translator module (see 2-nd diagram)

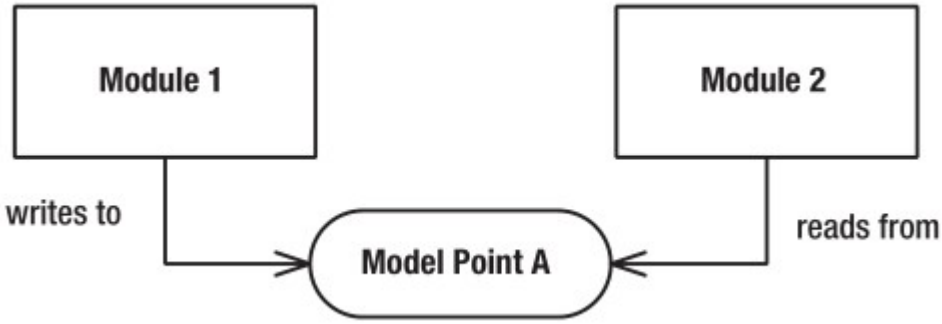
Data "Dropbox" – Read/Write to a Data Pile

(CF Patterns in the Machine, by John & Wayne Taylor, 2021)

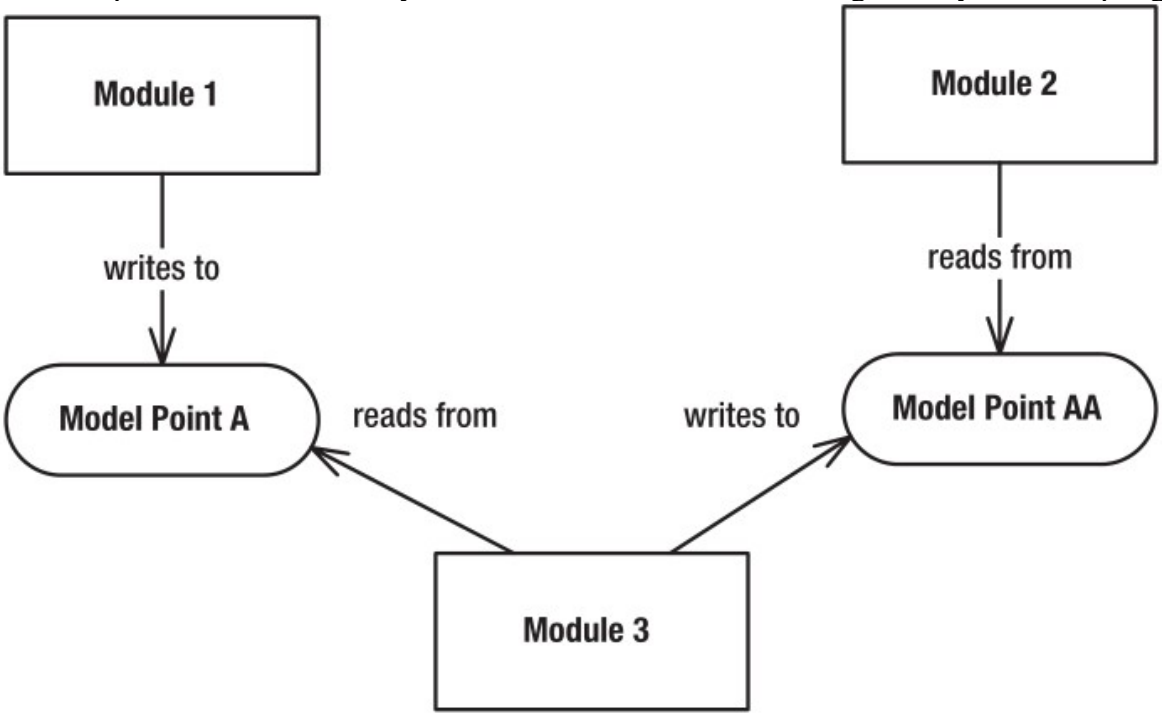
Direct Linkage – Moderate Coupling



Data Dropbox – Indirect Linkage – Lower Coupling



Data Dropbox with Intermediary – Modules #1 & #2 Do Not Change – Very Low Coupling



• facilitates parallel development, which is what allows you to add additional programmers to a project and have them actually contribute effectively toward shortening your development schedule

• It simplifies the construction of unit tests

• It simplifies writing platform-independent and compiler-independent source code

Example modification use – Before and After two Translators were added

Fig 9.2 – from Taylor & Taylor – a PID controller watching temperature

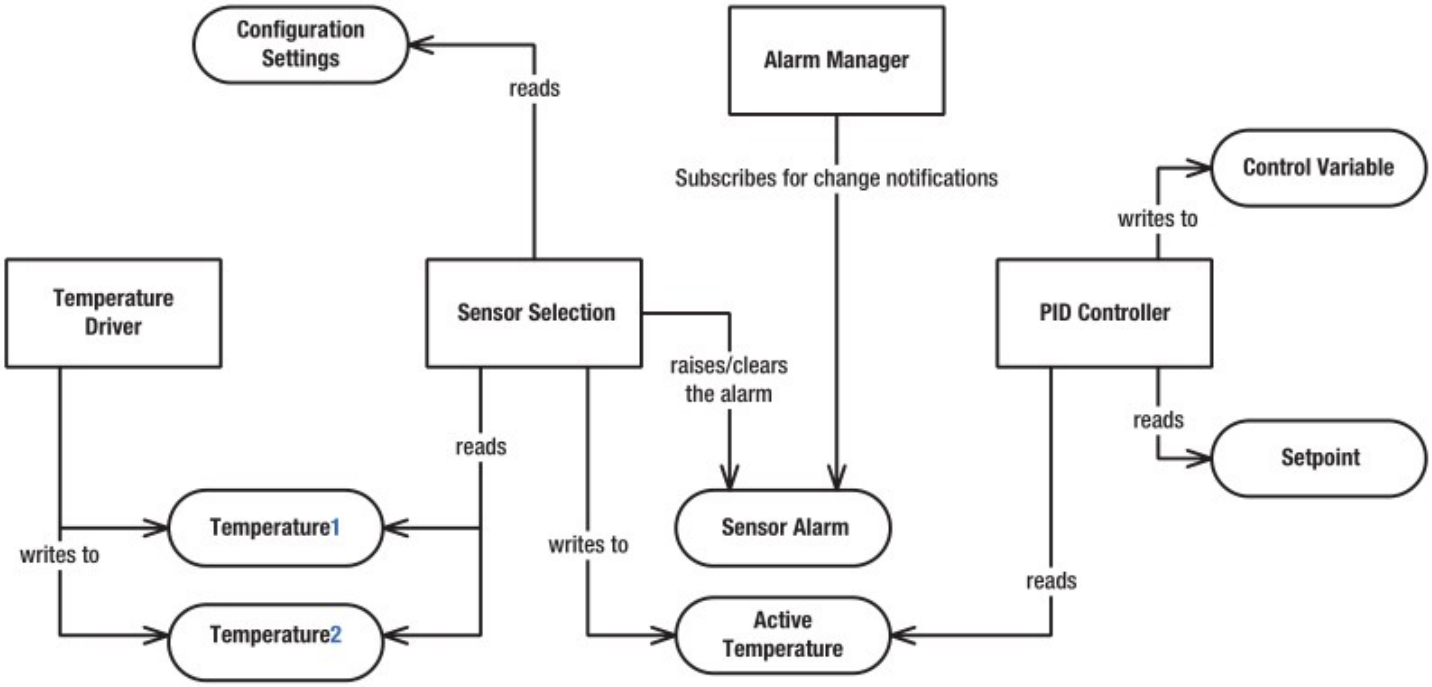
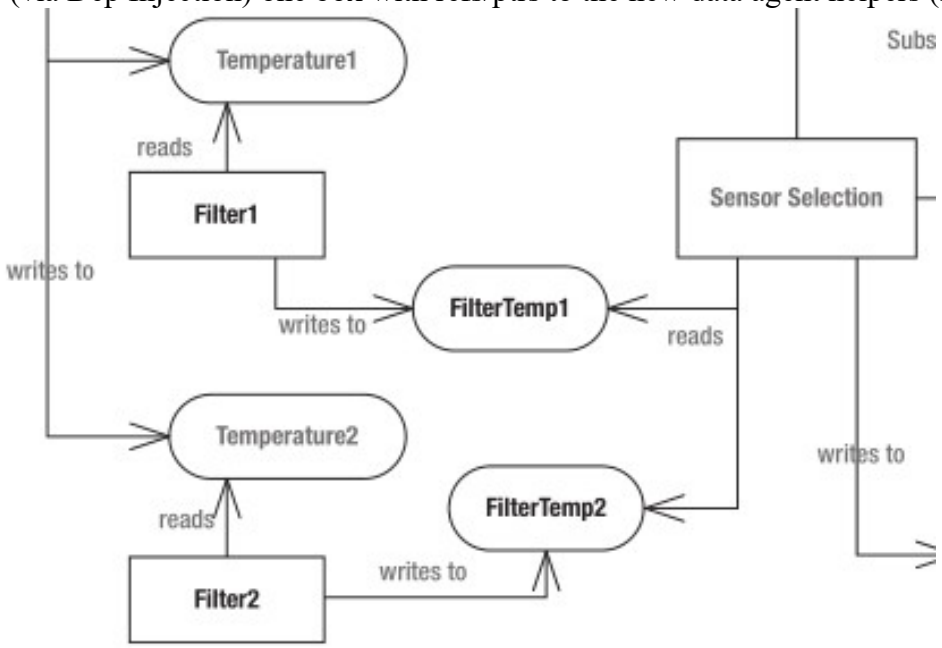


Fig 9.3 – a simple mod (add Temp "filters") – no other modules need to be changed
except to "seed" (via Dep Injection) one box with refs/ptrs to the new data agent helpers (same API, tho)

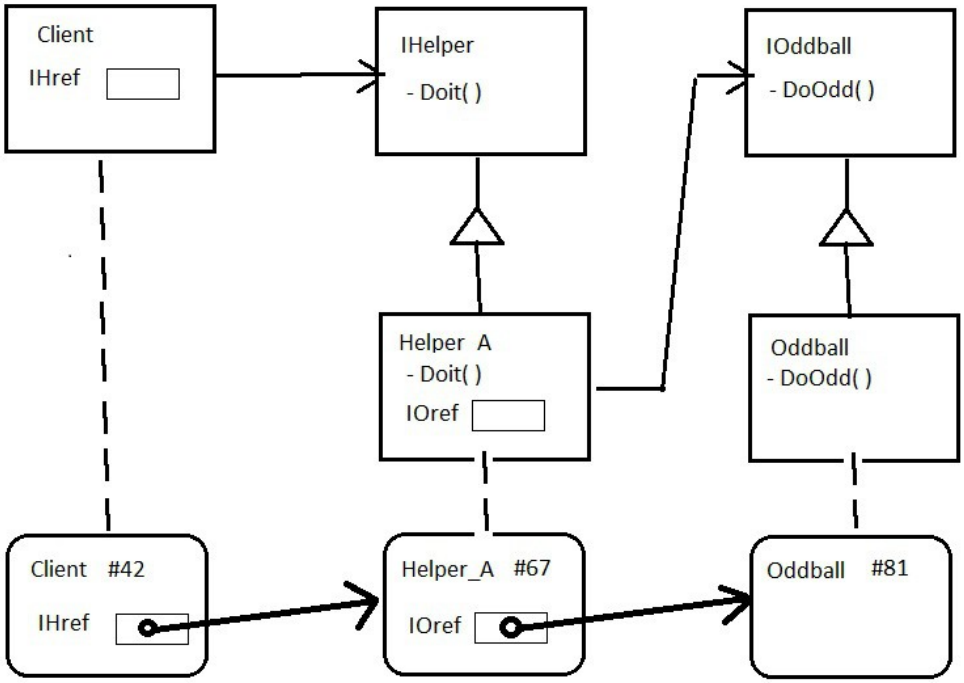


Basic Cutout Pattern

o1- Boxes are Classes (linked-in at Compile-time);
o2- Rounded are Objects/instances created/linked at Runtime – hence the ID numbers like "#42"
o3- Dashed lines are which class an object was "created" from (eg, via "new Foo(<args>)")

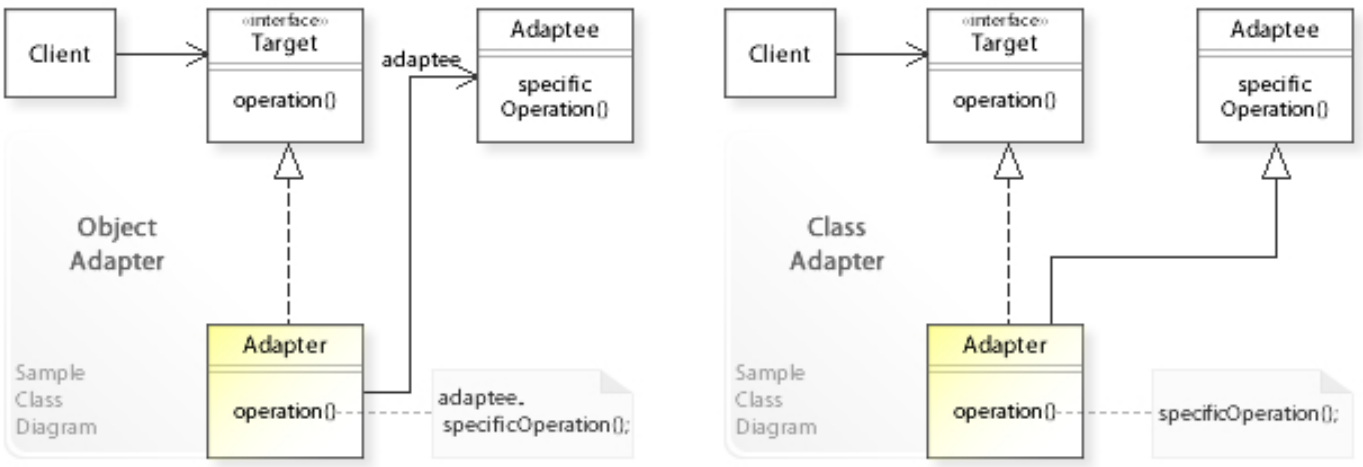
Q: Why?
A: Dependency Inversion:
To prevent Client from Knowing Helper guts (via Class Inclusion)
Client Knows **IHelper**
o- 'I' means Interface/API
o-- Has no method bodies
o-- Has no data slots
Client has **IHref slot**
o- Cli-ob.IHref to Helper-ob
o- Cli-ob #42 is an **Instance**

And –
If Helper is a Wrapper:
(GoF Adapter, maybe)
Then use Cutout again –
Helper knows **IOddball** & has IOref slot
o- Helper_A-ob.IOref to Oddball-ob #81

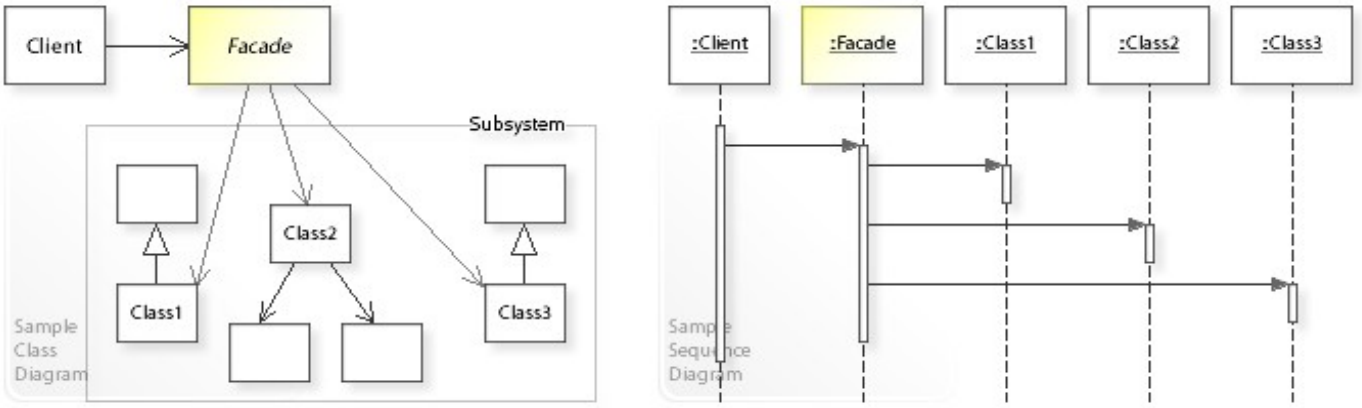


GoF – Design Patterns, 1995

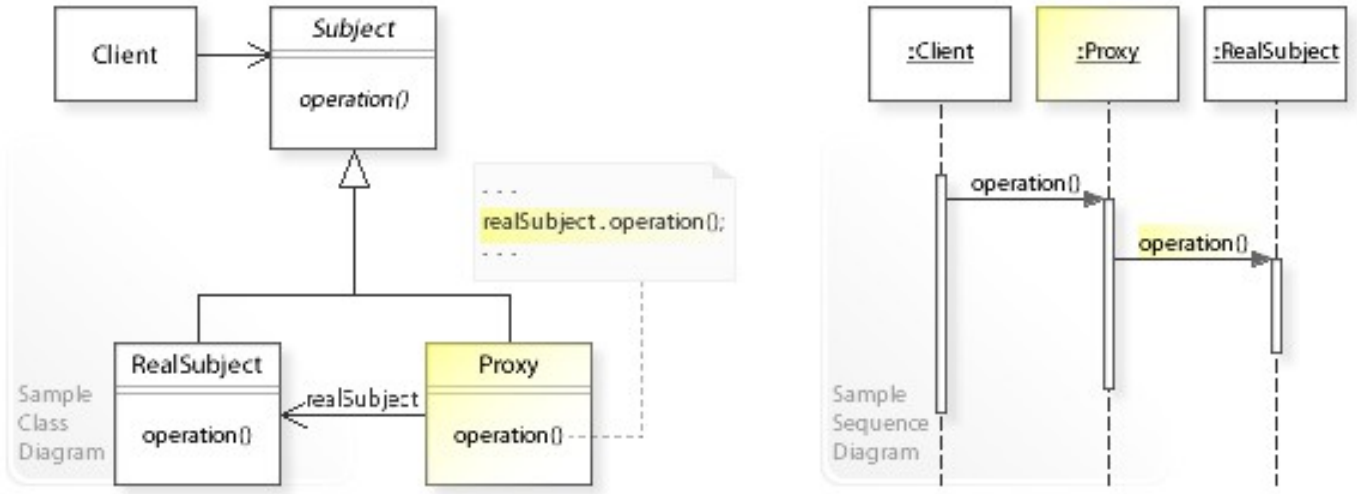
Adapter allows classes with **incompatible interfaces** to work together.
Wraps/hides ugly-API class with/behind a nice-API class.
In **Cutout-ese** language – Adapter=**Helper** & Adaptee = **Oddball**
o- Note below, Adapter-to-Adaptee isn't using a Cutout → higher coupling & less flexible to change
(CF https://en.wikipedia.org/wiki/Adapter_pattern)



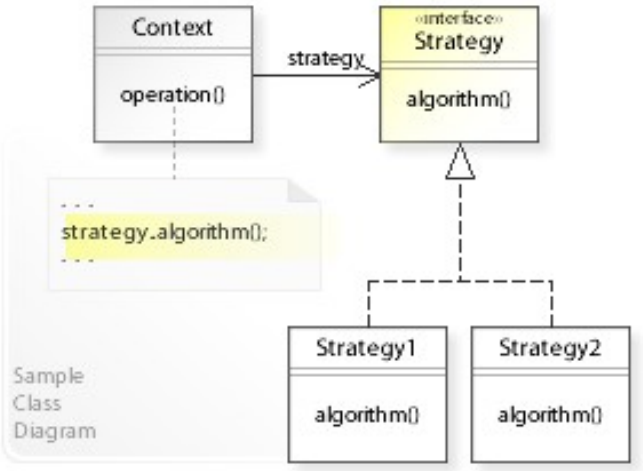
Facade combines and simplifies API to a group of objs.
Wraps multi-class weird-API group of classes with a single class nice-API class.
o- The Facade acts as a cutout so Client doesn't know who/what is helping
In **Cutout-ese** language – Multiple **Oddballs** coord'd by Facade=**Helper + IOref[-]** array of slots to Odds
o- Note below, Client-to-Facade isn't using a Cutout → higher coupling & less flexible to change
o- Ditto for Facade-to-Oddballs

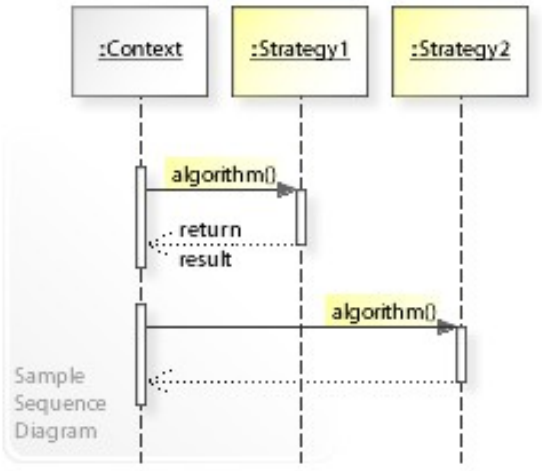


Proxy **controls access to another object**, usually external to your component (or pgm, but Subject prevents it)
o- in GoF: both proxy and inherit same Subject mom class – and Subject should be an Interface
Wraps naked-API class with a controller-API class.
In **Cutout-ese** language – Subject = **IHelper** & Proxy=**Helper** & RealSubject = **Oddball**
o- Note below, Client-to-Subject isn't using a Cutout → higher coupling & less flexible to change
o- However, if Subject is an Interface, then Subject-to-RealSubject is (or can be) a Cutout via an ISref slot
o-- This mech (2 classes inheriting from a mom) is similar to GoF tree-ish patterns

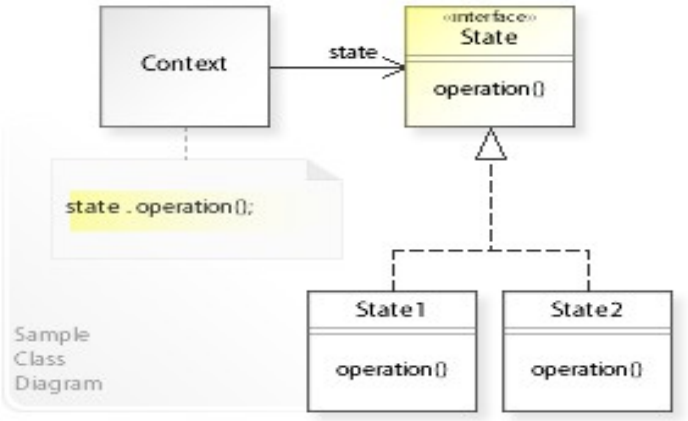


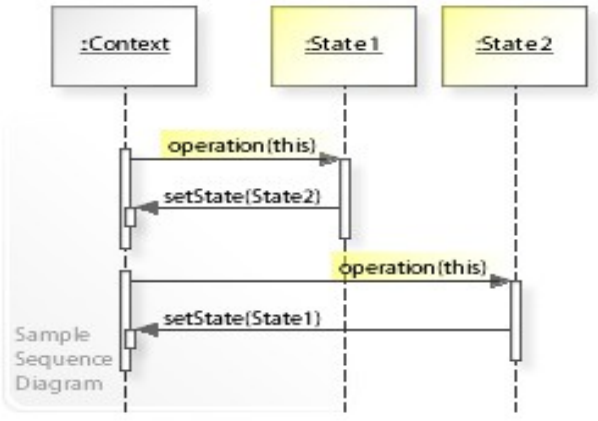
Strategy Client calls a Context (a Coordinator) who calls the currently active Strategy.
Point is to allow hot-swapping (ie, at runtime, AKA dynamically) of different strategies to do a job
  o- Swapped by changing Strategy-link slot in the Coordinator – and Strategy should be an Interface
In **Cutout-ese** language – Client is off picture & calls Context's operation(-) method (hopefully via a Cutout)
  & Context/Coordinator=**IHelper** & Strategies=**Oddballs**
  o- Note below, Context-to-Strategy is a Cutout → lower coupling & very flexible to change





State: Decouple change in FSM State graph (linkages) or a State's Actions from the FSM (= Context) class.
  o- An alternative to an FSM class method containing a Switch statement to access behavior for a current state
  o- State pattern implements a state's behavior as a State object's (sole) method (called "operation(-)", below)
Each State object knows its own behavior, so the Client & FSM/Context don't – **lowers coupling**
  but each State also knows and its outgoing FSM graph links to its next State objects
    o- see pole diagram below where the state objects do Context.setState(-) calls to set Context's next state
    o- So add/del a new State involves adjusting other connected States – **higher coupling** than we'd like
      o-- Old States that will have an edge to the New State need modified to call setState(<New State>)
In **Cutout-ese** language – **Client** is off picture & calls FSM/Context's operation(<event>) method
  & FSM/Context=**IHelper** & States=**Oddballs**
  Also, each State has a **IOref[-]** array of slots linking to its possible next FSM-graph States
    Alt (not shown): setState(next_state_ID), so that the FSM/Context would know all the State objects

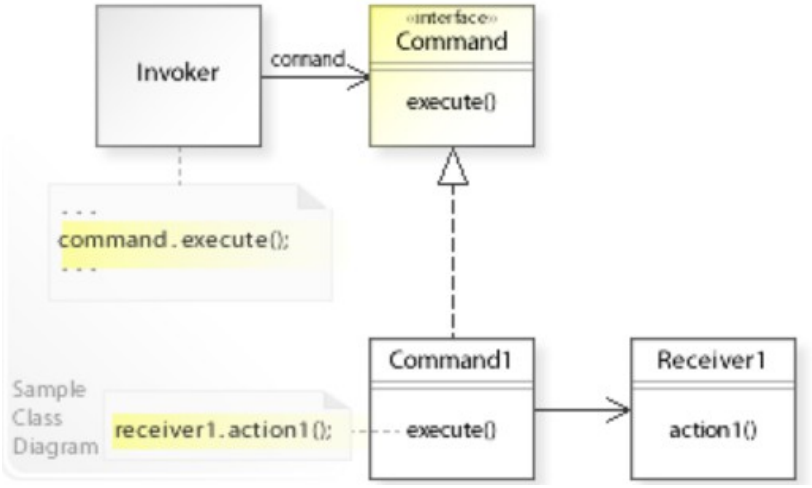


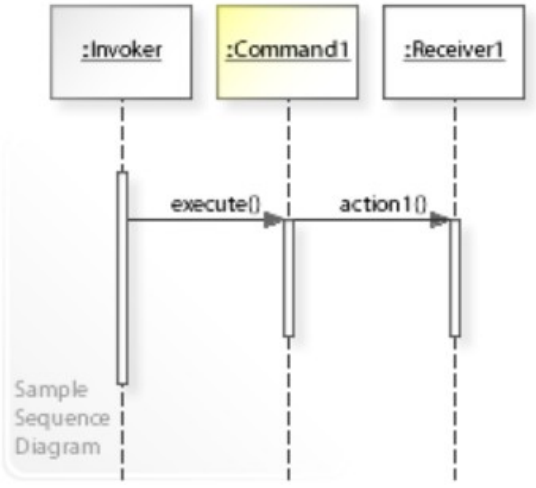


Command Simplify treating Commands and their Actions like Objects – many uses for this
- o- Invoke a Command in multiple ways (GUI button, keystroke, menu, etc.)
- o- Macro record a sequence of Commands for replay
- o- Multi-level Undo/Redo of Commands
- o- Dynamically choose to remotely execute some Commands (so Command must be sent over network)

So, split up knowledge of doing a Command
- o1- How **to call** a Command (the **Invoker**), & collects the arguments – who can also track Command usage
- o2- Preserving the **arguments** to a Command (called the "**Command**")
- o3- How to perform the Command's Action (the **Receiver**)
- o4- and maybe How to undo the Command's Action (another **Receiver**)
- o5- AND How to link these up properly, a Coordinator (usually called the GoF "Client")

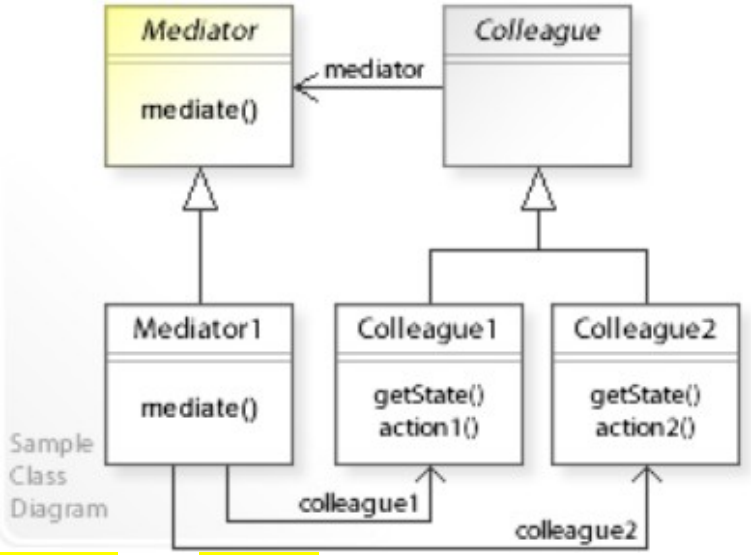
In **Cutout-ese** language – Invoker=**Client** & Command=**IHelper** & Receiver=**Oddball**
- Also GoF-Caller is an outer-Client that calls the Invoker
- Also GoF-Client=Coordinator that links up the Invoker-Command-Receiver triplet of objects

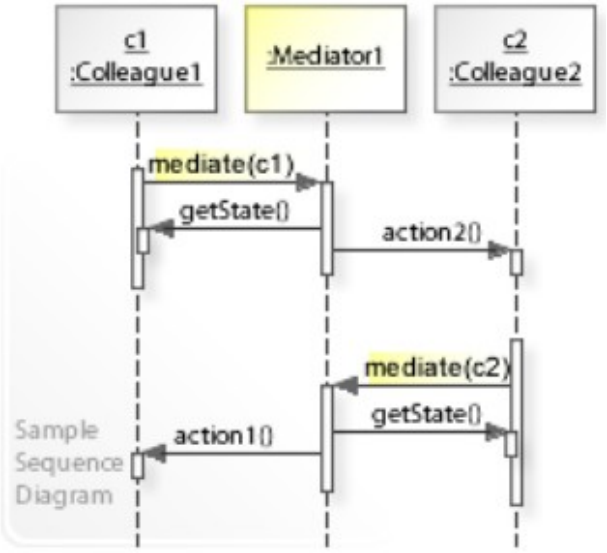




Mediator Agents (Colleagues) don't communicate directly with each other, but instead comm thru a Mediator
- o- Hence, the Agents **don't Know each other** – **low-coupling** between agents
- o- Agent interaction knowledge is contained within their Mediator
- o- So changing an Agent likely only affects the Mediator, and not the other Agents

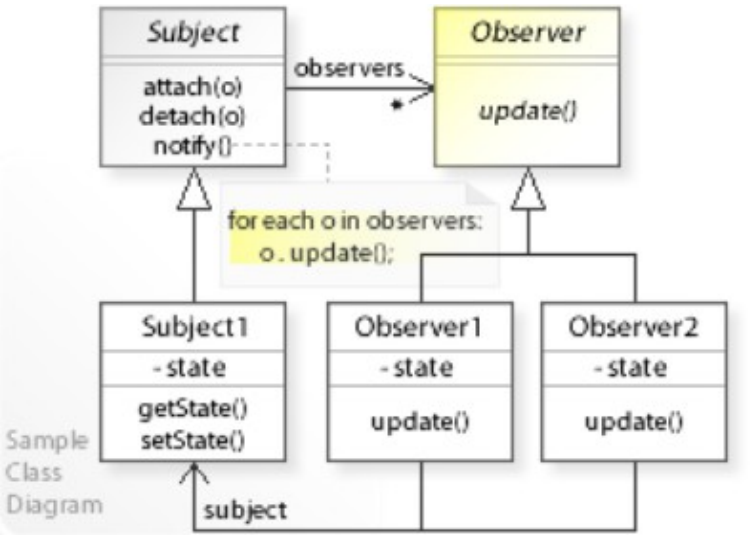
Agents make requests to the Mediator, who passes the requests on to other Agents (and can return results)
In **Cutout-ese** language – Colleague=**Client** & Mediator=**IHelper** & (other) Colleagues=**Oddballs**
Colleagues (both Client and Oddballs) inherit from a group IColleague interface (which is a bit restrictive)
Typically the Mediator has a **IOref[-]** array of slots linking to the Colleagues
EX: Coordinate a multi-step assembly line (maybe a DAG == splitting & merging roadways, not a "line")

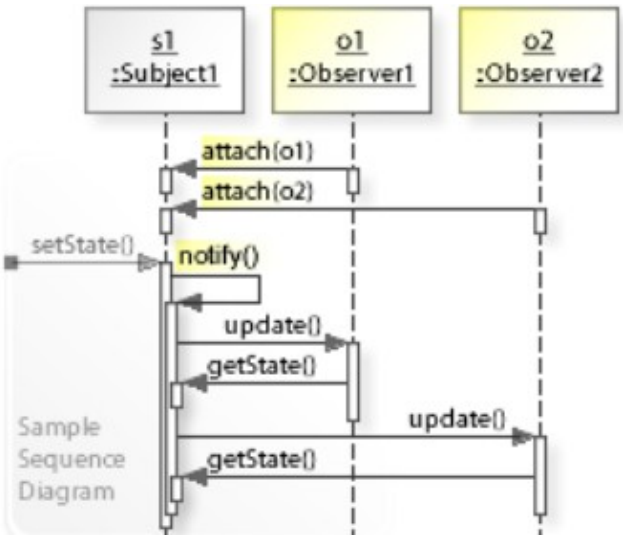




Observer AKA **Pub-Sub** (Publisher-Subscriber) – Agents want to be notified when a Subject changes state.
AKA Subscribers want to get the Publisher's newsletter when it is published.
(As opposed to Agents periodically **Polling** the Subject "Have you changed recently?")
- o- But each Observer does its own Subject → getState(-) call, when the Observer wants the actual data

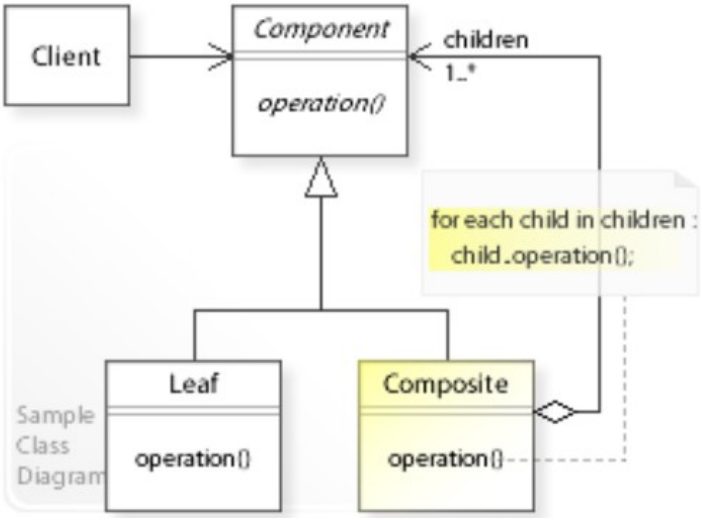
In **Cutout-ese** language – Subject=**Client** & Observers=**IHelpers** & Subject=**Oddball** (hands out actual data)
Typically the Subject has a **IHref[-]** array of slots linking to the Observers

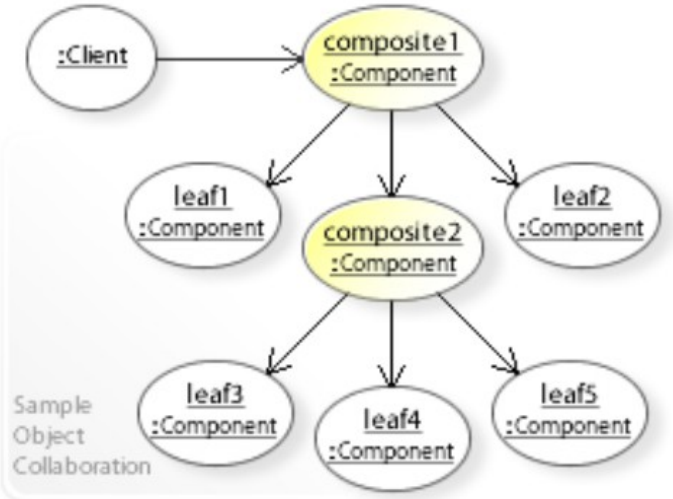




Composite (AKA Tree) unifies Leaf & Composite/Non-leaf agents. Non-leaf has add/del nodes ability
- o- Leaf node is the "Basis Step" of the Tree "recursive data structure"
- o- Composite/Non-Leaf does the std operation(-) by running it on all its kid nodes

In **Cutout-ese** language – Client & Non-Leaves=**Clients** & Components(Leaf & Non-leaf)=**IHelpers**
Typically the Non-Leaf has an **IHref[-]** array of slots linking to other Components

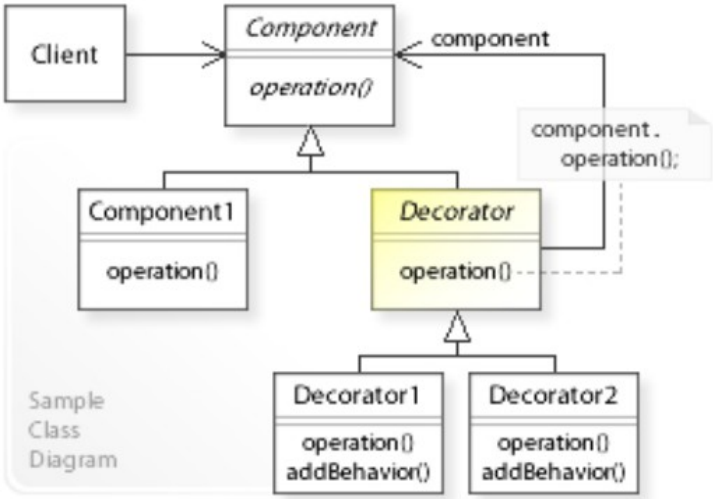


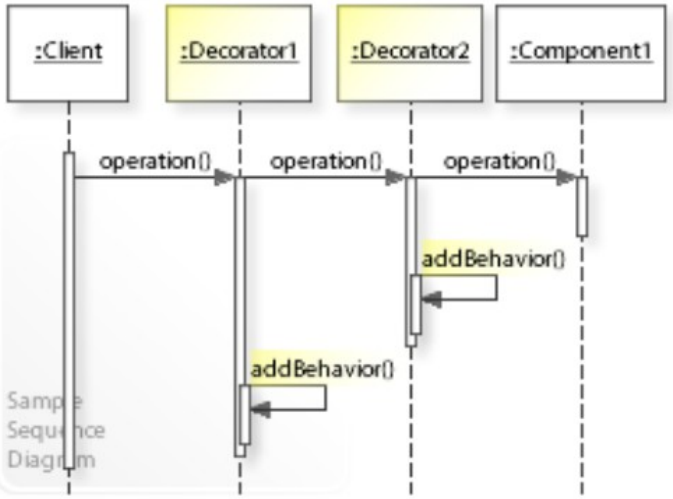


Decorator (AKA Tree/List) unifies Leaf & Decorator/Non-leaf agents. Non-leaf has add/del node ability
- o- Leaf (Component1, below) node is the "Basis Step" of the Tree/List "recursive data strucure"
- o- Decorator/Non-Leaf does the std operation(-) by running it on all its (one) kid nodes

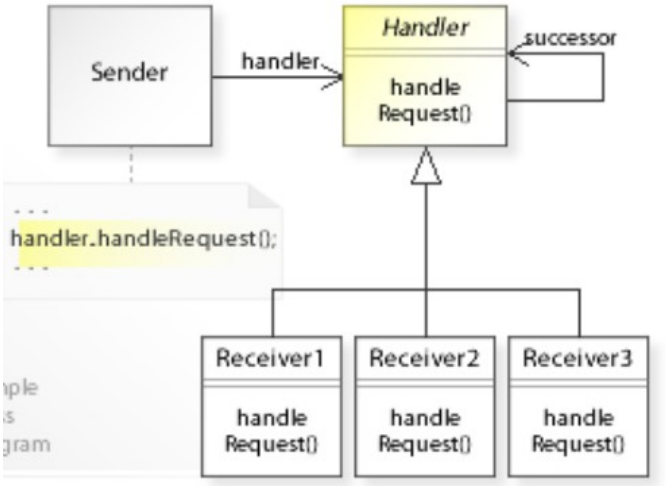
In **Cutout-ese** language – Client & Non-Leaves=**Clients** & Components(Leaf & Non-leaf)=**IHelpers**
The Decorator/Non-Leaf has an **IHref** slot linking to the next Component in the List
- o- Component/Non-Leaf does the std operation(-) by running it on its kid node

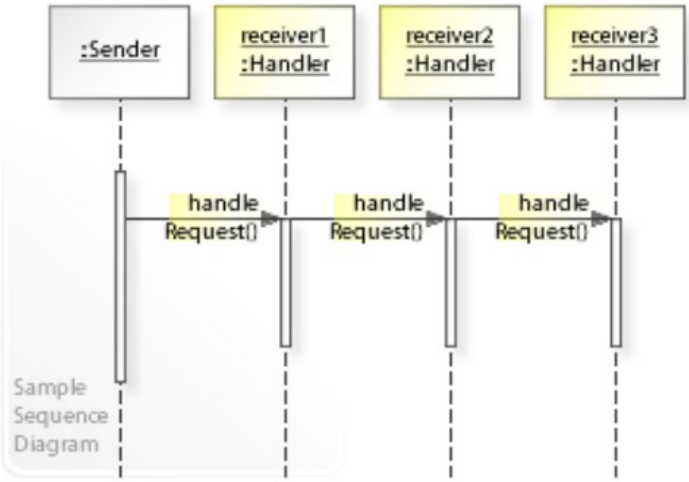
BUT the Non-Leaf usually does **specialty Before/After actions** (as is done in "Aspect-style" pgmg)
The Leaf (Component1) is usually the main object.



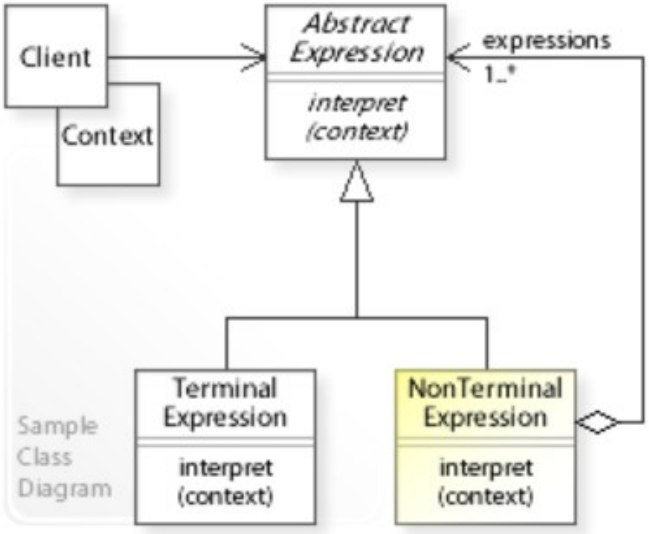


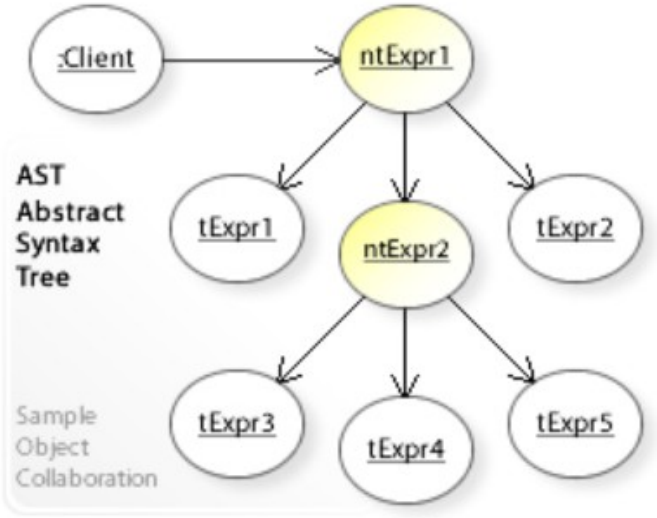
Chain of responsibility like Decorator, but the chain of nodes basically decide **if they want to act or not**
- o1- typically only one node in the chain acts, and all other nodes ignore (or never see) it
- o2- typically there is no beefy main (last) Leaf





Interpreter (AKA Tree) implements a specialized programming language.
(AKA a **DSL** = **Domain-Specific Language** – tailored to your Problem Domain to make it easy to solve)
- o- (**AST**) **Abstract Syntax Tree** of nodes, each has a doit(-), here called interpret(context).
  - o-- (Read a book on Compilers or Interpreters for AST stuff)
- o- Each node has a Kind slot – a different kind of language "construct"
  - o-- So, each Kind acts as needed (eg, a "while" node runs its LH kid "test expr" and runs RH kid "body", or eg, a "+" node would run its LH kid expr, its RH kid expr, and add their values)





GoF "Design Patterns" Upshot –
- o0- We didn't cover them all
- o1- Most GoF Client-Helper-Oddball patterns have (or should have) approximately identical patterns
- o2- Most GoF Recursive (Tree or List) patterns have (or should have) approximately identical patterns
- o3- To the extend that they don't, it is usually because they skip (or overlook) a Cutout when it could be used
  - o-- but to be fair, GoF was the first major book on these couple-dozen-class kinds of patterns

(*) Please pay attention to the diagrams which **blur the distinction between** OOP Classes and Objects

More **9.3.2 Architecture**
Also see 10.3.1 A Brief Taxonomy of Architectural Styles
DB-Centered Arch (AKA Data-Centered)
DB Major Data Parts (**RDB** – no Objects – **SQL**)

> RDB from "Relational Algebra", by Edgar Codd 1970 [nox]
> o- **Tremendously simpler** than prior DB kinds
> o- Took over by late 1970s
> o- Initial RDB "languages", mostly gone: Sequel, SQL, Postgres
> SQL Today (pronouced S.Q.L or "Sequel")

o-many **CRUD** clients (Create, Read, Update, and Delete data)
RDB must be Reliable/Resilient
RDB typically Distributed – DB is split up = diff parts of data in diff PU (Processing Unit) locations
Issues:
Redundancy == duplicate data copies on diff PUs, but local copy changes **must be mirrored** on other PUs
Consistency == all local copy changes **are mirrored** on other PUs
Consistency time delay == how long does it take to get a local data change mirrored?
Atomic (multi-part) transactions (all parts win or all parts lose) = complicates Redundancy, Consistency
Single-computer, multiple-disks (Redundancy) – typically uses "RAID-5" tech
o- Destroy any one (of 5 disks) and no loss of data – similar to doing parity-based Error Correction
EX: MySQL, SQLite (freeware)

Giant DB/Cloud
o- Too much for single-CPU processing – only Distributed or Parallel or both
NoSQL (generic term) for Giant (non-RDB) DBs – cuz it simplifies scaling up – (but scales down, too)
o- **Key-Val DB** (AKA Map/Dictionary)
EX: Redis, Memcached
o- **Document** DB – Hierarchical Structured Data: JSON, XML docs – large and small docs
EX: MongoDB
o- **Graph** DB – directed-edge/arrow relationships between data-objects (no methods)
EX: Neo4j
Issues:
ACID: Atomicity, Consistency, Isolation, Durability
Atomicity: all or nothing rule (parts of transaction) – needed for multi-part transactions
Consistency: only valid data in DB (w.r.t. Constraints) with diff data parts in diff PU locations
Isolation: seq of overlapping xtns can't interfere w each other – regardless of time-order of their sub-parts
o- Diff parts of each transaction can overlap in timing
Durability: DB atomicity even if crash during transaction (during its several sub-tasks)
o- you can pull the power plug & the DB is still safe – all the atomic transactions that completed on reboot
BASE: [**Nox**] (Wordplay on "ACID" from Chemistry)
Basic Availability (AKA local availability) – when can you get (completed) data?
Soft-state (AKA inconsistent for newest local changes – until they are all duplicated properly)
Eventual consistency (AKA consistent for older local changes – cuz they've been duplicated)
o- Badly chosen expressions (hard to remember)

CAP 'theorem': can't have **Consistency**, **Availability**, and **Partition** tolerance (in the cloud),
→ **you have to settle for two out of three.**
(Eric Brewer) [**Nox**]
Partition: split up == distributed data parts on diff PUs
Consistency: == all processors see/know same thing: How?
By no (immediate) **Availability**: (by delaying local access till remote data change is here too)
By no **Partition**: (by having no remote data & PUs) – IE, no cloud – just local data processing
Availability: distant changes are available "immediately"
By delaying local access till remote stuff is visible

o9.2.1 Software Quality Guidelines and Attributes p312
- o- **Non-Fcnl Reqts** (AKA **Quality Reqts**) (AKA the "**ilities**") – not Doing it, but How it gets done
- o- NB, Fcnl Reqts are actions your program takes; Non-Fcnl Reqts are how it gets the job done

There have been a lot of Quality Reqts Checklists – over the last 5 decades
FURPS (learn the acronym)
- o- **F-**unction**ality** (Actually, NOT a **Non-Fcnl** Reqt; included by "committee")
- o- **U-**sab**ility** (Doesn't make the user work hard [**UI Rule #1**])
- o- **R-**eliab**ility** (~ always works when the user needs it)
- o- **P-**erformance (AKA Efficiency) (**fast**, small, frugal)
- o- **S-**upportab**ility** (AKA Maintainab**ility**) (easy to **find/fix/extend**)

Quality Concepts
ISO 9126 S/W Quality Factors [**nox**]
- o-- NB, **ISO** == Internation Stds Org; **ANSI** == American National Stds Institute

FURPS + Portab**ility** [**nox**] (to another CPU, OS, Framework, Platform, Language)
- o- Was superseded a few years ago (now with 30+ ilities, and counting) (**ISO 25010**) [**nox**]

oXX. (S/W) Review Techniques p 325
- o- *"The later you find a mistake/defect, the costlier the fix."*

Kinds, reviews for coding: (usuallly by team mgmt, of junior coders)
- o- **desk check**: informal, **by yourself**, go over the code, try to find mistakes (design & code)
- o- **walk-through**: formal, go over the code w/ 3-6 onlookers, ("Action Items") task issues to be fixed
  - o-- time-consuming (3-6 staff for, say, an hour), so expensive
- o- **inspection** (Gilb 1993 nox): formal, go over randomized (say ~ 20%) (statistical) sample of code, pass if no issues in sample – (*) saves time

** Some people use "walk-through" and "inspection" as **synonyms**
- *- XP's (1993) **pair pgmg** has built-in "continuous review" (two heads at one desk/screen/kybd)
  - o-- Key Pbm with Pair Pgmg == Mgmt: paying two people to do the work of one?

- *- "**Lessons Learned**" **after project** (AKA Post-mortem analysis)
  - o-- to improve processes for next project
- o- **Agile** "**retrospective**", after each "sprint" (AKA devel period)
  - o-- "velocity" (Team feature dev "speed", in story pts per time) & quality, stats
  - o-- story points, estim: (should have error bars), trend line
  - o-- workflow hiccups (what went wrong)
  - o-- umbrella/admin issues // outside S/W dev team
  - o-- big **tech debt** issues (**Tech debt** is stuff you saw that **needs to be cleanup**, but wasn't)

oXX. SW Quality Assurance p 339
SQA == **SW Quality Assurance** (often a dept in a big S/W organization)
Goal: that stds/policies/SW-Dev-procedures are actually followed
Audits: **Reviews** are called "Audits"
Testing: ** To find errors (Mindset: "**Break it**")
SQA checks test planning, test execution processes, and test result docs, & coding stds being followed, ...

(*) **Improve via measurement**

*"To measure is to know." [**nox**]*

*"If you can not measure it, you can not improve it." [**nox**]*

Q: What are you not measuring?

Q: Can your measurements **predict reliably**? (most S/W measurements don't)

o- We collect measurements to predict the future (with error bars)

Q: What % of devr time do your measurements **take away from productivity**? (a reason for doing less meas'g)

Q: Is it simple to measure, or costly?

Std things to measure:

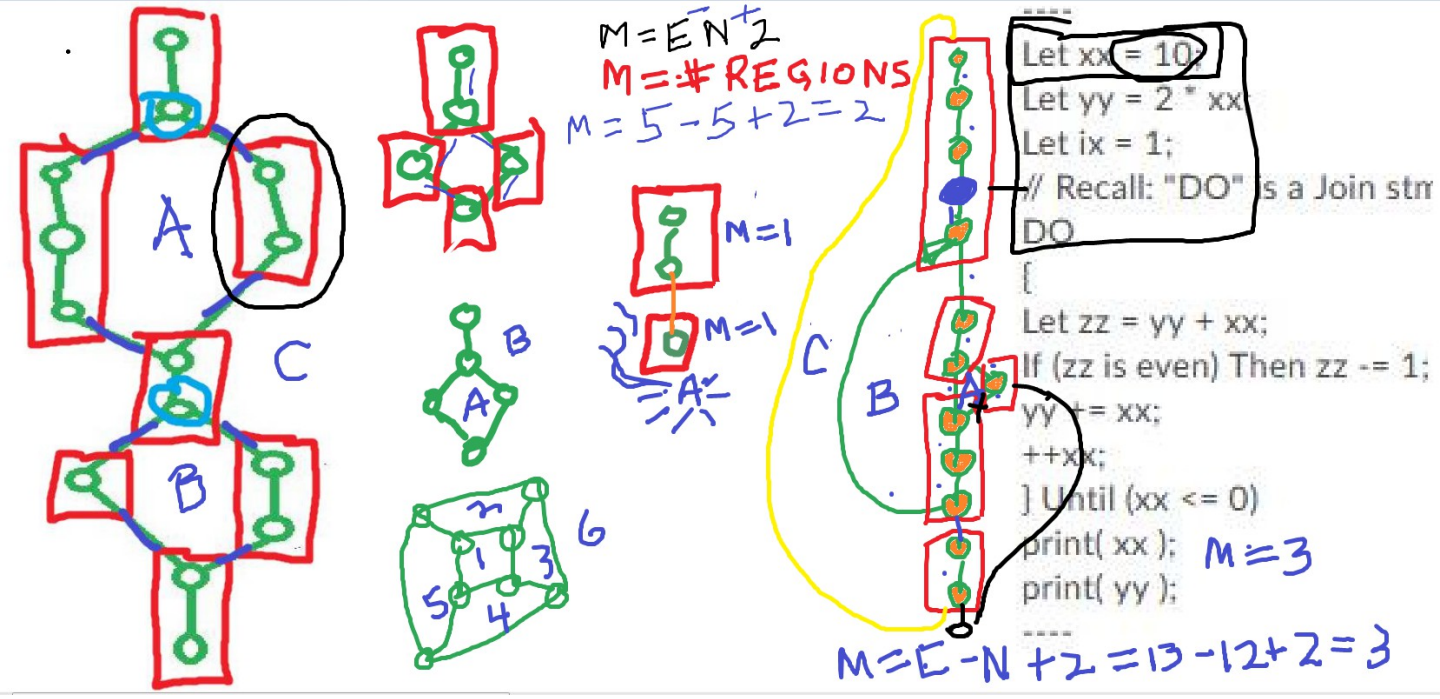
LOC/SLOC (KLOC/KSLOC) = **Lines of Code**, or Source Lines of Code, K = thousand
(usually has big error bars – 3x?)

Function Points (> 4 params; need 100s of hours to learn to do it well; used for up-front BUFD estimates)
"With 4 parameters, I can model an elephant; with a 5th parameter, I can make the elephant dance."
– John von Neumann (the Martian)

Cyclomatic Complexity (how many cycles/loops are in the program; done for a "unit" of code, or a fcn)
o-- "Unit" of code = one person, dev from 1 to 20 days (3 weeks)
o- Works for small fcns, mostly
o- Used as a measure of how complicated a Unit's design is – in a code review

Cyclomatic Cplxty

Created by McCabe, henced denoted "M"
o- Normal counting via "MEN2" → **M = E – N + 2**
N = nodes (AKA straight-line code segments; AKA "Basic Blocks")
o-- Node **ends** with A) a **branch** stmt (2+ ways out), or B) a "**return/exit**" stmt
o-- Node **starts** with A) an "**entry point**" (first) stmt or B) a "**fan-in**" or "**join**" stmt
E = edges between "nodes"
o- Alt: [what I use] Count the number of planar graph regions – you can "eyeball" it
o- Loop back up to a node is just an edge splitting at the loop bottom and joining at the loop top
o- Personal variation – Allow well-formed switches as one loop/region



Q: What's the Cyclomatic Complexity of this piece of code?

```
if (edges == 0 || subsets == 0) print("Warn Conga #1");
gInsertions += np;
if (hms == 0) {
..hms = 1;
..unsigned long n = np;
..while (n > 2) {
....n /= 2;
....hms++;
....}
..}
if (hms == 1) print("Warn Conga #2");
if (hms > MaxCongaSubsets) print("Warn Conga #3");
how_many_subsets = hms;
subset_sizes.Allocate(hms);
```

Q: What's the Cyclomatic Complexity of this piece of code?

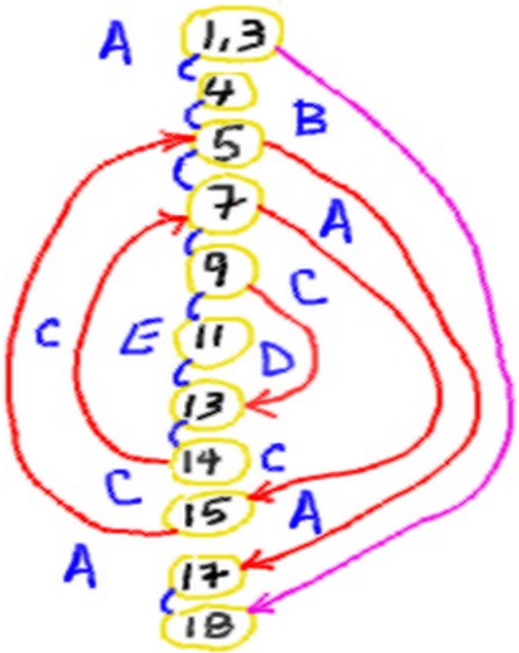
```
if (edges == 0 || subsets == 0) print("Warn Conga #1");
gInsertions += np;
if (hms == 0) {
..hms = 1;
..unsigned long n = np;
..while (n > 2) {
....n /= 2;
....hms++;
....}
..}
if (hms == 1) print("Warn Conga #2");
if (hms > MaxCongaSubsets) print("Warn Conga #3");
how_many_subsets = hms;
subset_sizes.Allocate(hms);
```

Q: What's the **Cyclomatic Complexity** of this piece of code?

- o- Size = 18 LOC – 8 action LOC – 2 decl LOC – 8 delim LOC
- o- NB, inside UC Main Scenario's 3-8 "Steps"
- o- NB, using brace on line #18 as tgt for return branch on line #3; and #14 as tgt for #13 and #13B stmts
- o- NB, for a fcn with extra Return stmts, supply an extra bottom node for them to go to, in doing Cyclo Cplxty.
- o- Alt: i**f we count Branch stmts** (Ifs and LOOPs) and add 1, we get M (because of correct nesting)
- o- Somewhat moderate complexity code; and M = ??
- o- High-complexity is M > 10-ish (→ > 1,000 distinct computational paths thru the code)
- o- To Reduce Complexity, put some of it in its Own Box (fcn/md) – well-named (= verb+object(s)).
- o- Number of distinct paths, (each touching at least 1 different stmt) – #paths >= 2^#branches

```
1. int foo (int ra, int rb, int rc)
2. {
3.   if (0 => ra ) return 0;
4.   int retv = 0;
5.   while (0 < rb)
6.     {
7.       for (int ix = rc; 0 < ix; --ix)
8.         {
9.           if (0 == (rc % 2)) // rc even?
0.             {
1.               retv = rb * rc;
2.             }
3.           else { retv = rb + rc; } //odd
4.         } // end for
5.       --rb;
6.     } // end while
7.   return retv;
8. }
```

(corrected graph)



o19 SW Testing Strategies p 373

ITG = Independent Testing Group
- o- better at finding bugs
- o- costs more
- o- typically used by very big companies

Kinds of Tests commonly used in S/W:
- o- **Unit** T // TDD == Test-Driven Dev (usually used by some Agile teams) // earliest tools eg, JUnit
- o- **Integration** T (for joining modules/code built by diff pgmrs)
- o- **I&T** == **Integration & Testing** (for joining **major parts**, the big stuff) // older: merge S/W & H/W
  - o-- Major APIs work as expected
- o- **Smoke** T: See if a "build" of the whole program starts up (and maybe try a couple features)
- o- **System** T (usually for joining HW & SW) // See if the entire system does "reasonable stuff"
- o- **V&V** == Verification and Validation (V.2 can be Acceptance T, usually at the customer's site)
  - o-- Verif == works per spec – (spec is formal written reqts)
  - o-- Valid == users like it (assume they are fair users)
  - o-- IV&V == Independent Verification and Validation (done by outside group)
- o- **Acceptance** T: Validation on-site, at the customer's site – (sometimes)
- o- **Alpha** T: give system to **in-house** pseudo-users (not on dev team; AKA fresh eyes) – to keep bugs in-house
- o- **Beta** T: give system to small sample of **external** users, not employees (often with NDA)
- o- No such thing as a "Gamma T"
- o- **Recovery** T: (can sys recover from "planned for" faults?) **Inject faults** and see if sys recovers
- o- "**Pen**" (Security) T: (AKA **Penetration**) can sys be crashed/captured by BGs (bad guys)
- o- **Stress** T: check max sys loading (eg, transactions per second) – usually you plan to handle 1.5x or 2x load
- o- **Performance** T: does sys work at req'd perf level – or how good is the perf? (per non-fcnl reqts)
- o- **Configuration** T or **Deployment** T: does sys work on all target OS's, in all feature combo configs
- o- **Installation** T: does this come up/run on user's H/W config w features they bought – done in-house
- **(*)** Regression T**: Verify old bugs remain fixed/dead -- don't regress to a time when these bugs were alive
  - o-- Every bug fix has a test; it is added to the "regression test suite"
  - o- NB, Regression suite is almost as important as Morale, and equal with Rules 0,1,2 (Fast,Optim,Hunts)

(*) You are Never done testing -- always bugs left

(*)* Testing Mindset – "Write tests to try and **Break It!**"

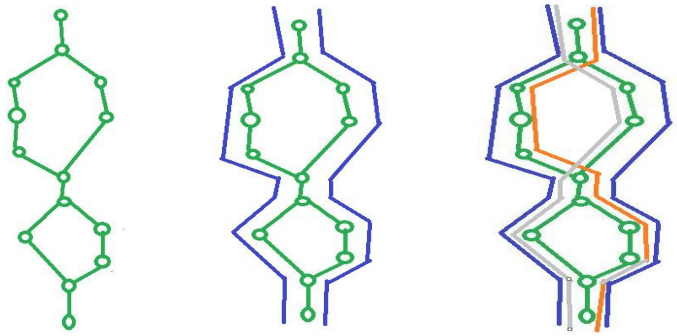
Black Box and White/Glass Box Testing

(*) You can't see bugs until you test.

Black box Testing: can't see the code inside the box – painted black
- o- Test the API & the "concepts" the box represents

White (AKA Glass) box Testing: can see code working

Code Flow **All stmts** **All paths**
M=3 (Branch Coverage) (Paths Coverage) → **grows exponentially** w #branch points



- o- **Branch Coverage** (min) <= **McCabe Cyclo Cplxty M** <= **(All) Paths Coverage**

(*)** If you want to understand why there are **Always More Bugs** in a shipped S/W product,
it is because **you can't** even reasonably **test all paths** (ignoring how many times you go round a loop)
– There are **ALWAYS** conditional control paths (from entry to return) **that haven't been tested**
– and that is also why there is still an effort to prove (small) programs correct (**Formal Methods**)
– and why there's still an effort to do **Functional Pgmg** as much as possible (same args → same result)
– and why we avoid using **global vars**, and **separate** code into modules, build & test a **bit at a time**, etc.

To check, put outputs at the start of each branch (including after a loop)
- o- **All loop iterations,** run to end
- o- **All simple conditions** (w all their var values?) – else how do you know they all work?

EX: `if ((3 <= xx) && (xx < len) && (KRED == color))`
==> **3 conds** ==> 2^3 = **8 test combos**
- o- All var/slot values run -- never done
- o- **Call Tree**: log/print **entry** to every fcn at fcn body start.
- o- **Entry-Exit Tree**: log both **entry and exit** to each fcn.
- o- **Entry-Exit State Tree**: log **state+args on entry** and **state+retval on exit**.

Black box Testing: "can't" see the code inside the box – painted black
- o- All I/O "ports" run: argl + retval – There may be more than one entry point to the code
- o- All Input vals run (each combo for the input args) – usually too expensive
- o- All EIO pairs – & maybe add them to the Regression Suite

Range of Values Testing: EX: an int value, enum months, floats
- o1. **All "Boundary" values** (both inside & outside)
  - EX: **int xx** = 0; // xx runs **from 0 to len-1**.
  - **Boundaries** (Bdys) – AKA"Corner Cases":
  - Bdys: **inside**: xx is{0, len-1} should work.
  - Bdys: outside: xx is {-1, len} for catching errors?
- o2. **Random interior sample** – of interior values
  - EX: var xx = values from 0..99 – say these four {5, 31, 66, 82}

o25.6.1 Software Sizing p511 (8th ed in Ch 33)

Based on: **4 Keys** –

o1- **Code Size** – the degree to which you have properly estimated the size of the product to be built

o-- Estimated in LOC (by summing over all estimated sub-tasks) or in FP (Function Points)

o-- LOC for the same function can easily vary by a factor of 2

> FP Cons – Pressman & Maxim, p481, 23.7 Software Measurement
>
> Opponents claim that the method requires some "sleight of hand" in that computation is based on **subjective** rather than objective data, that counts of the information domain (and other dimensions) can be **difficult to collect after the fact**, and that FP has **no direct physical meaning**—it's just a number.

o2- **Effort from Size** – the ability to translate the Size estimate into human effort, calendar time, and dollars

o-- Historical Baseline – requires "reliable software metrics from past projects"

> Pressman & Maxim, p481, 23.7 Software Measurement
>
> However, to use LOC and FP for estimation (Chapter 25), an **historical baseline** of information must be established. It is this historical data that over time will let you **judge the value** [ie, accuracy] of a particular metric on future projects.

o3- **Team Abilities** – the degree to which the project plan reflects the abilities of the software team

o-- You must "know" your people well

> **Steve McConnell**, p568, in Making Software: What Really Works, by Oram, 2010
>
> o- Author of textbooks – eg, Code Complete, Rapid Development, and Software Estimation
>
> The original study that found **huge variations in individual programming productivity** was conducted in the late 1960s by Sackman, Erikson, and Grant [1968]. They studied professional programmers with an average of **7 years' experience** and found that the ratio of initial coding time between the **best and worst** programmers was about **20 to 1**; the ratio of debugging times over **25 to 1**; of program size **5 to 1**; and of program execution speed about **10 to 1**. They found **no relationship** between a programmer's **amount of experience and code quality or productivity**. …
>
> In the years since the original study, the general finding that **"There are order-of-magnitude differences among programmers"** has been confirmed by many other studies of professional programmers [Curtis 1981], [Mills 1983], [DeMarco and Lister 1985], [Curtis et al. 1986], [Card 1987], [Boehm and Papaccio 1988], [Valett and McGarry 1989], [Boehm et al. 2000]. …
>
> When I was working at the Boeing Company in the mid-1980s, one project with about **80** programmers was at **risk of missing** a critical deadline. The project was critical to Boeing, and so they moved most of the 80 people off that project and brought in **one guy** who finished all the coding and delivered the software **on time**.

o4- **Stable Reqts** – stability/unchanging of product requirements and the (development) environment support

o-- Does it matter if the Reqts are Stable?

> **Steve McConnell** (p58 in Software Estimation: Demystifying the **Black Art**, 2006)
>
> "potential differences in how **a single feature** is specified, designed, and implemented can introduce cumulative differences
> of a **100x or more** in implementation time"

(*)* One Feature → 100x impl time variance from an initial Estimate == very large Error Bars

o-- Agile M.O. assumes **Reqts can not be stable**

Empirical models (eg COCOMO III (Boehm), or Function Points)

Cons:

- o- needs good baseline for knobs settings (AKA parameter values)
- o- poor sensitivity → change knobs slightly, but get big (non-linear) change in results?
- o- too many knobs to be trustworthy – famous quote by John von Neumann, via Fermi via Dyson

> Freeman Dyson being schooled on model parameters by Enrico Fermi:
>
> In desperation I asked Fermi whether he was not impressed by the agreement between our calculated numbers and his measured numbers.
>
> He replied, "How many arbitrary parameters did you use for your calculations?"
>
> I thought for a moment about our cut-off procedures and said, "Four."
>
> He said,
>
> "I remember my friend Johnny von Neumann used to say,
> with four parameters I can fit an elephant, ["fit" means "make a model for"]
> and with five I can make him wiggle his trunk."
>
> With that, the conversation was over.

More on Team Ability Estimates

> **Steve McConnell**, p568, in Making Software: What Really Works, by Oram, 2010
> [A Tale of Two Spreadsheets]
> One specific data point is the **difference in productivity** between Lotus 123 version 3 and Microsoft Excel 3.0. Both were desktop spreadsheet applications completed in the 1989–1990 timeframe. Finding cases in which two companies publish data on such similar projects is rare, which makes this head-to-head comparison especially interesting. The results of these two projects were as follows: Excel took **50 staff years** to produce 649,000 lines of code ... Lotus 123 took **260 staff years** to produce 400,000 lines of code … Excel's team produced about 13,000 lines of code per staff year. Lotus's team produced 1,500 lines of code per staff year. The difference in productivity between the two teams was **more than a factor of eight**...
>
> Both Lotus and Microsoft were in a position to **recruit top talent** for their projects.

(*)* Management style trumps individual team member abilities

Combine Multiple Estimates & **Guess** Distribution **Mean (but not Variance/Std Deviation)**

Simple 3-point weighted approximation to a **Bell Curve** (AKA Gaussian, Normal Distribution Curve):

- o1- Do 3 Estims: S=**Optimistic**, M=Normal/middling, L=**Pessimistic**
- o2- Combo Estim of the Mean: **Size = (S + 4*M + L)/6** – very rough approx of a bell curve – Eqn 25.3 p519
  - o-- Often the L is much larger than the S is smaller – lop-sided toward the L side
  - o-- Take (L – S)/2 as 1 std deviation from the mean; hence, Mean + (L – S) "could" be the 95% win Pbb

Con:

- o- Very rough
- o- This isn't a good explanation for Mgmt

o McConnell – Estimate vs Commitment

(*)* Always add a likelihood/Probability % to your estimate (eg, "80% of the time") when giving it to Mgmt

- o- Mgmt think – Estimate == a Commitment & a Guarantee to doing the job in that time with that budget

o25.7 Project Scheduling 520 (8th ed Ch 34)
o25.7.1 Basic Principles 521

1. **Compartmentalize**: Make the WBS
2. **Interdependence**: Task B can't start till Task A is done
3. **Time-Allocation (to tasks)**: duration, start-end; LOE (Level of Effort) (eg 50% time)
4. **Effort validation** (AKA **Max #Tasks in Parallel)**: based on **who is available** to work them?
5. **Defined Responsibilites**: who gets each task, for scheduling
6. **Defined Outcomes**: What proves a task is completed?
7. **Defined Milestones**: Date to review/approve a **cluster of completed tasks** (eg, for a feature)

o25.7.2 People & Effort p 522 (8th ed 34.2.2)

- o- When fallen behind, adding more people usually doesn't help.
- o- **Brooke's Law**: "Adding manpower to a late software project makes it later."

Why?

- **o1. Up-to-Speed Time, Pgm** – getting the added people familiar with the pgm being developed
- **o2. Dev-Teacher Time** – dev'rs taking time away from working tasks to teach the added people
- **o3. Comm Time Increase**: usually increases by N^2, where N = #staff – but heavily mitigated by Standups
- **o4. Up-to-Speed Time, People** – who does/knows what, who's easy/hard to talk with

Lab: CF 343-p3-vcs-merge-crc-arch-draft.pdf

from Brooks' Silver Bullet paper

- o-**5.3 Incremental Dev [and Review by Cust]: Grow, don't Build**

"The **morale effects** are startling. Enthusiasm jumps when there is a running system, even a simple one."
"One always has, at every stage in the process, a working system."

Recall: **Morale is the Most Important Thing** in S/W Development

o SW Proj Scheduling

Time Concepts: **Lead, Lag, Slack/Float**

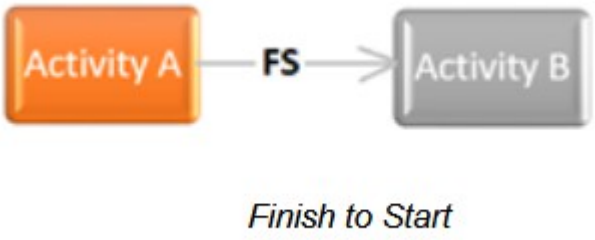
Informally:

o-- Kid Task – depends on its mom task(s) finishing first (DAG: graph parent-child node stuff)

o- **Earliest start time (ES) -** The earliest time, a task can start once previous mom tasks are over (or started)
o- **Earliest finish time (EF) -** This would be (ES + task duration) for the kid (or mom) task
o- **Latest finish time (LF) -** The latest time a task can finish without delaying the project (or a kid task)
o-- Alt: without delaying any of its kid tasks
o- **Latest start time (LS) -** This would be (== LF - task duration)

Inter-Task Constraint Relationships:

o- **Parallel** (ie, no constraint relationships between tasks)

<table>
<tr>
<td>o- Finish-to-Start (FS) – S/W Engr Normal
 

Finish to Start</td>
<td>o- Start-to-Start (SS) == Req'd delay for kid
 

Start to Start</td>
</tr>
<tr>
<td>o- Start-to-Finish (SF) == kid has a hard finish time/date
 

Start to Finish</td>
<td>o- Finish-to-Finish (FS)
 

Finish to Finish</td>
</tr>
</table>

Lag Time: Intended Delay
- o- FS with a Lag: mom.end (+delay) to kid.start // EX kid coat waits for mom-paint coat to dry
- o- SS with a Lag: mom.start to kid.start // EX do complex rebar work before setting forms around it
- o- FF with a Lag: mom.end to kid.end // EX after concrete pour is done, wait 10 days to remove forms

Lag Time: Unintended Delay
- o- FS with Slack time: kid could start earlier (mom's already finished) but there is nobody available
  - o-- Short (and usually reasonable) staffing size
  - o-- Requires S/W domain expert (who is not yet available) – eg, SQL (DBA – DB "Administrator")

Lead Time: Interleaved:
- o- From kid.start to mom.end
  - o-- Apparently, kid only depends on an "earlier part" of mom – maybe mom could be further split
- o- Equiv to splitting mom into non-dep + dep sub-tasks
- o- Alt: Equiv to splitting kid into non-dep + dep sub-tasks

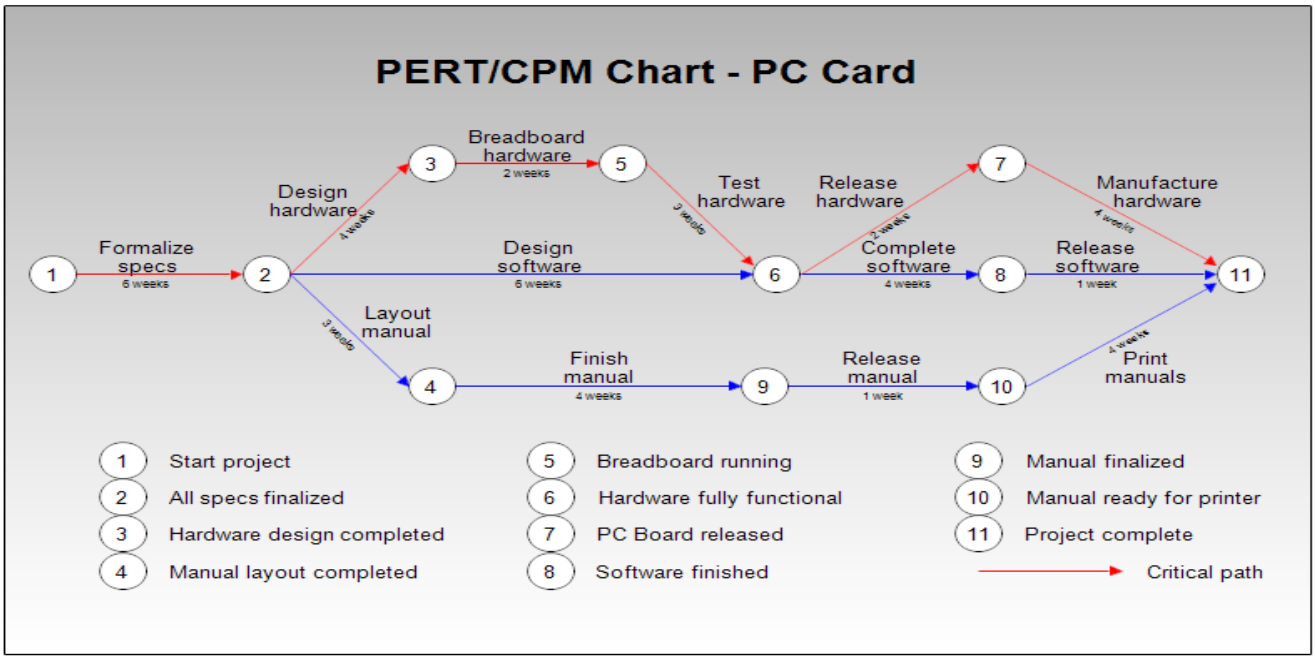
EX: Start painting-walls (on all drywall sections finished), don't wait till all drywall is up
- o-- Means we DON'T have to create a blizzard of small drywall-next-4-feet tasks & their paint tasks

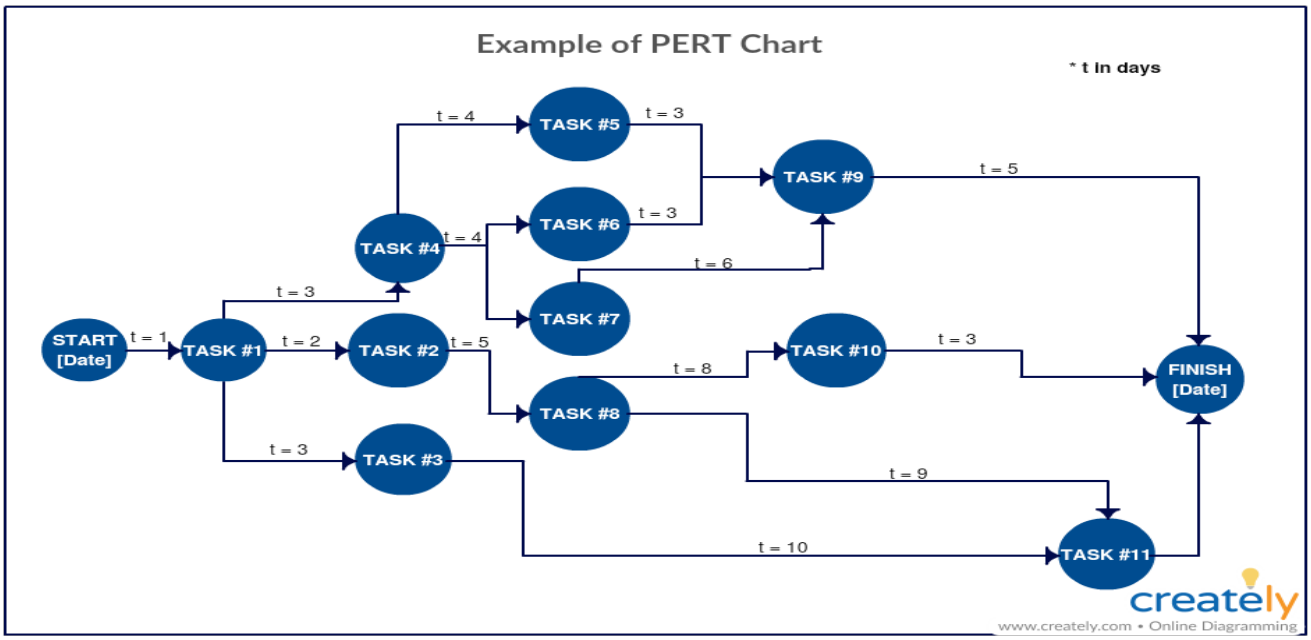
Slack (AKA "free float" or just "**Float**")
- o- kid-**start movable** in interval from mom.end to kid.start
  - o-- Like **kid floating in a bathtub** – mom.end **is left side** of tub & grandkid.start **is right side**
- o- But **kid can't push mom or grandkid tasks**

Represent task dependencies

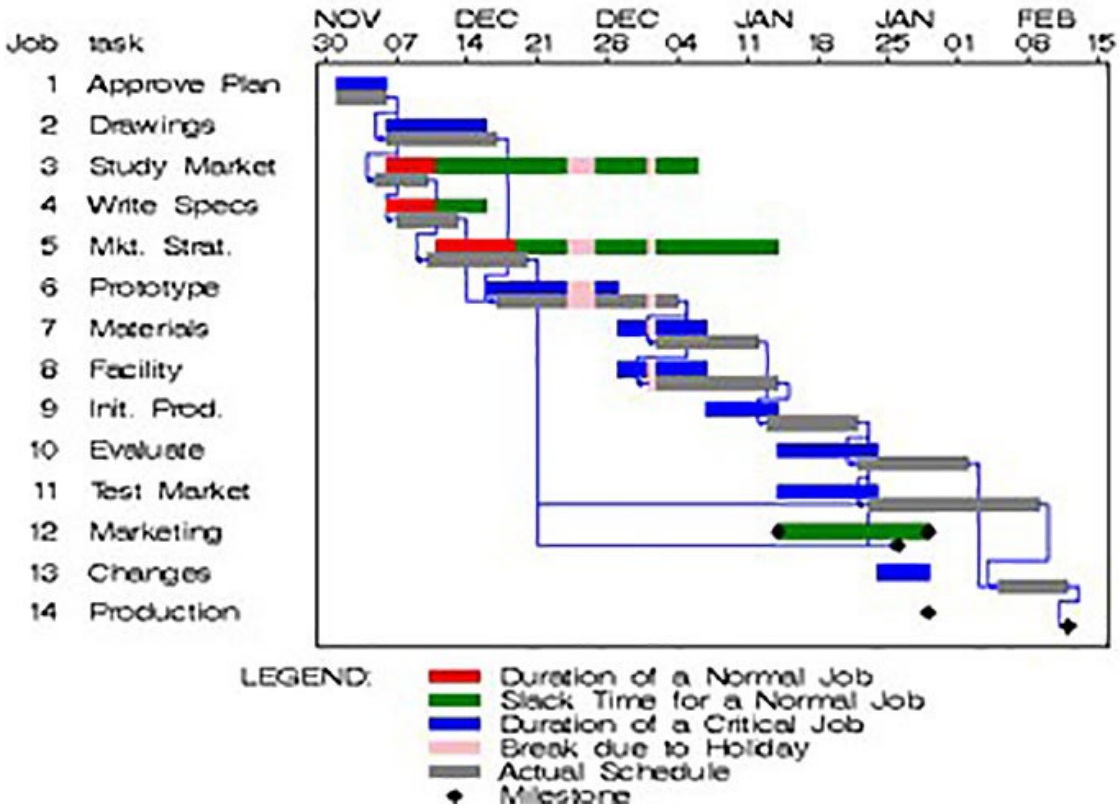
Use a **PERT chart** **mech** to generate a **graph**
- o- based on **mom-to-kid inter-task dependencies** (for S/W, normally FS dependencies)
- o- and then use a CPM (**Critical Path** Method) mech to find **minimum project timeline**
  - o- Critical Path is required – if you believe the inter-task dependencies you've laid out

PERT Chart: (Pgm Eval and Review Technique, nox) (Charts from Wikipedia images)
AOA Style: "**Activity on Arrow**"
edge == duration (in Graph Thy, called "edge weights")
o- Task Name on edge, Sometimes
node == milestone (AKA Task Completion Event)
Alt: **AON**: Activity on Node :: node == task+duration :: edge == milestone & kid task
Big Question: What is the minimum time to complete the project, given the Task Times?
(*) AKA the "**Critical Path**" **– tasks on minimum-time path (from start to finish)**
**** There is NO SLACK/FLOAT on the Critical Path**

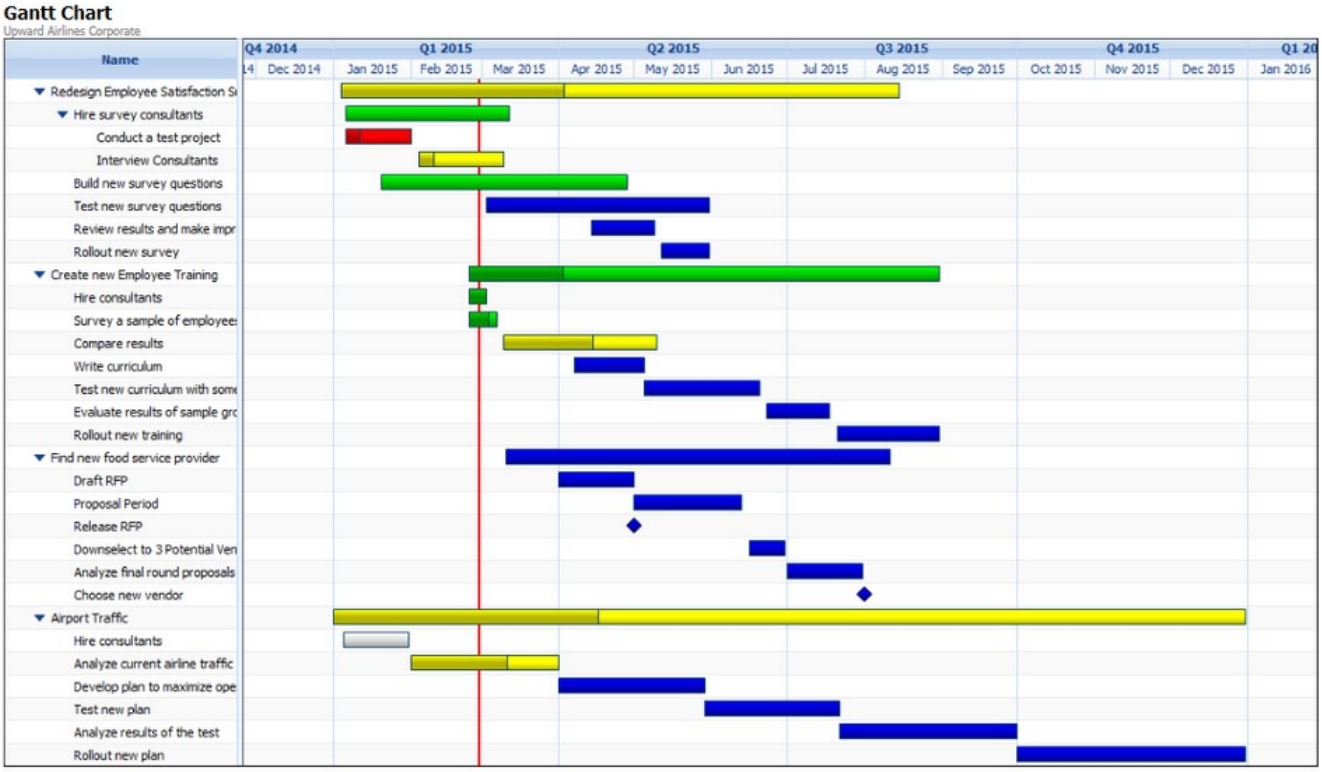


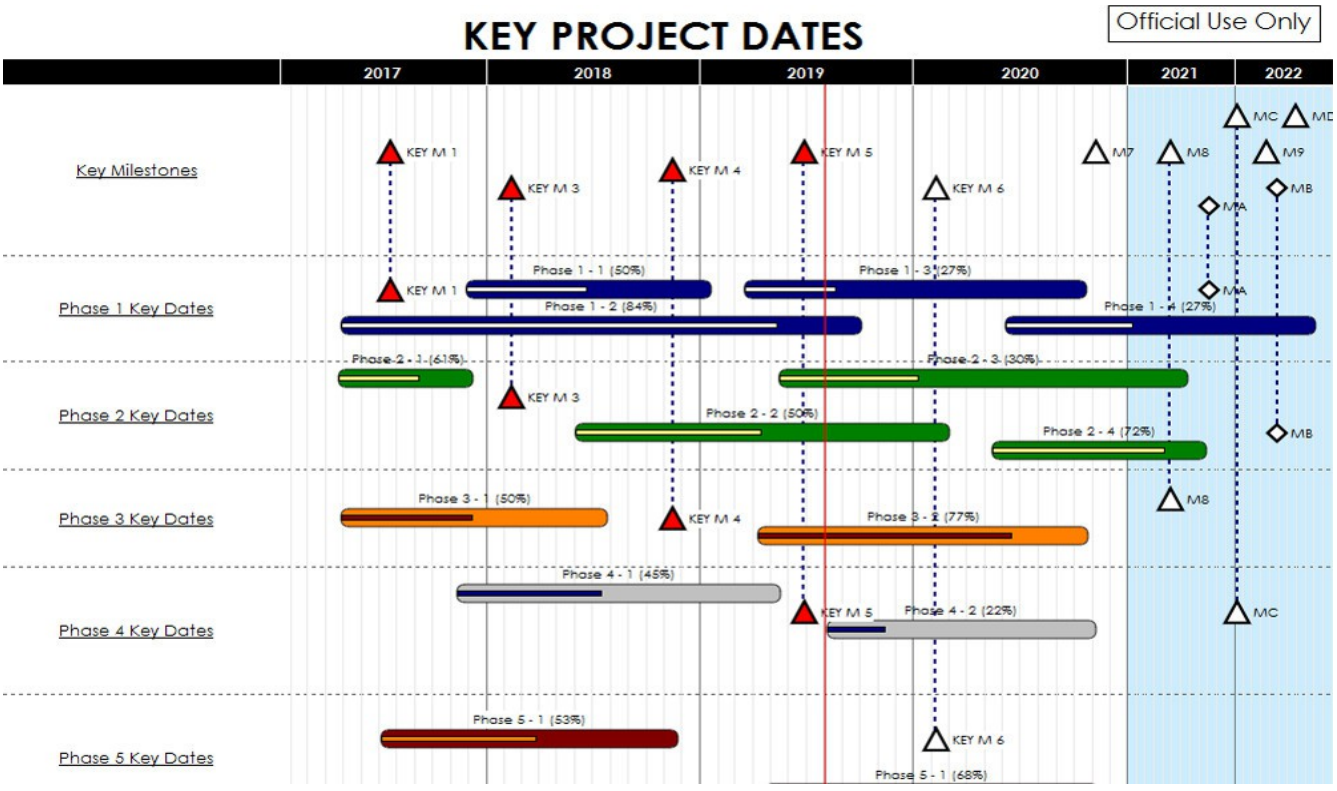


Gantt Chart + Pert dependencies



Gantt Chart


Gantt Chart
Upward Airlines Corporate
Name
Q4 2014
Q1 2015
Q2 2015
Q3 2015
Q4 2015
Dec 2014
Jan 2015
Feb 2015
Mar 2015
Apr 2015
May 2015
Jun 2015
Jul 2015
Aug 2015
Sep 2015
Oct 2015
Nov 2015
Dec 2015
Jan 2016
Redesign Employee Satisfaction Si
Hire survey consultants
Conduct a test project
Interview Consultants
Build new survey questions
Test new survey questions
Review results and make impr
Rollout new survey
Create new Employee Training
Hire consultants
Survey a sample of employees
Compare results
Write curriculum
Test new curriculum with some
Evaluate results of sample gro
Rollout new training
Find new food service provider
Draft RFP
Proposal Period
Release RFP
Downselect to 3 Potential Ven
Analyze final round proposals
Choose new vendor
Airport Traffic
Hire consultants
Analyze current airline traffic
Develop plan to maximize ope
Test new plan
Analyze results of the test
Rollout new plan


KEY PROJECT DATES
Official Use Only
2017
2018
2019
2020
2021
2022
Key Milestones
Phase 1 Key Dates
Phase 2 Key Dates
Phase 3 Key Dates
Phase 4 Key Dates
Phase 5 Key Dates
KEY M 1
KEY M 3
KEY M 4
KEY M 5
KEY M 6
M7
M8
M9
MA
MB
MC
Phase 1 - 1 (50%)
Phase 1 - 2 (84%)
Phase 1 - 3 (27%)
Phase 1 - 4 (27%)
Phase 2 - 1 (61%)
Phase 2 - 2 (50%)
Phase 2 - 3 (30%)
Phase 2 - 4 (72%)
Phase 3 - 1 (50%)
Phase 3 - 2 (77%)
Phase 4 - 1 (45%)
Phase 4 - 2 (22%)
Phase 5 - 1 (53%)
Phase 5 - 1 (68%)

o26 Risk Mgmt p 532
RMMM [nox] == **Risk**
o- **Mitigation** = lessen impact (being proactive)
o- **Monitoring** = Tracking = Monitoring – hold mtgs, update risks
o- **Mgmt** = (create new mitigation tasks) – mitigation means making a bad situation less bad

Reactive vs Proactive
o- **Reactive**: Wait until event (bad thing) occurs, then "think of something"
o- **Proactive**: Think of how to deal with possible upcoming event beforehand
o-- Risk Mitigation Plan + Monitoring (eg, periodic meetings)

Kinds of Risks:
o1. Project Risks: Impact: schedule slip, cost overrun
o-- Pbm Sources: (changes to) budget, schedule, staffing, resources, **reqts**
o2. Tech Risks: Impact: SW quality, on-time delivery
o-- Pbm Sources: design, impl, interfacing to other systems, verification, maintenance, spec ambiguity
(spec == formal written requirements), inferior tools, tool obsolescence,
bleeding edge tech that you decided to use on this project
o3. Biz Risks: Impact: up to stopping project – or even product line
o- Mktg/Marketing risk: cust/user hates it
o- Strategic risk: bad business fit (eg, fratricide – it kills off sales of your older "cash cow")
o- Sales risk: confusing to potential buyers/custs, or confusing to sales representatives (Sales Reps)
o- Mgmt risk: C-level (Company executives level) changes in focus and/or people
o-- Historically, this has been an issue for adopting Agile Dev M.O.
– the point of Cockburn & Highsmith's observation that Politics wins over (strong) People
o- Budget risk: C-level reduces budget; or refuses extra funding to extend project; can't win over customer

Assessing Overall Project Risk
o **Key Questions**, in order of importance – "**No**" answer means it **needs tracking**
o1. Top SW & Cust mgrs formally committed to support the project? (enthusiastic, or at least hopeful)
o2. Users enthusiastic to getting their hands product?
o3. Reqts understood by dev team & custs/users? (usually can't know this till part way into project dev)
o4. Cust/Users involved in reqts dev? – (Actually, users involved in reqts + CRC modeling & hand-sim?)
o5. Users realistic about product? – nothing fanciful is expected?
o6. Scope stable? – Scope == set of Features == set of Requirements → this is very rare
o7. Dev team have req'd skills? – especially if you expect to hire some specialty-skill programmers
o8. Dev team has tech experience in the S/W area (maturity level experience) to be used?
o- (With respect to) extra knowledge not written down
o9. Staff size adequate? -- Key people (specialty S/W areas)?

Risk "Projection" (AKA Risk Estimation)
4 Steps
o1. **Pbb**: Establish pbb scale (Pressman & Maxim rec's 10% increments; but it's far too detailed)
o-- Instead, use tee-shirt sizes for Pbb: small, medium & large – more appropriate for accuracy
o2. **ID consequences** of risk event: What happens?
o3. **Estimate event** cost or **impact**: (see next section)
o4. **Assess estimation accuracy**, if possible (add Error bars – plus/minus one tee-shirt size)
o-- Crowd-source to get more varied viewpoints on the Risk

Risk Scale Class – very rough categorization – These should reflect Mgmt "Concern"
[Pressman & Maxim recommended scale – **nox**]
1=**Catastrophic** – main purpose failure – system basically unusable
2=**Critical** – reduced capability – significant loss of some major functionality
3=**Marginal** – failure of some minor functionality (AKA "nice-to-haves") – painful to users
  o-- EG, like going from laptop word-processor program back to a typewriter – doable, but painful
4=**Negligible** – failure causes **annoying workaround** – all user needs except reduced quality
  o-- EG, like going from automatic zip-code city name entry to looking up the city by zip manually

Assessing Risk Impact
Risk Exposure = RE = **Risk Severity**
RE = Pbb * Cost // Pbb_of_Occur * Cost_of_Event
  o-- **Cost is the same as Impact**

Mgmt Steps:
ID (Identify) Risks
  o1. ID Risk: summary, tag-line, title, index number (#X)
  [o2. ID Risk Scale Class – recommend using names, not numbers (eg, " Marginal") – usually ignore this]
  o3. ID its Pbb: (Lg,Md,Sm) = (3..1) – very rough estimates (only kind you likely can likely do)
  o4. ID its occurrence Impact or Cost: Lg,Md,Sm (3..1) – very rough estimates

Select Risks to Track
  o1. Calc: Severity = **RE = Pbb * Cost** == (9..1) == (LL..MM..SS)
    o-- Is No 5 or 7, prime – only RE is in (9, 6, 4, 3, 2, 1)
    o-- Throw out (3,2,1) cost risks – they are too tiny to care about
    o-- Maybe keep RE=3 → Pbb=1 but Cost=3
  o2. Keep: (9..4) to track 3 values = (9,6,4)
    o-- NB, (9,6,4) aren't hard numbers – R=6 isn't 150% of RE=4 – they are rough relative numbers
    o-- This type of estimate is **very subjective**
  ** On large projs, tracking 40+ risks not unreasonable – typically never track 200+ risks

ID Mitigations, if any
  o- For Risk ID #X
  o- Reduces Pbb?
  o- Reduces Impact?
  o- Benefit of Mitigation – Reduces Severity (RE eqn) to what?
  o- Cost of Mitigation? – is it worth doing?
  o- When can it be applied?
  o- Who needs to be involved in performing the Mitigation?

Mgmt: Track Risks – Periodic Mtgs (eg, weekly, fortnightly, monthly)
  o- Write Risks down
    o-- **Short summary** sheet (one risk per line): ID #X, title, class (Project,Tech,Mgmt), Pbb, Impact, RE, Has-mitigations (Y/N)? Mitigation tasks in progress (Y/N)? – 8 columns
    o-- **Long form** Risk entries in a booklet: add tag-line/summary, mitigations (potential & tasked)
  o- Re-evaluation – Weekly/fortnightly/monthly (hopefully quick – < 30 mins per meeting)
    o-- Risk eval "committee", usually 3-4 people

o-- Existing risk active (tasked) mitigations on track
EX: Contacted backup supplier
EX: Updated As-Built arch doc (eg fancy UML printout) this month
EX: Cross-area training/tasking is up to date – lose the SQL expert, but you have a backup
o-- New risks? – add short line & long page – calc RE – guess mitigation if any
o-- Changed Pbb?
o-- Changed Impact?
o-- Changed Mitigation? (eg, thought up some new (reasonable) mitigations)
EX: Alice (on diff proj) was backup for Bob, but Alice quit – need a new backup
EX: Hiring cost for newbies just lowered because extra training docs available

Risk Figures in Pressman & Maxim ch26
Fig 26.1 **Short summary sheet**, **partial**
o-- **Short summary** sheet (one risk per line): ID #X, **title**, **class**, **Pbb**, **Impact**, RE, Has-mitigations (Y/N)? Mitigation tasks in progress (Y/N)? – 8 columns
Missing
o-- ID#, risk index number let's you find the **long-form** (eg paragraph) **risk page** quickly
Replace "RMMM" column with these
o-- Has-mitigations (Y/N)?
o-- Mitigation tasks in progress (Y/N)?
Silly
o-- Probability too precise but inaccurate → Use T-shirt sizes instead: S,M,L (or Low, Medium, High)
o-- Impact ditto → skip 1-to-4 "new" terminology & use T-shirt sizes instead: S,M,L

FIGURE 26.1 **Sample risk table prior to sorting**

Risk	Category	Probability	Impact
Size estimate may be significantly low	PS	60%	2
Larger number of users than planned	PS	30%	3
Less reuse than planned	PS	70%	2
End users resist system	BU	40%	3
Delivery deadline will be tightened	BU	50%	2
Funding will be lost	CU	40%	1
Customer will change requirements	PS	80%	2
Technology will not meet exceptions	TR	30%	1
Lack of training on tools	DE	80%	3
Staff inexperienced	ST	30%	2
Staff turnover will be high	ST	60%	2

Skip Fig 26.2 3D cartesian view of RE surfaces – including Pbb, Cost, 3-rd axis = Mgmt "Concern"
o-- So, **Mgmt "Concern" is irrelevant** for RE – s/b folded into Risk Scale Class choice

Fig 26.3 **Long form Risk page** – **looks okay**, maybe too formal (rare on most med/large/XL projects)
o-- The fancy details are for contractual "work product" documents to customer (not to users)
eg, to a gov't agency

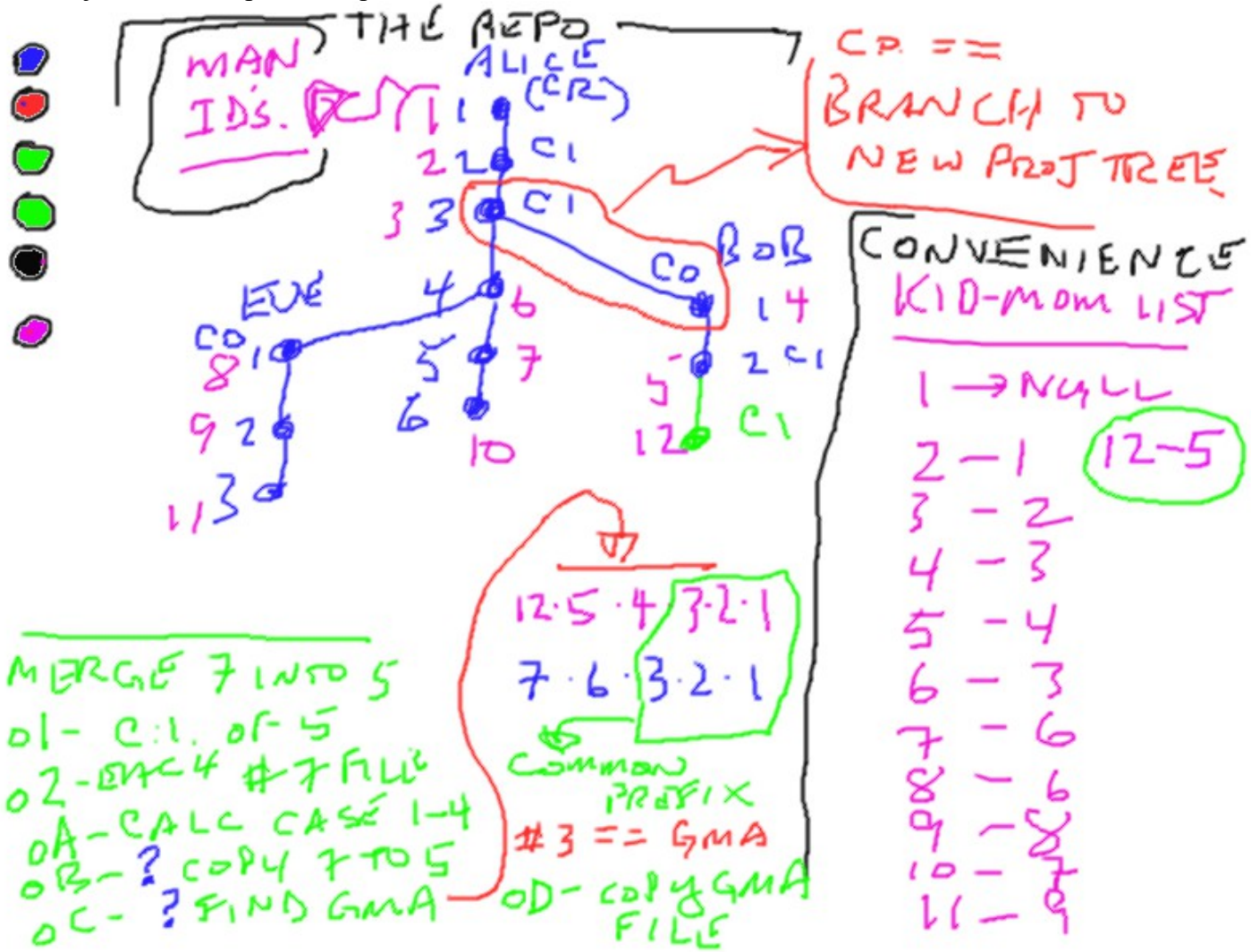
Functional Programming

- o- Fcn always gives **same answer for same inputs** – like adding or multiplying
- o- **Assign only once** – value to variable
  - o-- Need more values, create more variables
  - o-- (In compiler tech: called SSA (Static Single Assignment) – allows simpler compiler algos)
- o- Use **immutable structures** – ie, create a copy instead of modifying them
  - o-- Means a var always has the same value, even if it is an array or a linked list
  - Caveat: An **agent** (AKA object) is typically not immutable – so think of it as an Agent

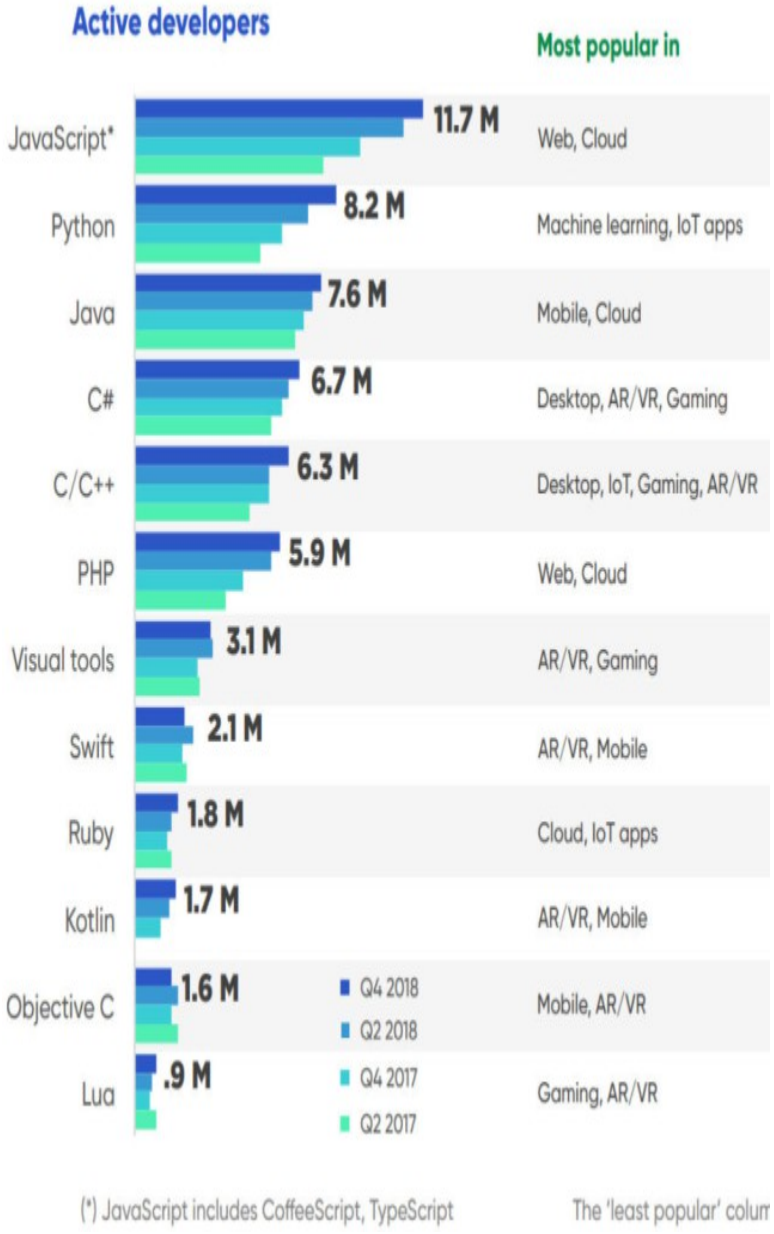
Best API is a Buffer, not a Call

Lab

*** Project #3 due 11pm Sun April 25th



Pgmg Lang "Popularity" – Slashdata vs Tiobe "Index"



Programming Language	Ratings	Change
C	14.32%	-2.40%
Java	11.23%	-5.49%
Python	11.03%	+1.72%
C++	7.14%	+0.36%
C#	4.91%	+0.16%
Visual Basic	4.55%	-0.18%
JavaScript	2.44%	+0.06%
Assembly language	2.32%	+1.16%
PHP	1.84%	-0.54%
SQL	1.83%	-0.34%

Q: How to reduce project failures?

Idea:

-- **Brooks** (Turing) **says** Iterative Incremental Delivery wins (30yrs ago)
-- **Weinberg says** very short (**half-day**) iterations, with **demo** (54yrs ago) – Project Mercury S/W
  o- planning and writing **tests before** each micro-increment
-- **Agile says** Mini-Incremental Dev & Delivery (weeks at most)
  o- TDD = write **tests before** design/code
-- **DevOps CI says** Daily/Continuous build+regression_test of every "completed" checkin
-- **Rule #2 (Hunt) says** small Add-a-Trick & see visible success
  o- Rule #4 (EIO) says write **tests before** design/code
-- **Smart Std says** to always have **visible progress** to show, **daily**
-- **Rule #5 (Half-Day) says** plan in **visible half-day** tasks

Get the Picture? --> Stop writing large/medium chunks of untested code

Q: How to reduce program complexity? --------

Functional Programming (FP)

(Amusing side-node: John Backus (led Fortran 1957, also has 11-th Turing) invented his own FP, called 'FP')

o1- Function always gives **same answer for same inputs** – like adding or multiplying
  o-- It's FP if you can replace the calculations with a lookup table (args → answer), always the same
  Caveat: **In real life**, nearly all SW requires (**args + state→ answer**), nearly always different
    o--- state can be local, global, or both – local is simpler than global

o2- **Assign only once** – value to variable
  o-- Need more values, **create more variables**
  o-- (In compiler tech: this is called SSA (Static Single Assignment) – allows simpler compiler algos)

o3- Use **immutable structures** – ie, create a copy instead of modifying them
  o-- Means a var always has the same value, even if it is an array or a linked list or a "structure"
  o-- EG, so that you can be handed an array ref, and change slot [6], and the owner cannot see the change
  Caveat: An **agent** (AKA object) is typically not immutable (not FP) – so think of OOP as with Agents
    o--- But that's okay, cuz so are DBs and other Data-containing objects – they just add more complexity
  o-- **Immutables are Slower** to use – but We Don't Care
      → Rely on **Rule #1 (Optim)** – Be slow, until some **tiny code spot** is **Proven** to need more speed

Q: What langs are FP?

A: **Style:** All of them. You just avoid changing "in place", but make copies instead.
A: **Features:** Some default to immutable, and force you to use special "Dirty" Features for side-effects.
  o- Fast Immutable – Uses "**Persistent Data Structures**" under the hood, in the run-time mech.

Bugs Still Remaining even when you use FP:

o- **In**: Did you call it with the correct args?
o- **Mid**: Did you test it enough to be sure the Fcn calcs what you wanted?
o- **Out**: Did you get the result you were expecting for those args? A second time, too? Repeatedly?
(*) Q: But Side-Effects are what Agents's do, so **why use FP?**
  A: **To simplify 95%** of the architecture & code.

Short Fcns

o- One of the biggest benefits of writing short fcns.
  --> There is a **tendency to test** a short fcn right after it is written.
  --> This gets you closer to doing Rule #2 (No Bug Hunts), Add-a-Trick
o- Another big benefit – All the code is on the screen at one time → **eyeball** moving is faster than scrolling
  o-- plus, you don't "lose your eyeball place" while scrolling → worth 2x in productivity

More – Q: How to reduce program complexity? --------

Q: Another way to look at improving SWE success?

failure ← complexity ← info-convolving ← (#paths * their "control" variable's values → cause switching)

o- Recall **Brooks' – The Essence of Invisibility** → the architecture is too complex to "see" all at once

o-- Why important? → "see all at once" only involves eyeball movement → easier to understand it all

o-- Why even that is not enough?

→ **info-convolution** (mingling side-effects) means the arch dynamics are still too hard to follow

→ like looking at a very busy street scene and understanding what everyone is doing at the same time

o- So, How to **reduce the dynamics complexity**?

→ make the boxes/**agents simple** (and FP if possible, or some of their ops FP)

→ make each box/agent only handle **one (simple-ish) concept** (including its internal state)

o-- It can do multiple closely-related (same-concept) operations with-respect-to that data

→ try to avoid having agents talk to each other – but using buffers is more complicated

Q: What archs/patterns support more info-hiding/lower-coupling?

SWE Ease of Modding = Ease of Understanding = Reduced Complexity

o- Use SOLID/D's DinV-DinJ cutouts, to disconnect client-helper linkages

o-- Need DinJ injection mgrs

o- Use **Brokers/Mediators/.../Buffers** to disconnect direct-calls between Clients and Helpers

o-- Broker or Mediator's concept is knowing & managing **the "coupling mechanism", or "helper order"**

o-- **Buffers** can do the same kind direct-call separation but are **'dead' data**

– Only data format is known -- minimal "format" (lesser concept than an API)

– but they likely need 1+ binary flags (eg, new-data, sync flags, etc) – more complex – ? do it in FP ?

– goes under the "Reader-Writer problem"

o- Use **Entity-System IDs**, to avoid hierarchies – each entity has **multiple features**, each with a **behavior**

o-- Runtime behavioral bucket lists easily alterable

o- Use **Helper Wrappers** (eg, GoF **Strategy**), to reduce client-helper knowledge

o-- Helper wrappers hide more complex behavior

o- Use **Declarative style**, to avoid sequencing steps (AKA timing issues)

o-- Things may/must/will/need happen, but **don't care about when**

– but it's possible to create timing effects in declarative stuff – eg, with FSM state, or "current-step" var

Q: How does "virtual method" overriding work?

o Mom-Hatting – Behind the Run-time Curtain

The **Liskov** Substitution Principle (LSP) in Action -- from SOLID/D

o- Assume class Chris inherits from class Bob which inherits from class Alice. **(pic has Chris.Aleph wrong)**

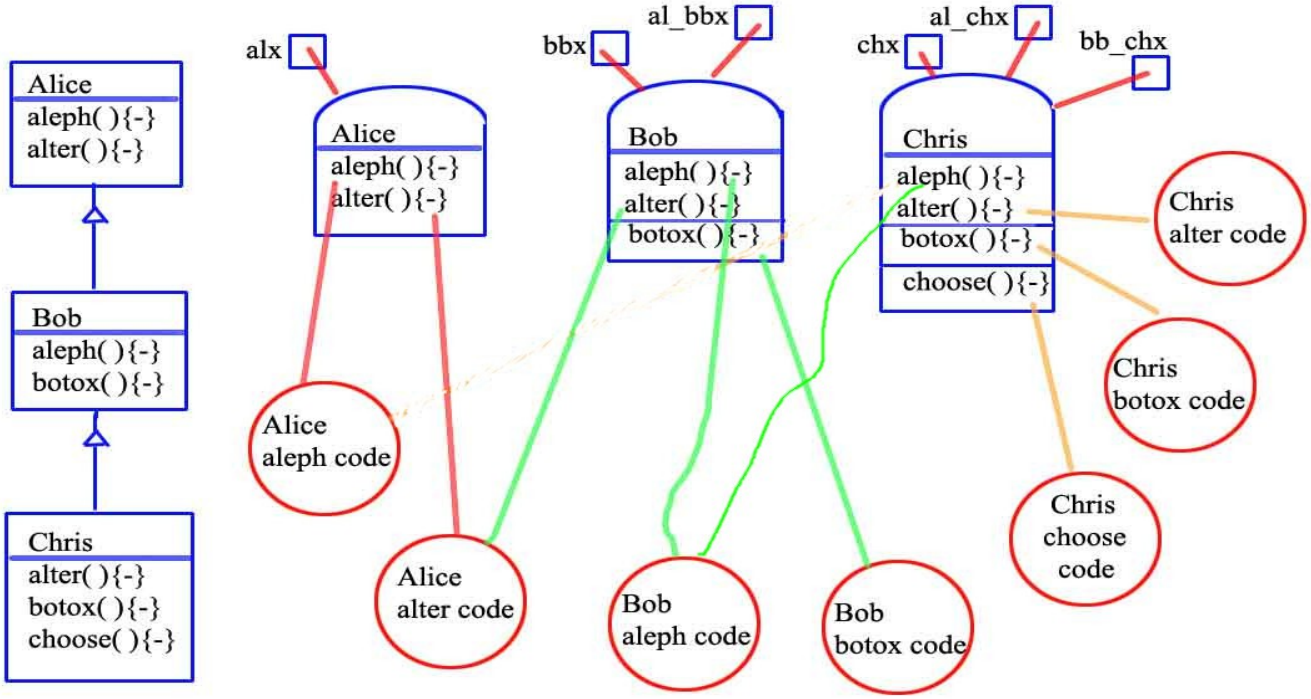
o- Class Alice defines an **aleph()** method and its kid class Bob overrides **aleph()**.

o- Alice defines a **alter()** method and its grand-kid class Chris overrides **alter()**.

o- Bob defines a **botox()** method and its kid class Chris overrides **botox()**.

o- Here are the key local object variables. o- All methods are declared public.

```
Alice alx = new Alice();        Alice al_bbx = bbx; // bbx upcast to Alice.
Bob   bbx = new Bob();          Alice al_chx = chx; // chx upcast to Alice.
Chris chx = new Chris();        Bob   bb_chx = chx; // chx upcast to Bob.
```



For each of the following method calls, **which class's method code gets called? (or a compile-time error)**

a. bbx.alter()	**Alice's**	Bob's	Chris's	Error
b. chx.alter()	Alice's	Bob's	**Chris's**	Error
c. al_bbx.alter()	**Alice's**	Bob's	Chris's	Error
d. bb_chx.alter()	Alice's	Bob's	**Chris's**	Error
e. alx.botox()	Alice's	Bob's	Chris's	**Error**
f. chx.botox()	Alice's	Bob's	**Chris's**	Error
g. al_bbx.botox()	Alice's	Bob's	Chris's	**Error**
h. bb_chx.botox()	Alice's	Bob's	**Chris's**	Error

Caveats: this doesn't use the "**Vtable**" memory-saving hack (that is always used in compiler run-times)

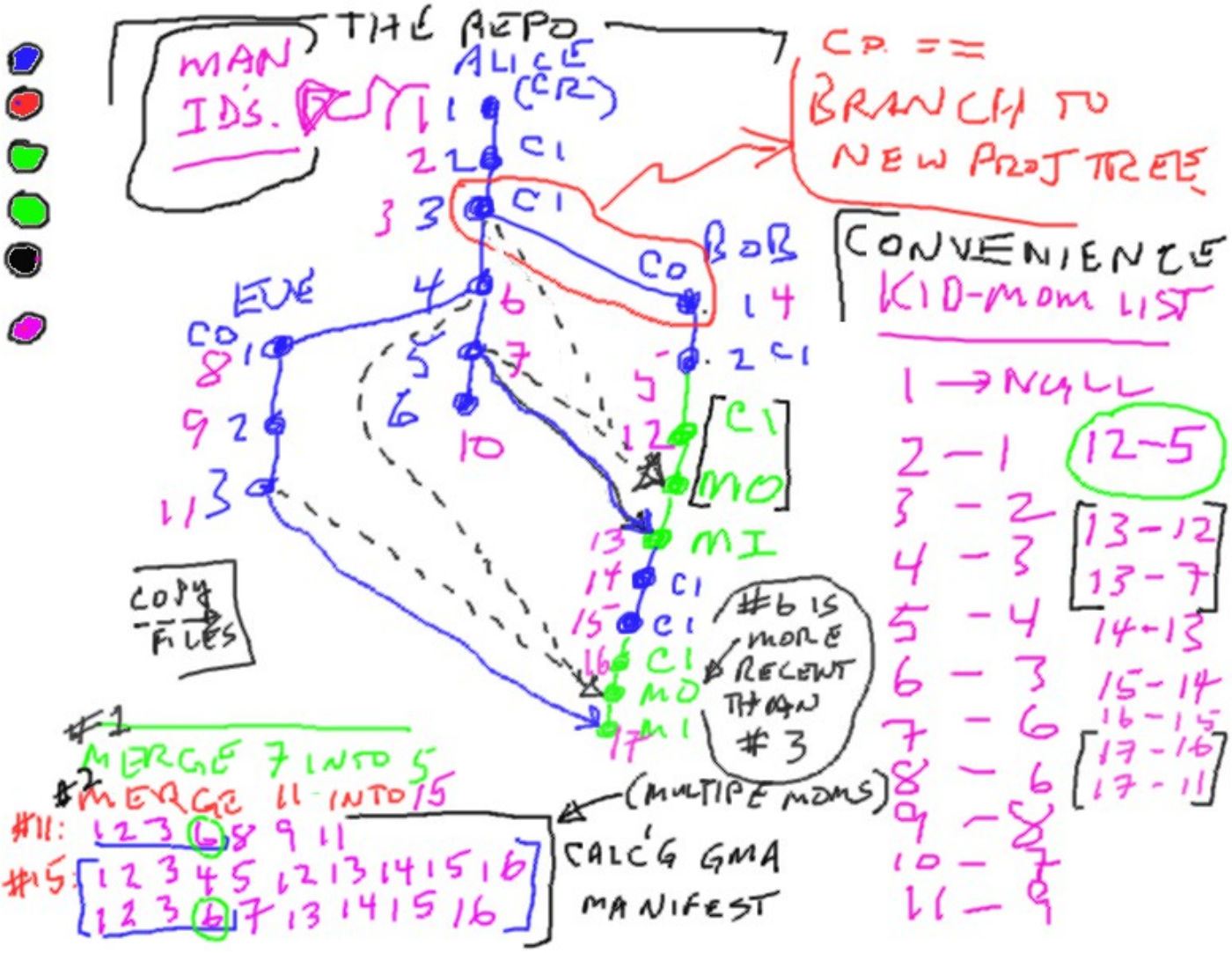
o- Also, while method refs aren't in an object (cuz vtable), the data slots ARE, and they use "class frames", too

RT memory areas: Global constants, Global vars, Code Space, RT Call Stack, RT Heap (for New/Malloc stuff)

Lab

Q: What happens when you merge? A: You get another "kid-with-two-moms" in your Repo DAG.

(*) This complicates finding the most recent common ancestor, cuz of (maybe) multiple root-kid paths.



SOLID/D – LSP – (Barbara) Liskov's Substitution Principle
(**Turing Award 2008 –** For contributions to practical and theoretical foundations of
programming language and system design,
especially related to data abstraction, fault tolerance, and distributed computing)
o- "When a Kid (ref) wears a Mom-hat, it must behave like Mom."

LSP Contract Rules – But why bother with Rules? What could go wrong?
(*) Key: Don't Burn Mom's Friends
o1. Kid's override method is going to be called by Mom's Friends
o2. Mom's method can call Mom's Friends, and they expect Mom's method will work correctly for them
o-- They were calling Mom's method long before the Kid was born\
o3. Treat the Mom method as the Target and the Kid's Override Md as a before-and/or-after Decorator

For Override Methods (in Kid/Descendant Classes)
1. Preconditions cannot be strengthened
Preconditions → arg value checking
o- Mom method has restrictions on what it allows
o-- Can't Return Early just cuz Kid method thinks more processing is unneeded – cuz Mom's method didn't
o- Always – Mom method must be called -- Behave like Mom
What about Kid override method's preconditions?
Q: Can Kid method return early when Mom wouldn't?
A: No, cuz Kid method might be called by Mom/Friends who expect results (eg, to be in Mom's slots?)

(*) Key to Kid method is to **call Mom** – so Kid code can be run **before and/or after** calling Mom's method –
AND Kid method cannot mod any Mom slots

2. Post-conditions cannot be weakened
Post-conditions → return value restrictions
o- Mom method callers might rely on Mom's return value restrictions
EX: int return value is shipping cost: always positive cost
But you subclass to create Free Shipping & return 0 cost
o- Especially, return is an object of a new (sub) class – be very careful here
o- Kid wearing Mom Hat can't relax these, Mom's Friends might choke
(*) Unless: returning a subclass that does LSP correctly, too
What about Kid method's post-conditions?
Q: Can Kid method return a value Mom wouldn't (EX: ==0 when Mom returns >0)
A: No, cuz Kid method might be called by Mom's Friends, They could choke

3. During Processing – Exceptions cannot be strengthened or weakened
Q: Can Kid method throw new Exception class obj?
Q: Can Kid method handle some Mom Exception, for some new reason?
A: No to both -- not Mom behavior

4. Code Invariants must be maintained
Invariants --> stuff Mom & Friends thinks are true before & after calling Mom's (Overridden??) method
o- So, let Mom's method modify all old Mom slots & globals that Mom modifies/uses
o-- Don't modify Mom's slots from the Override method – Mom & Friends might choke on it
(***) Helpful**: Don't allow Mom slots to be directly accessed → use Mom methods instead

GRASP SWE Acronym (OOAD)

General Responsibility Assignment Software Principles – Assigning Responsibilities to Classes

General Software Principles for Assigning Responsibilities to Classes/Objects
AKA Who Does/Knows What?
[Somewhat backwards – you should Create an agent because it will be responsible for doing something
- o- Either managing a collection of related data parts
- o- Or a collection of related actions
- o- Or both
- o- And via CRC Cards, linkages, and hand-sim]

(*) Doing comes from UCs, and their helper agents
(*) Knowing is what data an Action uses/needs/creates, etc.
6 kinds of GRASP objects/agents/classes
- o- **Controller, Creator, Info Expert, Pure Fabrication, Indirection, Protected Variations**

Controller Agent/Obj

What does it do?
- o- "Owns" knowledge of multiple helpers/agents
- o- **Coordinates sub-tasks** and diverse objects/agents
  - o-- Esp synchronization between objects/agents
- o- **Gathers** resources (eg via helper) & **establishes** connections (eg via helper) to get a task done
  - o-- Incl creating objects & passing them around – ie, D2=Dependency Injection (SOLID/D)
    - o--- But it would use a **Creator** agent to get this done
- o- **Delegates** rather than does detail work

When to use?
- o- To Control/Mgr overall "system" – eg, a "Main" manager
  - o-- AKA **GOF Mediator**
- o- To Control/Mgr access to external device/system/component
  - o-- AKA **GOF Proxy**
- o- To Control/Mgr a Use Case's behavior – which is made up of 3-8 steps
  - o-- AKA **GOF Mediator**
- o- To Control/Mgr a Session of behavior (recall: diff between "Session" and "REST" style of doing things)
  - o-- AKA **GOF Proxy** – some kind of proxy helper

Creator Agent/Obj – Building objects and handing out refs to them

Q: Who should be **responsible for creating** a class obj/agent, named B? – **Dependency Injection** (SOLID/D)
A: GoF Mediator between (helper) agent B and Callers of B (to this extent, like a Controller)
- o-- Use Cutout & Dependency Injection – to inject a ref to helper B into caller C.

A: Container/agent, named C, that agent B is a part of – but maybe use GoF Strategy, for more flexibility
A: GoF Factory pattern (build objects on request) is sometimes used for this

Info Expert – (in our terminology, an Agent)

Q: Who (in Pbm Domain) should be assigned a responsibility for some batch of specialized knowledge?
- o- Batch of specialized knowledge == Major data part – to be hidden inside its agent handler

A: Assign responsibility to Domain Expert agent with the specialized Info (about that data, and its handling)
- o- Expert may delegate to a helper agent who knows part of the problem

NB, Expert has Pbm-Domain level specialized knowledge
Not at CompSci level – cuz that is called a "Pure-Fabrication"

Pure Fabrication Agent/Obj

o- Some stuff needs to be done, but when you look around, nobody seems to be responsible for it
o- Means you are inventing an agent/obj that doesn't exist at the Pbm-Domain level
Hence the title "**Pure Fabrication**"

Q: Responsibility & related info doesn't well-match any existing Info Expert (per the users)?
o- Some batch of coherent knowledge has no Info Expert at the Pbm-Domain level (according to the users)
o- Or, the coherent info isn't at the Pbm-Domain level, but at the CompSci level

A: Create special convenience Pure-Fab Expert agent/obj/class for it
And be on the lookout to merge the Pure-Fab class into a better already-existing class
Because it is often the case that merely creating the Pure-Fab agent will cause you to notice that what that agent handles is already almost being done by some existing agent

EX: RDB manager – ORM agent (Object-Relational Mapper – object inter-linkages to RDB tables)
A CompSci level example

Indirection Agent/Obj – we need a cutout agent (who keeps client and helper ignorant of each other)

(*) To lower coupling

Q: How to avoid Client know details about a Service Provider (AKA helper)?

A: Use Indirection – via an Intermediary and an Interface
o-- use a Cutout pattern – Dependency Inversion – with Dependency Injection
o-- **GOF** patterns **Adapter, Bridge, Facade, Observer, Mediator, Proxy, Strategy**

EX: Mediator (ORM) OO-RDB-Mapper
o-- Knows how to translate OO to RDB & back

Protected Variations Agent/Obj – involves predicting the future needs – and these are VERY expected

(*) Hide expected changes behind a wrapper so that the rest of the system need not change, too

Q: How to protect against **expected changes** to a helper agent/algo/class/subsystem?
(But first remember Rule #1 – Never Pre-optimize)

o1- as **Provider/Helper**: Wrap access to helper with a small API (hence, low coupling)
o-- and try to use simple polymorphic-friendly method signatures – if it makes sense
o-- Consider **GOF Strategy** pattern – specifically designed for this kind of thing
o- And it also allows plug-replacing agents at run-time (via Dependency Injection)
o2- as **Client/Caller**: Ensure all calls out are to Interface (mom-hatted) objs
o3- **Extreme situations**: Hide the helper behind a Controller/Mediator

Lab
Arch possibilities
Q1 How would check-in #4 know that it was checked-out from manifest #2?

Kid-to-mom file (trick)
On a manifest-creating command, create a kid-to-mom line item in a kid-to-mom file.
- o- Command has a source and a target

Create-Repo source project-tree and a target repo
  Kid = CR manifest; Mom = null; Tree= project-tree → all info contained in CR command
Checked-out source manifest and a target project-tree
  Kid = CO manifest; Mom = source manifest; Tree= new-project-tree → all info contained in CO command
Checked-in source project-tree and a target repo
  Kid = CI manifest; Mom = manifest **for most recent same Tree**; Tree= project-tree
Q: How to get it?
A1: It would be nice if we could see that manifest in the project-tree, top folder (s/b a dot-file)
  Q: How to get the project-tree manifests into the project-tree, top folder
  A: Put a copy of the command manifest in the project-tree's top folder – for every mani-creating command
A2: It would be nice if we could see that manifest name in a file in the project-tree, top folder
  Q: How to get the project-tree manifest names into a file in the project-tree, top folder
  A2a: Add the manifest name to the mani-list file (s/b a dot-file) for that project-tree,
    and the file resides in the project-tree, top folder
  A2b: Create/overwrite the latest-mani-name file (s/b a dot-file) for that project-tree, with that manifest name
- o- One mani-name in one std file (eg, latest-mani-name file) at top-level folder in its project-tree

checkout tree -- git branch

Recap
-- Brooks (Turing) says Iterative Incremental Delivery (IID) wins (**30yrs** ago)
-- Weinberg says very **short (half-day) iterations, with demo** (**54yrs** ago)
o- planning and writing tests before each micro-increment (= EIO)
-- **Agile says** Mini-Incremental Dev & Delivery (20yrs ago) – (25yrs for XP & Scrum)
o- TDD = write tests before design/code (= EIO)
-- DevOps CI says Daily/Continous build+regression_test of every (micro-slice) checkin
-- Rule #2 (Hunt) says small Add-a-Trick & see visible success (focus on zero run-time bugs)
o- Rule #4 (EIO) says write tests before design/code (helps focus design & avoid gold-plating)
-- Smart Person Std says to always have visible progress to show (your manager), daily
-- Rule #5 (Half-Day) says plan in visible half-day tasks
Get the Picture? --> **Stop writing large/medium/small chunks of untested code – tiny = micro-slice**
Upshot: Throw out Predictions and do day-to-day/week-to-week deliveries & hope users like it
o- This is what Agile recommends
o- Try to focus on working at a **"sustainable" pace** (non-death-march)
o- Means much less at risk at the "silver bullet" point in the project (AKA Validate the model)

Pbm
o- Make program that **works adequately** (ie, V&V is successful)
o-- Solid understanding of the **users' problem** – all important mis-communications resolved
o-- Solid understanding of how to achieve the kind of **UI the users will like**
o-- Solid understanding how to achieve the **quality the users will like**

o- Make program that that is very **easy to understand** –during new-dev and for upgrades/bug-fixes
o-- Q: Will the **As-Built docs** be available? // "As-Built" means after its built, you document it correctly
o-- Q: Will the docs be **written for newbies**?

o- Make program with group of pgmrs, **as a "team"** – cuz it's (almost) always too big for one pgmr
o-- All working together, pulling in the same direction (toward project success)

o- Make program inside the **time-frame**, and under the **budget**, available
o-- Adequate **prediction of effort**, plus adequate "contingency padding" (extra budget money)

State of the Art
o- Numerous **SWE "Quality" Groups** now exist – founded to find & tell how to "solve the SWE pbm"
o-- **SEI** founded in 1984 – **37 years** ago – spent **> $1Billion** on SWE research since then
o- Parnas/Lawford: **30+ years SWE research hasn't helped** [Parnas 2003] (ie, from1968 to 2003)
Q: Why 1968?
o-- **"SWE" coined 1968** (at NATO conf) → after a decade-plus of failed projects – big-gov big-bang
Upshot: S/W Engineering is still a black art

Pitfalls
o- **Complexity** – (Brooks has this as his focus)
o-- Scales **exponentially** – and **unavoidably** – both statically **and (worse) dynamically**
o--- And you can **NOT** test all possible static paths in the code – and we're not counting all dynamic paths
o-- Model/Arch **adequate** but **simple to understand**
o-- CS-level Arch ditto – low-coupling and high-cohesion
o- **Mgrs poor** (mostly due to lack of knowledge, preparation, tracking, and "light touch" control)

o-- Dev'r **morale** → **10x productivity** at risk (from very bad thru poor to very good)
o-- Arrange for adequate/effective **comm w users**
o-- **Looming risk** events – proactively ID, eval impact, and track risks
o-- **Gel group** into a team, and maintain the team

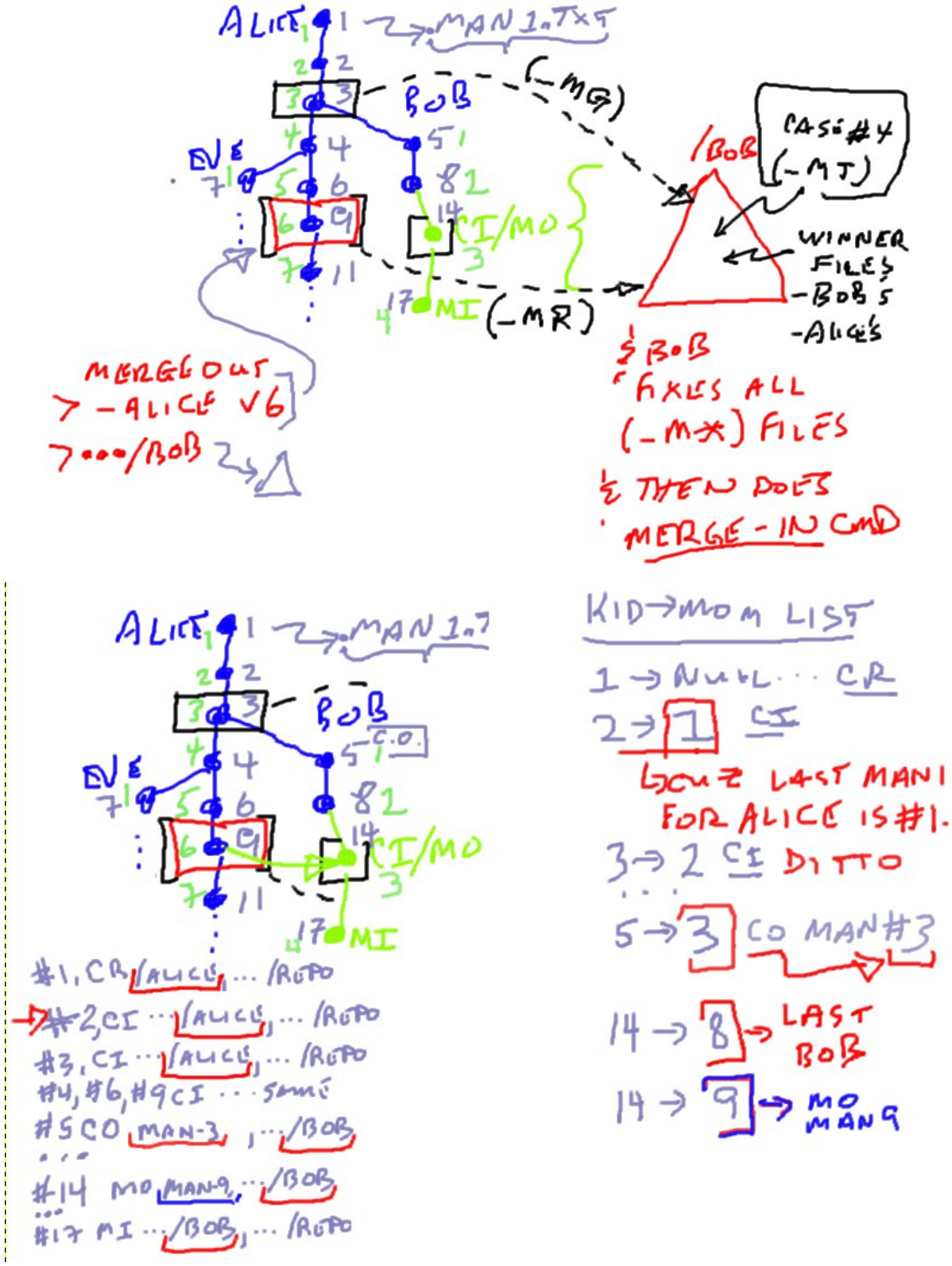
o- **Comm poor** with "users" – reqts result in building an ill-fitting, unsuitable product
o-- ID-ing who they (users) are
o-- Adequate comm time
o-- Talking in the user's (Problem Domain) language – to help uncover mis-understandings
o-- Manage their expectations to maintain their morale during comm

o- **Prediction Bad** of "The Plan" (Features/Effort/Timeline – AKA the "Project Triangle")
o-- "Prediction is hard, especially about the future" – Yogi Berra
o-- Manage customer expectations, avoid "bad news" surprises
o-- Avoiding "error bars" = avoiding any show of uncertainty about completion & delivery & quality

Soln: No Perfect Sol'n
o- No perfect way to avoid **Reqts changes** (users never get everything correct up front)
o- Brooks' **No Silver Bullet** – and that's just for Comm+Model correctness – **Model-phase V&V**
o- Hard to guarantee Model will have **required FURPS** (although Functionality is easiest to be confident of)
o- No perfect way to **avoid RT bugs** – so, must include "**contingency time**" to handle a "large pile" of bugs
o- No perfect way to guarantee major **integration** won't reveal major **design flaws**
o-- Large **rewrite delays are expensive** and typically introduce another "pile" of bugs
o- No way to **understand** entire program in **adequate detail** – Brooks' **Invisibility** pbm
o- No way to test all pgm interactions – the static ones are all-paths, the dynamic ones include global state
o-- Means there can be bugs lurking in untested pathways through the code

Workarounds:
Complexity –
o- Avoid building new S/W → **COTS** (where feasible)
o- Avoid major RT bugs → Rule #2 (Hunt) via **Add-a-Trick**, and build/demo in **micro-slices**
o- Avoid major integration by doing **mini-slice integration** → IID (eg, Agile style), or "**daily-build**"/CI style
o-- **One-Day Tasks** = all tasks (leading to completed testable micro-increment) take less than one day
o- **Separation of Concerns**
o-- **Separate agents** → low-coupling
o-- **SRP** in SOLID/D → high-cohesion → agent handling a concept only manages that concept, ops + data
o-- **FP** agents (as much as possible) → same inputs give same outputs (eg, like 3+4 always equals 7)
o- **Reduce complexity** in each function you build – eg, McCabe's Cyclomatic – or UC's "3-8 steps"
o-- IE, very few executable code statements (as opposed to declarative var stmts) – easy to understand
o-- Whole function fits on screen (without pan & zoom) – eyeballs are faster than pan/zoom
o--- To keep it in your head (all at once), you need to constantly refresh parts – eyeballs are fastest at this

Lab – manifest kid-to-mom parentage list – how is it created?



S/W State of the Art – Workarounds:

Complexity (Continued) –

- o- Avoid building new S/W → **COTS** (where feasible)
- o- Avoid major RT bugs → Rule #2 (Hunt) via **Add-a-Trick**, and build/demo in **micro-slices**
- o- Avoid major integration by doing **mini-slice integration** → IID (eg, Agile style), or "**daily-build**"/CI style
  - o-- **One-Day Tasks** = all tasks (leading to completed testable micro-increment) take less than one day
- o- **Separation of Concerns**
  - o-- **Separate agents** → low-coupling
  - o-- **SRP** in SOLID/D → high-cohesion → agent handling a concept only manages that concept, ops + data
  - o-- Functions give lower complexity than OOP classes
  - o-- **FP** agents (as much as possible) → same inputs give same outputs (eg, like 3+4 always equals 7)
- o- **Reduce complexity** in each function you build – eg, McCabe's Cyclomatic – or UC's "3-8 steps"
  - o-- IE, very few executable code statements (as opposed to declarative var stmts) – easy to understand
  - o-- Whole function fits on screen (without pan & zoom) – eyeballs are faster than pan/zoom
    - o--- To keep it in your head (all at once), you need to constantly refresh parts – eyeballs are fastest at this
- o- **Write code that is so simple** that jr pgmrs can understand your code
  - o-- NB, Functions are SRP, Objects are typically not SRP – so use Agents for big things – and lots of fcns
  - o-- **Avoid class hierarchies** whenever possible – they lead to Liskov bugs and OCP bugs, & low maintainab
  - o-- **Abstraction trap** – good useful abstractions are simple obvious proven concepts
    - – the original OOP point was for **reuse**, which almost always fails – requiring 3x-6x extra work to do
    - – so don't invent your own abstractions (as classes), but wait till you've built 3+ of em
      (3+ data pts to provide solid foundation for extrapolation to a good reusable abstraction)
  - o-- Avoid fancy complicated OOP class stuff – fcns are much simpler – less bloat, less complexity
  - o-- **Don't use globals** (or more than a tiny few, if pressed)
    - – instead **Pass needed data as arguments** and Return any needed data changes to the caller
    - – and each **Agent holds a slice of global state/**data; they OWN it so only they mod it – no getters/setters
    - – the only globals you typically need are agents (except when starting with Rule #0, Fast)
  - o-- Try to **avoid passing around "object references/pointers"** – (exception, D2=dep injection)
    - – this changes the distributed architecture couplings dynamically, exciting (in a bad way)
    - – if you must, consider brokering/mediating to hide the knowledge of who currently "knows" who

Sidebar:

Single Dynamic Dispatch vs Multi-Dispatch vs Polymorphism

(Single) **Dynamic Dispatch** – pick fcn body based on 1 arg data type, the 1-st arg (to left of fcn name)

- o- EX: foo.doit(...) // conceptually: call "doit" fcn with foo (ref/ptr) as the first argument
  // Also, the doit body of code is specific to the foo class (if it is an override method)
- o- We're selecting the correct fcn body based on the fcn's name and the 1-st argument's data type (it's class)

Multi-Dispatch – pick fcn body based on 2+ arg data types

- o- EX: doit(fredx, bobx, …) // call "doit" fcn with fredx "object" + bobx "object" as 1-st two args.
  // Also, the doit body of code is specific to the fred class and the bob class (& **check class hier** for body)
- o- EX Alt lang syntax: fredx@bobx.doit(…)
- o- EX – see GoF "Double-Dispatch" design pattern

Polymorphism – pick body for a fcn's name together with **all arg data types**

- o- EX – int xx; float zz; … xx + zz; // we use the "+" fcn body that converts xx to float & does a float add
- o- EX Alt syntax: add(xx, zz); // ditto
- o- EX Alt syntax: +(xx, zz); // ditto, if we allow more punctuation chars to be identifier "letters"
- o- To use this **args-choose-the-most-specific-body** for a fcn name (eg, "doit(-)") // check datatype hier
- o- EX: print_me(…); // but in Java, there is a generic print fcn that shows an object's guts, so override it

Sidebar (Redux):

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)

o- Max fcn size = 50 LOC, 3-8ish main lines (mostly decls, empty/brace lines, error handling)

XS = 100 LOC, 2 pgs (AKA Tiny)

SM = 500 LOC, 10 pgs – begins to be affected by poor S/W dev practices (incl mgrs)

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

Complexity (Continued) –

o-- Use **D2 (Dependency Injection)** to provide clients with helpers, or maybe even comm via buffers

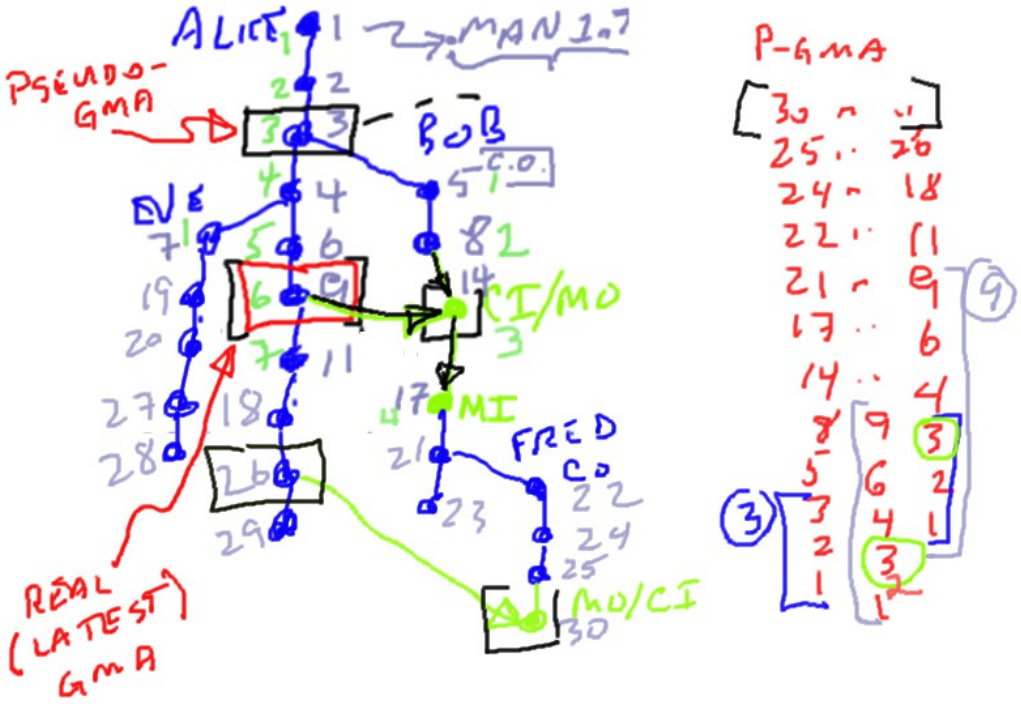
o-- **Never use getters/setters** cuz it means the object doesn't OWN its data

– instead build a structure/struct/record (AKA an object with slots but no methods of any kind)

– The reason for having "objects" is information hiding – if you're not hiding, don't use an object

Lab

Finding Grandma in a Repo with prior merges (ie, the manifest graph is a DAG, not just a Tree)



Quiz #4 Answers

. As discussed in lecture, there are three key features that are normally supported in a functional programming language. Very briefly, what are they?

A: 1) Same answer for same inputs // 4pts
2) SSA: assign value to var **only once** // lessen state confusion
3) Immutable structures: always modify a copy // outsiders unaffected

2pt = "higher level functions" // eg map reduce apply curry – produce fcns or take fcns as args
0pt = "value to variable"
0pt = "Lazy evaluation" – Don't do the work until somebody asks for it (EX, JIT = just-in-time compilation)
0pt = "Supports nested functions" – define a "local" fcn, **only callable inside its mom-fcn (scoping)**
0pt = "Efficiency" – usually FP is slower (cuz of copying before modifying a single item slot)
0pt = "Inheritance" – FP doesn't use classes, or dynamic hierarchy linkage (eg, in JS) for dynamic dispatch
0pt = "Abstraction" – doesn't do this
0pt = "Encapsulation" – doesn't do this
0pt = "Designed on the concept of mathematical functions"

map = input is a fcn & a data set X → output is a data set Y where each Y.J item = fcn(X.J)
reduce = input is a fcn & a data set X → output is fcn(X) – like add(X) → sum of X items
apply = input is a fcn & a data item Z → output is fcn(Z)
apply(fcn, Z) → fcn(Z)
curry == curry(fcn1(A, B, C)) → fcn2(B, C) where fcn2 has arg value A internally & does same thing as fcn1(-) except the A value is always used – EX: current add(3, X) → add3(X)

SWE State of the Art – Workarounds: (For Complexity, Comm, Mgrs, Prediction) (Arch)

Complexity (Continued) –

- o- (More) **Write code that is so simple** that jr pgmrs can understand your code
  - o-- Use helper layers – lower/kids never call higher/clients –
    kids are typically fcns (rarely agents), ditto grandkids and even lower layers
  - o-- Solve "cross-cutting concerns" (involving 2+ agents) with fcns – simpler than complicated classes
  - o-- **Constrain local variable scope** (where in code the var is usable) – use blocks (curly braces) to limit em
- o- Write down your tests first – use EIO Rule – consider TDD tools – EIO doesn't always have to be code
- o- **Build up your regression suite** while you are developing – don't break your older code that was working
- o- Use the **Clean Rule** to ensure newbies can easily understand your code – and that **you understand it well**
- o- Convert pseudo-code (natural lang) into **code section header comments** – preserve your pseudo-code
  - o-- NB, some pseudo-code makes a good section comment **but doesn't translate well** into code, eg looping
- o- Make sure that everything you think is a **req't has a test(s)** (1+ EIOs) attached to it
- o- Make your code **tell you what's going on** – log/assert key interim results as **proof stuff is working** inside
- o- Ensure each agent does one conceptual thing (one verb) – **SRP** – **avoid noun-to-class** cuz data != class
- o- Remember to **split** a task into just a **few sub-tasks** (eg, 3-8) with **reasonable conceptual names**
  - o-- 3-8 is for main processing path; okay to add error handling; okay to under-count for a switch
- o- Solve the **main (no-mistakes) processing path first** – only then add in the error & frill handling
  - o-- Simplifies your fcn/method while you are making it work – much easier to understand
- o- Take advantage of arch/processing explanations with **simple visuals** – post a CRC layout where all can see
  - o-- Use task **hierarchies**, **pole** (sequence) diagrams, **FSM** diagrams, **simple** client-helper "**assoc**" diagrams
- o- Build & regression-test full system for each completed feature-slice or bug-fix
  - o-- Target for half-day micro-slice builds – AKA **Daily Build** (or "Continuous" Integration – "**CI**")
  - o-- Equivalent to Add-a-Trick for Integration – ie, integrate small pieces to **limit RT-bug locations**
- o- **Learn new tech** every month – 5 years goes by fast – so does 10 years
- o- Use **simple** Arch/Model **tools** – white boards, post-its, CRC cards – stuff trivial to learn & revise
  - o-- **Don't fight overly-complex tools** to do a simple job – "crude diagrams now" are better
- o- Ensure **branch coverage** (AKA all-stmts coverage) testing
  - o-- If you did the EIO Rule, this should be trivial – cuz your design **only covers** the EIO samples
- o- Ensure **corner-case testing** (AKA Range of Values)
- o- **Agents are PAs** – they **memo** their own data – they have **more than one** closely-related action/method
  - o-- Each should represent **simple concept**
  - o-- No memoed data? **Then no agent**, just use a fcn (or a **cluster of fcns**, maybe in their own **namespace**)
  - o-- **Never force data to become an agent** – agent's concept w/ memoes & actions/methods is the reason
  - o-- A **data transformation function doesn't need to be an agent**
    – fcns don't need an object instantiated to work
- o- Favor **Switches** over the OOP sub-classing+polymorphism trick – switches are much simpler to understand
  and the logic is all in one place, making it easy to find and modify – don't spread it out into sub-classes
  - o-- The OOP sub-classing+polymorphism trick **uses override methods in place of switch cases** –
    and requires double the lines of code – and can lead to Liskov (LSP) bugs
    – (oh, and this "polymorph" trick looks way cool, but it isn't simpler or easier to mod)
- o- Use higher-level fcns in place of looping where possible
  - o-- map = input is a fcn & a data set X → output is a data set Y where each Y.J item = fcn(X.J)
  - o-- reduce = input is a fcn & a data set X → output is fcn(X) – like add(X) → sum of X items

Comm Poor –
- o- Focus on reqts changes by IID (Incremental Iterative Dev) – build/delivery in mini-slices/micro-slices
- o- Let users change focus after delivery of each slice – give them priority
- o- CRC hand-sim with users – solves terminology & other misunderstandings
- o- Use the 2-Story Rule (user pbm domain naming) to ensure users understand your architecture/model
  - o-- used exclusively in DDD (Domain-Driven Development) M.O.
- o- Remember that the users need your help to get their PA solution – **cut them slack, be polite & nice**

Mgmt Poor –
- o- Avoid most foreseeable surprises with **Risk mgmt**/tracking
  - o-- Brainstorm to ID events, Pbb of event (S,M,L), Cost, Impact, Mitigation/Workarounds, Tracking
- o- Avoid mgmt surprises with fair (not fear) **daily standups** – ensure entire team **knows what's going on**
  - o-- **Be open** about your progress and immediate goals – try very hard to be transparent in your development
  - o-- Use the **Reasonable Person Std** – Ask for help early & show what you have
- o- Ensure mgmt also does significant coding – **lead from the front** rather than from above
- o- Avoid SWE def processes/tools that **invent specialty terminology** for ordinary concepts
  - o-- (for fun, check out RUP's phase names)
- o- Remember that the dev members **need your help** to make the PA solution – cut em slack, be polite & nice
  - o-- Always give people a **second chance** (ie, cut them slack) – be "more than fair"
  - o-- Always be **open to change** – even if it means what you're working on get revised

Lab

RPC = is a function call but to a remote-CPU's function, where you wait for the function's answer
Blocking call = is a normal (or RPC-remote) function call, where you wait for the function's answer
EX:

```
xx = foo( 42 ); // and xx "waits" for the foo(-) answer; "blocking" call
zz = cos( ww ); // ww was set above
yy = 3 * xx;
```

Non-Blocking call = is a function call with a "callback fcn" specified (AKA "asynchronous pgmg")
- o- where you continue without getting an answer (no wait)
- o- and when the function has an answer, it calls your "callback" fcn with the answer
- o- and your callback fcn installs the answer in some nice accessible variable for further use
- o- and (somehow) "you are/can-be notified" that the answer is available, finally

"okay, so would an analogy for this be like ordering a pizza,
pizza shop calls you when the pizza is ready,
but in the meantime do what ever you need to do while you wait?" – James Austin Jr.
A: Yes.

SWE State of the Art – Workarounds: (For Complexity, Comm, Mgrs, Prediction) (Arch)
Prediction Bad –
- o- Predict only for small time-periods (eg, Agile 2-4 weeks, or sooner)
  - o-- Also, exists a "**No-Estimates**" movement within Agile – instead ask them to rely on your "good will"
- o- Give cust/users **confidence** with **quick repeated deliveries** of feature mini-slices
- o- Give wide-but-defensible **error bars** on all estimates – maintain your personal track record as "backup"
- o- Use **Half-Day Rule** for training – and to **create a personal track record**
- o- Use S/M/L (**multiple) estimates** & try combining with the **discrete bell-curve eqn**
  - o-- Get multiple **independent** (they don't talk it over) developer estimates – to see the estimate spread
- o- If you're estimating more than a staff month of effort, break it up into **micro-tasks** (half-day)
  - o-- Use what you know well – Half-Day Rule
  - o-- Remember non-work activities, allow for sleep, eat, home-life, holidays, etc.
  - o-- Think about sources of risk – try to estimate (S/M/L) their likelihood and impact

Mini Arch Help
- o- Learn the few dozen architecture mechanisms (styles of multi-agent interactions)
  - o-- Others (should) know how these work – so they are easy to visualize

Buffered Arch (a data-only interface) – reduces coupling
- o-- Separating two agents (or two fcns) complicates the arch, so it should be balanced with big benefits
- o-- Complicates the Static Arch, but usually Simplifies the Dynamic Arch (at Run-Time (RT))

Agents – Mgrs (below) – PAs (Personal Assistants) who do/coordinate higher-level tasks
- o-- Usually, this is what an "object" should be

Data Agents – GRASP "Info Experts" – wrapper for specialty data-handling, & owns the data
- o-- Ie, Agents with their OWNED data – no getters/setters

Wrappers – CRC "Boundary" Agents, SOLID/D (ISP) "Interface Segregation Principle"
- o-- Wrappers **provide simplified API** to Clients
- o-- Can have **more than one** wrapper to the **same box** (eg, specialty Wrappers for a Data Agent)
- o-- Wrapper **simplifies (AKA also reduces) access (AKA coupling)**, but **adds another box** to the arch

Mgrs – CRC "Controller" Agents, GoF Mediators etc, GRASP "Controllers"
- o-- They have 2+-agent **coordination knowledge** (knowledge can also include data)
  - EX: they coordinate multiple sub-agents used as **procedural steps**
  - EX: they decide **which** sub-agents to invoke and **when**
- o-- Separating (sub-) agents both simplifies arch → isolates/hides inter-agent coordination knowledge & complicates the static arch → more boxes to understand – (so keep it simple and obvious)

Micro Arch Help – (Micro == half-day to one-day)
- o- Reduce fcn complexity with
  - o-- Fcn should fit on screen – for better eyeball-vs-scrolling productivity
  - o-- Lower McCabe Cyclomatic Complexity measure – preferably < 10 for **main active exec stmts**
    - o--- Consider exemption for **simple Switch** multi-case stmt
      - – No case-with-exec-stmts that falls through
      - – Has a Default case – simplifies static understanding (it wasn't an MIA oversight)
      - – Has no branches inside a case
      - – Count it as 3 IFs (ie, +3 for McCabe)
  - o-- Use simple (newbie-ish) tech
  - o-- Only does 3-8-ish steps (main active executable stmts) – (maybe excepting a simple Switch)
- o- Rule #4 (EIO) – written down the samples, but not necessarily as actual (say TDD) code
- o- Reduce pre-test incrementally-added fcn complexity (with Rule (Hunt) Add-a-Trick)

o-- pre-test – amount of code added before compile-run-test
o-- IE, Don't add big (eg, >20 lines) batches of untested code
SOLID/D – avoid classes & especially hierarchies as much as possible
o-- to avoid OCP, Liskov bugs – and **SRP "class-blizzard"** – and D1/D2 overkill (see all, below)
DRY (Don't Repeat Yourself) Principle – means never have a code section show up in two places
o-- Two intents (points) – avoid code "bloat", and "less code means simpler (static) arch to understand"
o-- Avoid it – it is an artificial restriction that gets in the way of Rule (Fast)
GRASP –
o- Creator(Ctor) – Ctor = Constructor, Dtor = Destructor
o- Pure Fabrication – a class that is not a concept in the user's problem domain
o- Indirection(Cutout, or Wrapper) – you (or your knowledge) has to "go through" a middleman
o- Protected Variations (a Wrapper) → Future Prediction, better be very good; usually this is bad
o-- A sub-concept of Indirection
o-- Maybe use GoF Strategy, but definitely "wall off" the changable area
o-- PV isn't a class arch mech, but it is what you are protecting
FP – use it whenever you can – it's usually rarely needed – but it rarely slows down a pgm where it counts
– means **slow copying** of structured data in order to mod a slot – immutable data structures
o-- So **outsiders see no change** if they also had a ref/ptr to the structure before the mod
o-- BUT, extra CPU time rarely matters (only 1-2% of your code is a bottleneck, good odds)
o-- AND, preventing leakage to outsiders is simpler to understand (esp. under project stress)
o-- SO it **simplifies** the static arch – and especially the **dynamic arch**
– don't use it for the static/stable agents themselves
– but consider it for any data structure that is referenced/pointed-to by more than one agent
– so that one agent modifying it doesn't surprise other agents looking at it

OOP Trade-Offs

(*) Key: **Avoiding** complex component-interaction **RT bugs**
o- Hard, expensive to find/fix
o- OOP has some "issues"

LSP Polymorphism vs Fancy Naming

Q: Why are we stuck with LSP?
A: Dynamic dispatch of override md based on call fcn body based on
(generic) polymorphism --> ie, **based on name + arg type(s)**, and no class hier needed
o- simple LSP polymorphism → same fcn/method name but diff object class and override of mom-class
o- Double-dispatch polymorphism → same fcn/md name but 2 diff classes, each overrides their mom-class
o- "Generic Functions" == **(generic) polymorphism**
(*) **LSP** applies to Not just OOP class hierarchy, but any polymorphism
o- But polymorphism naming makes static arch simpler
o- Hence, we trade simpler arch for possible LSP bugs
o- And single-inheritance OOP still allows LSP bugs
(*) Reason for using polymorphism – one name and (almost) any mix of arg types – & "it just works"
Alt: Fancy Naming Use common "family" name prefix with arg-types-specific suffix name:
eg, add_strings, add_string_n_int, etc.
(*) Get simpler arch but a **large pile** of type-specific **fcn names**
(altho, family+specific naming helps a lot)
o- Polymorphism was supposed to simplify this AT NO COST
o-- And its mech is "geek candy"
o- But COST is LSP bugs **if misused**

OCP vs Upgrade Class In-Place

Q: Why are we stuck with OCP?
A: Cuz we use Class Hier to get 1) Polymorphism & 2) auto Cut-n-Paste
o- And mechs are "geek candy"
(*) But with (say) 40 old sub-classes built on mom-class,
you need massive regression tests with very good coverage,
Else a **mom-class mod** (which violates OCP) can cause **OCP RT bugs**
(*) But regression testing is thin for original product internals (fcns & methods)
But good for defect fixes – AKA after-deploy bugs, not pre-deploy bugs
o- Cuz pre-deploy tests are primarily top-level requirements tests
o- Most unit tests, etc., are hard to include in regression suite
** Cuz small internal changes can ruin low-level tests – IE, they become obsolete due to dev bug fixes
o-- And they're expensive to upgrade
* Some TDD tries to keep up the unit (& I/T) tests
o- Complex code-sharing can make Arch less simple, trade-off
Alt: Upgrade Class In-Place Cut-n-Paste, so no class hier needed
o-- maybe a very shallow class hierarchy – thus avoiding **SRP "class-blizzard"**
o- Each class is (mostly) self-sufficient – depends on no mom class
** Simpler Arch – (almost) one-stop to understand a class (ie, object kind)
But during dev:
o- Maybe mistakes using cut-n-paste to build similar classes
o-- EG, mod one class & cut-n-paste mod to a dozen others
o-- Esp, mod one method & cut-n-paste (slightly modded) to a dozen others
o- Code & memory code "bloat" -- duplicated code
o-- Memory exe size ("memory footprint") a big pbm in "old days"
o-- Mistakes propagating class method mods cause RT bugs
o-- Each class md likely has tiny class-specific stuff, but the bulk is code copied verbatim
o- Pgmr "sneak-around" trying to share code in weird ways – reinventing RT class supt
o-- NB, these weird ways are usually class-hier built-in stuff hidden in the RT supt
EG, "rolling" your own VTables, class "frames"

SRP vs Lemmas

o- SRP (Single Responsibility Principle)
Q: Can SRP be bad?
A: Collecting and en-classing ALL tiny bits of commonality
o- Each in its own class in its proper spot in the hier
AKA "Factoring out" commonality – a useful math-ish technique – NB, this is unrelated to "Refactoring"
o- Jockeying the class hier as each common piece is found
(*)* Leads to "**Blizzard Of Classes**" – **SRP "class-blizzard"**
o- Often under pressure to abide by DRY (Don't Repeat Yourself)
o- Complicates arch, under guise of (math) "factoring"
** But math-ish mechs are "geek candy"

(*) Most situations are not complex Trees (kid = sub-class) – at the problem domain level
o- where you are converting nouns to classes in a class (tree) hierarchy
But rather, most are complex DAGs (common kid & many moms) – (and some are general digraphs)
o- A single object can have multiple attributes → means object belongs to multiple clusters/groups
(*)** Very common arch fail is using trees that s/b DAGs or a list of groups each object belongs to
o-- Cuz, it's easy to see the tree relationships at first
o-- And modding a big Tree is very hard
due to very strong hier coupling (CF OCP) – and it is worse when faced with an **SRP "class-blizzard"**
(although compile-time errors help a bit)

Alt: Lemmas Reserve SRP for **known named concepts** (as it should be)
o- Math actually does this:
o-- "Theorem" is (close to) a named concept
o-- "Lemma" is a wrapping up of **convenient commonality –** ("husk" in Latin)
(*) Don't force convenient commonality to be a class,
Reserve classes for well-named concepts (eg, in Pbm Domain)
o-- "Lemmas" are GRASP "Pure Fabrications", and s/b rare (or well-known comp sci things, eg Hash)

SOLID's D1/D2 vs DIY Helpers

D1/D2 = (Dep-Inversion)/(Dep-Injection)

Q: Can D1/D2 be bad?

A: Maybe

Recap:

D1 (Dep-Inversion) -- use abstract/interface cutout class

o- So, changes to Helper guts (but no change to API)
don't cause recompilation of Client

(*) But, in most OOP Langs, using Ctor causes this recompilation (even if using a cutout interface class)
o-- cuz using the Helper's Ctor requires knowing ("including the header") the Helper class, too
So, still need to use D2 – meaning you don't do the Ctor to create the Helper agent

D2 (Dep-Injection) -- special Helper Builder **injects** (links up) Client with Helper-ref at run-time
o-- Client uses cutout class also (D1)
o-- Guarantees non-API mods to helper (class) guts don't recompile Client

** And D1/D2 mechs are "geek candy"

Alt: Client Builds/Owns Helper **till proven Need**

(*) For most situations Helper only works for Client
So, Client building Helper isn't a problem, cuz Client OWNS Helper
AND, DIY is a simpler arch

** But, slightly inhibits generalizing/"factoring out" Helper as a reusable part (Caveat, Rule #1)
o-- "**Factoring out**" as "splitting out & making it its own thing (fcn, md)" in the mathematical sense

o- Cuz Client-specific Helper API likely needs modified,
Meaning Client will need to be modified,
OR Client won't use the new Helper (so some duplicated code)

(*) Avoid complicating arch with extra one-off cutout class – (**trades simpler arch for lower-coupling**)

o- And DIY avoids Pre-Factoring Before Need

(*)* Pre-Factoring Before Need violates both JIT and YAGNI
JIT = Just in Time -- wait till proven need
YAGNI = You Ain't Gonna Need It -- almost always

Alt #2: Client calls Mediator/Middleman for help, then Mediator calls Helper

o- Adds a Mediator/Middleman agent into the mix

(*) Avoid Client knowing anything about Helper – Mediator can even transform the API

o- Also, this is **doable in non-OOP styles**

Helper Class vs Single Fcn

Q: Can a Helper Class be bad?

A: If it's only a single method, then

INSTEAD of calling a named fcn? – AKA "Procedural Style Coding", which is usually simpler code

Why add extra code to

o1- **Define** a Helper class, and

o2- **Instantiate** a Helper object, and

o3- **Pass** around a Helper object ref/ptr, and

** And do you need an Extra class to Construct the Helper obj, so you can do Dep Injection?

o4- Have to **call** for help **via** the Helper **object ref/ptr**

(*) This adds **extra Code bloat** and (slightly) complicates the arch

Q: How about a Helper with 2 methods?

A: How related are those 2 methods?

o- Couldn't they be just 2 named fcns?

o- Is this a misplaced attempt at grouping?

Quality Assurance (QA) – Assuring likelihood of project success w.r.t. Validation (ie, users like result)

o-- First, of course, assure **project gets to deployment phase**

o-- **Second** – make sure **process** isn't getting in the way of (development) **progress**

– Q: How to know? A: Fewer mtgs (on subjects below), & more progress on Features completed

o-- NB, QA is a Mgr issue, but it may be imposed on the project team (incl mgr) from higher-up in biz

Ensure --

o- **VCS/CMS** for all notes, EIOs, docs, code, tests, logs – find, or rollback to, any version of any item

– NB, non-code docs & notes rapidly go out-of-date as system is being built

o- **Users** (more broadly, stakeholders) **approve** of features/reqts & product quality levels (ilities)

– to the best of their ability (cuz users aren't compsci tech knowledgeable)

o- **Bug Tracking** process for all bugs found in code that has been submitted to VCS as done/tested

– NB, sometimes low-level bugs may be left in place to meet delivery schedule

– But users should be apprised of these

o- **Reqts EIOs** before/during UC development – so that EIOs drive UC design – every reqt s/have a test(s)

– (NB, **EIOs** at all levels **should drive design** at all levels)

o- **Users agree with EIOs and UCs before**/during UC & arch/model/CRC development

o- **Users agree with preliminary arch/model** – eg, via CRC hand-sim, or fancier prototypes

o- **Detailed/Unit EIOs before**/during Detailed Design & Unit Design – less gold-plating or future reuse

o- **EIOs for testing** of component and unit **APIs** – so devrs can do pre-integration API verification

– reduces integration bugs (which can still happen – **APIs** merely help, but **aren't the entire coupling**)

o- **Unit testing** (with EIO data) **is performed** – & test "scaffolding" with EIOs & run logs go into VCS

o- **Component-level integration testing** (with EIO data) is performed – ditto

o- **Reqts-level testing** (with EIO data) is performed – ditto

– & reqts tests go into **Regression suite**

– NB, lower-level tests typically require scaffolding code, which complicates use in Regression suite

o- **System internals description docs** are revised to reflect actual product (ie, "**as-built**" docs)

o- **Major bug**, causing major system (eg, including some arch) changes, should also **include EIOs**, etc.

o- When behind schedule, **devrs re-estimate** their tasks – as often as weekly

o- When behind schedule, **users** are allowed to **help revise project reqts/features priorities**

SEI CMM (CMMI) – Capability Maturity Model (, Integrated)

o- For **Evaluating likely success of a business** running a S/W project – rate a business selling "fresh S/W"
– Success = on-time, under budget, features desired, validation by users

o- Classifies software dev organizations into **5 levels**:

Level 1 : Initial (eg, "free-for-all")

o- These organizations tend to use **code-and-fix** style development.

o- Software development has no written (auditable) process – anarchy.

o- **Projects tend to run over** budget and behind schedule.

o- Organizational knowledge is contained only in the minds of individual programmers;
SO, when a programmer leaves an organization, so does the knowledge.

o- Success **depends largely** on the contributions of individual "**hero**" programmers (ie, "strong" pgmrs).

Level 2 : Repeatable

o- Basic project management **practices** (**written**) are established on a **project-by-project** basis,
and the **organization ensures** that they are followed – Q: How? A: via QA auditing.

o- Project **success no longer depends solely** on specific individuals – (ie, devrs are "plug-replaceable").

o- The strength of an organization at this level depends on its **repeated experience with similar projects**.

o- The organization **may falter** when **faced with new** tools, methods, or kinds of software projects.

Level 3 : Defined – (AKA you build it well, and you use written processes for all projects)

o- The software organization **adopts standardized technical and management processes across the org**,
and individual projects (within the org) tailor (in writing) the standard process to their specific needs.

o- A **group** within the organization is **assigned responsibility** for software **process activities**. (eg, QA)

o- The organization establishes a **training program to ensure** that managers and technical staff
have **appropriate knowledge and skills** to work at this level.

o- These organizations have moved **well beyond code-and-fix** development,
and they **routinely** (but no percentage indicated) **deliver software on time and within budget**.

Level 4 : Managed – (AKA you collect blizzard of "useful" data on all projects, ideally for improvement)

o- Project outcomes are **highly predictable**.

o- The process is stable enough that **causes of variation** (in outcomes) can be **identified and addressed**.

o- The organization **collects project data** in an **organization-wide database**
to evaluate the effectiveness of different processes.

o- All projects follow organization-wide **process measurement standards** so that the data they produce
can be meaningfully analyzed and compared.

Level 5 : Optimizing – (AKA you have a meta-process to improve your processes, & hence success rate)

o- The focus of the whole organization is on continuous, proactive identification and dissemination
of **process improvements**.

o- The organization varies its processes,
measures the results of the variations,
and diffuses beneficial variations as new standards.

o- The organization's quality assurance (QA) focus is on defect prevention
|through identification and elimination of root causes.

Caveats :

o- YMMV – AKA "gaming the system"

o-- "**Potemkin**" (AKA fake) **processes** in order to gain CMM cert – but SEI samples employees on practices

o-- **Buying "influence"** with accreditation organizations

o-- **Hiring senior accreditation mgrs** into your business after positive Level certification – quid pro quo

o- Why? There is tremendous money available to companies with CMM Level certifications

o-- Can a biz maintain such a facade in the long term? – Maybe? – Best way is to do the Mature thing well

Run-Time Environment – Compiler-based Languages

RT Memory Areas – 5 Kinds (for most major langs)
- o- **Global Constants** – that can't fit into a machine instruction – usually > 8- or 16-bit (instruction upper limit)
  - o-- EX: string constants, like "Hello there" – EX: Vtables for the classes
- o- **Global Variables** – accessible from all code, (except when global names are shadowed by local names)
  - o-- Includes "**private globals**", eg C/C++ keyword "auto" added to a local var makes it a private global
    - – Local → only accessible from inside its fcn or block
    - – But in Global Vars, not in the fcn's local Call Frame on the Call Stack
    - – Hence, var value survives fcn return and is still there on next call to that same fcn
    - – AND not accessible from outside its scope (AKA inside its fcn or block, or block-remainder)
- o- **Code Space** – both global fcn code and nested (ie, local, hence private) fcn code is here
  - o-- **For regular fcns**, caller has exact address of code – uses a "call" or "jump-to-subroutine" instruction
    - o--- Compiler sets a symtab (symbol table) index for the fcn name/symbol in the "call" instruction
      - – AND the Linker (which lays out Mem Areas) replaces the index with the fcn's address
  - o-- **For dynamic dispatch fcns**, caller ends up jumping thru offset in some **VTable** (a Constant table)
    - o--- Compiler sets up an indirect call through the named VTable and with an offset to the fcn/md name
      - – AND the Linker replaces the VTable symtab index with the VTable's address in Global Constants
- o- **Call Stack** – (invented to handle recursion, and later all CPUs added H/W support for it – SP = Stack Ptr
  - o-- **Every fcn/md call** creates **Call Frame**, pushed onto the Call Stack
    - o--- **Call Frame** – has 1 slot for "return address"
      - for (next instruction to run, caller's next, or its mom's next – specific to CPU+Lang_Runtime)
      - – has "parameter" slots (local var slots) for all its arguments
      - – has enough slots to store all its local vars (those active at the same time, so <= the actual #vars)
    - o--- Allows multiple copies of a recursive fcn's parms & local vars to exist at the same time (easily)
      - – Before this, you built & manages your own stack in code to do recursion
    - o--- Earliest major lang to supt recursion – as with most things, Lisp 1958
    - o--- Likely still some extreme embedded pgmg systems that "can't afford" recursion
- o- **The Heap** – not a "heap tree", but just a large area ("pile") of bits – leftover
  - o-- All dynamic data is placed here – a data-sized pile of bits, a "**block**", (with random values) **is "allocated"**
  - o-- When the pgm is done with a dynamic object/structure/array, the **memory is "reclaimed"**
  - o-- The **Heap Mgr** (part of the lang's RT mech) handles **allocating and "reclaiming"** memory blocks
    - o--- It usually has a **"Free List" of unallocated blocks**
      - – at startup, one really big block (usually)
      - – **after** running awhile (ie, **allocating and reclaiming**), it has a bunch of disconnected blocks
      - – AND between disconnected blocks are allocated "holes"
      - – AND the **Free List** is typically a **linked list of free** (currently unallocated) **blocks**
  - o-- **Reclaiming** is done either by the Pgm/Devr **calling a "free-me" fcn** → calls the Heap Mgr
    - – OR **automatically** done by the "**Garbage Collector**" (AKA "**GC**")– present in some langs
  - o-- The "Garbage Collector" does fancy reclaiming – (1-st was Lisp, of course)
    - o--- Like recursion, GC was once thought to be too expensive (slow and plump)
    - o--- Fast GC algos (for Lisp) were invented in mid-80's – but any lang can do this
    - o--- Interpreted langs (which are usually about 30x slower) typically don't worry about GC speed
  - o-- Without GC, when no single block is big enough for an alloc request, the Mgr tries to merge blocks
    - o--- The Free List can end up with side-by-side blocks because it is (thot) too costly to merge on return
      - – Cuz the Free List isn't kept in block-address order (costly to resort List on every block return)
      - – So the Heap Mgr only tries to merge on "can't find block big enough"
  - o-- Either way (**GC or not**), if big enough block isn't available, pgm might crash (or throw exception, etc.)

(in Facts and Fallacies of Software Engineering, by Bob Glass, 2003)

Fact 2
- o- The **best** programmers are up to **28 times better** than the worst programmers,
  according to "individual differences" research.
  Given that their pay is never commensurate, they are the biggest bargains in the software field.

Caveats (it's very controversial):
- o-- Many **senior pgmrs disagree** both with this kind of "fact" and with some of the research studies that initially reported this kind of result (eg, 10x or more difference). Including at major biz (eg, Google).
- o-- OTOH, there are **many anecdotes** of **extremely strong pgmrs** who exhibit this difference.
- o-- OTOOH, it is often claimed that such extremely strong pgmrs are "prima donnas" that **cause social pbms**.
  But, that kind of defeats the claims that there are no such pgmrs.
- o-- OTOOOH, the 80-20 Rule (for pgmrs doing the work) means 80% of the S/W coding
  is done by 20% of the developers – 80-20 Rule is Pareto's Rule, Pareto stat distribution of who/what contributes the most, least.
  → in 343, the "80-20 Rule" means 80% of the bugs come from 20% of the code
  Pareto Distribution is followed more accurately in larger systems (where differences are "washed out")

Fact 29
- o- Programmers shift from design to coding
  when the problem is decomposed to a level of "primitives"
  that the designer has mastered.

  If the coder is not the same person as the designer,
  the designer's primitives are unlikely to match the coder's primitives,
  and **trouble** will result.

Fact 33
- o- Even if 100 percent test [branch] coverage were possible [in a large pgm],
  that is not a sufficient criterion for testing.
  Roughly **35 percent** of software **defects** emerge from **missing logic paths**[branches] ,
  and another **40 percent** from the execution of a unique combination of logic [branch] paths.
  They will not be caught by 100 percent [branch] coverage.

– Nox below here

Fact 41
- o- Maintenance typically consumes 40 to 80 percent (average, 60 percent) of software costs.
  Therefore, it is probably the most important life cycle phase of software.

Fact 42
- o- Enhancement is responsible for roughly 60 percent of software maintenance costs.
  Error correction is roughly 17 percent.
  Therefore, software maintenance is largely about adding new capability to old software, not fixing it.

Fact 44
- o- In examining the tasks of software development versus software maintenance,
  most of the tasks are the same
  — except for the additional maintenance task of "understanding the existing product."
  This task consumes roughly 30 percent of the total maintenance time
  and is the dominant maintenance activity.
  Thus it is possible to claim that maintenance is a more difficult task than development.

"No Silver Bullet: Essence and Accidents of Software Engineering",

5. Promising Attacks on the Conceptual Essence (Brooks' opinion)

(*) Want Methodology/Tech to speedup "Essence Building"

o-A **Bottleneck**: define & linkup arch components & data sets

"the formulation of these complex conceptual structures" p 18

[CS: Left out – **[nox]**

o- **2nd Bottleneck**: Resolve RT bugs in complex implementation, found in I&T phase

o- **3rd Bottleneck**: Resolve spec bugs, found in Validation (users like it?) phase

o- **4th Bottleneck**: Avoid bugs from gold plating (unneeded code)

o-- Spec & Arch gold plating]

o-B **Faster expressing/coding** arch components doesn't speedup overall

o-5.1 Buy versus Build (COTS)

o-- $100K tool == 1 FTE/yr, strong-ish pgmr (FTE = Full-time Employee)

PRO: If you can use them (eg, framework) maybe save 80-90% of your coding

o- Most applications of COTS are **too specialized** for medium to big pgms

o- Upshot, doesn't help enough

o-- Caveat: Frameworks are a good start -- fit into their arch

o-5.2 Reqts Refinement and Rapid Prototyping

"The hardest single part of building a software system is deciding precisely what to build." (Spec (& maybe parts of the Arch))

o- Spec is most important (obviously)

"Therefore the most important function that the software builder does for his client is the **iterative** extraction and refinement of the product **requirements**."

o- Get spec wrong, validation fails, project fails

"impossible for a client... to specify ... the exact requirements... before having built and tried some versions of the product he is specifying."

o- The users have to **play with it to know** if they got the reqts done right

o- W/out iterating, project fails

(*) Key: rapid prototyping/delivery to get spec right

Prototype: (4 Things)

o- Does main fcns

o- Shows important UI

o- No Frills

o- Performance not req'd

(*) Purpose: validate spec – users like it, so the spec must be right

o-- for consistency and usability

o- **Std S/W Proj (Big Bang style, AKA BUFD Big Up-Front Design)**

o1. Specify in advance (by Cust, eg via RFP)

o2. Get bids to build

o3. S/W Biz builds it

o4. Installs & it works per spec

(*) Brooks: Big Bang style is "Fundamentally wrong"

o-**5.3 Incremental Dev [and Review by Cust]: Grow, don't Build**

"The **morale effects** are startling. Enthusiasm jumps when there is a running system, even a simple one."
"One always has, at every stage in the process, a working system."

Recall: **Morale is the Most Important Thing** in S/W Development

o-**5.4 Great Designers**

"**Whereas** the difference between poor conceptual designs and good ones **may** lie in the soundness of design method, the difference between good designs and great ones **surely does not**. Great designs come from great designers. Software construction is a creative process."

o- Sound design method gives you good or great designs dep on the designer

"Study after study show that the **very best designers** produce structures that are faster, smaller, simpler, cleaner, and produced with less effort. The differences between the great and the average approach an **order of magnitude [10x]**."

o- Develop ways to grow great designers
[CS: How? Learn as many arch styles & algos as you can. Keep Learning: SOA & P2P & uSvcs arose after Brooks' paper.]

Q: What HAS been a Silver Bullet? (at least somewhat)

Agile's extreme user feedback loop:

o- 2-8 wk sprints/deliveries
o- Reset/Update the Features & Priorities list from scratch for each sprint/time-box – per User feedback
o- **And NO OTHER Agile features bear directly like these 2.**

How Helps?

o- Avoids multi-year massive failure
o- Allows plug pulled early, by cust – saving future wasted cost

How Addresses Essence? (Essence needs the Arch to be Validated)

o- Addresses the Keys: Quickly debugs Spec, incrementally
  o- Spec (What user needs, not about S/W internals)
  o- Design (Arch/Model -- parts to deliver the needs)
  o- Testing (Does the Arch deliver? Validation)

Drawbacks:

o- Still assumes can get to final arch via baby-steps (by tiny delivery increments)
o- Requires very modular arch to keep complexity low

CRC Paper-&-People Hand-Simulation

o- Lets both dev team **and users** see if the Arch makes sense
o- Finds bugs & holes in the Arch and in the Reqts before the Build phase

[CF The Mythical Man-Month, 2e, 1995]

Ch 17 "No Silver Bullet" Refired p195

o- Object-Oriented Programming—Will a Brass Bullet Do?

James Coggins, author ... of the column, "The Best of comp.lang.c++" in journal "The C++ Report", offers this explanation:

"Unfortunately the self-same strong type-checking in C++
that helps programmers to avoid errors
also makes it hard to build big things out of little ones."

David Parnas (Brooks' IBM & UNC partner), whose paper was one of the origins of object-oriented concepts, sees the matter differently. He writes me (Brooks):

"The answer is simple. It is because [O-O] has been tied to a variety of complex languages.
Instead of teaching people that O-O is a type of design, and giving them design principles,
people have taught that O-O is the use of a particular tool.
We can write good or bad programs with any tool.
Unless we teach people how to design, the languages matter very little.
The result is that people do bad designs with these languages and get very little value from them."

o- Reuse

Capers Jones reports that all of his firm's clients with over 5000 programmers have formal reuse research, whereas fewer than 10 percent of the clients with under 500 programmers do.

David Parnas writes,

Reuse is something that is far easier to say than to do.
Doing it requires both good design and very good documentation.
Even when we see good design, which is still infrequently,
we won't see the components reused without good documentation.

Ken Brooks comments on the difficulty of anticipating which generalization will prove necessary:

I keep having to bend things even on the fifth use of my own personal user-interface library.

Capers Jones reports that a few reusable code modules are being offered on the open market at prices between 1 percent and 20 percent of the normal development costs.

Tom DeMarco says,

I am becoming very discouraged about the whole reuse phenomenon.
There is almost a total absence of an existence theorem for reuse.
Time has confirmed that there is a big expense in making things reusable.

Brooks: my estimate of the effort ratio [to build something as reusable] would be **threefold**.

Ch 18 Propositions of The Mythical Man-Month: True or False ?

#2.9. As a discipline, we **lack estimating data**. // CS: data is plentiful → lack validated data w error bars
#2.11. **Brooks's Law**: Adding manpower to a late software project makes it later
#3.3. A small sharp team is best
#5.3. "The second [system] is the most dangerous system a person ever designs;
the general tendency is to over-design it."
o- Don't generalize [for reuse] from 1 or 2 data points

#7.14. **Parnas** argues strongly that the ...
parts should be encapsulated
so that no one needs to or is allowed to see the internals of any parts other than his own,
but should see only the interfaces
o- Very Low Coupling between agents – AKA "Information Hiding"
#8.3. Programming [effort] increases; goes as a power of program size

Net on [Silver] Bullets—Position Unchanged
R. L. Glass, writing in 1988, accurately summarizes my [Brooks's] 1995 views:
So what, in retrospect, have Parnas and Brooks said to us?
That software development is a conceptually tough business.
That magic solutions are not just around the corner.

Perspective on Agile and its key ingredient – Iterative Incremental Deliveries
[CF "Iterative and Incremental Development: A Brief History", by Larman and Basili, **2003**]

1960s – NASA's early 1960s Project Mercury
o- Ran with very short **(half-day) iterations** that were time boxed (AKA demo or ship when time's up)
o- The development team conducted a technical review of all changes [ie, change reqts or arch]
o- Applied the Extreme Programming [mid-90's] practice of test-first development
o-- planning and writing tests before each micro-increment
o- They also practiced top-down development with stubs (AKA "mocks")
Gerald Weinberg, on the development team, said:
All of us, as far as I can remember, thought waterfalling of a huge project
was rather stupid, or at least ignorant of the realities

1970s – Light Airborne Multipurpose System LAMPS
o- Incrementally delivered in 45 time-boxed iterations (one month per iteration)
o- Four-year 200-person-year effort involving millions of lines of code
o- Every one of those deliveries was on time and under budget.

1975 – "Iterative Enhancement: A Practical Technique for Software Development" by Vic Basili and Joe Turner
o- Develop a software system incrementally
o-- [Their sample real project] 17 iterations over 20 months.
o- Allowing the developer to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the system.
o- Learning comes from both the development and use of the system, where possible.
o- Key steps in the process were to start with a simple implementation of a subset of the software requirements
o- Iteratively enhance the evolving sequence of versions until the full system is implemented.
o- At each iteration, design modifications are made along with adding new functional capabilities.

1976, "Software Development," by **Harlan Mills**
Software development should be done
incrementally, in stages
with continuous user participation and replanning
and with design-to-cost programming within each stage. [Don't do full cost planning up front.]

1977 to 1980, NASA Space Shuttle, primary avionics software
- o- 17 iterations over 31 months, averaging around eight weeks per iteration
- o- feedback-driven refinement of specifications

1980, "Adaptive Programming: The New Religion" by **Gerald Weinberg**
- o- The fundamental idea was to build in small increments
- o- With feedback cycles involving the customer for each.

1985, "Evolutionary Delivery versus the 'Waterfall Model'," by **Tom Gilb**
- o- Recommending frequent (such as every few weeks) delivery of useful results to stakeholders [users, etc.]

1986, **Fred Brooks**, from "No Silver Bullet"

On using an **Iterative Incremental Deliveries** S/W Dev MO
- o- "Nothing in the past decade has so **radically changed my own practice**, or its effectiveness" [as incremental development has done].

On using a **Waterfall** S/W Dev MO
- o- "Much of present-day software acquisition procedure rests upon the assumption that one can specify a satisfactory system in advance, get bids for its construction, have it built, and install it. I think this assumption is **fundamentally wrong**, and that many software acquisition problems spring from that fallacy."

2003, **Larman & Vasili**, "Iterative and Incremental Development: A Brief History"

"A 1999 review of failure rates in a sample of earlier DoD projects drew grave conclusions:"

"Of a total $37 billion for the sample set, [total $$ for all these projects together]
75% of the projects failed or were never used, and
only 2% were used without extensive modification."

2010, **Fred Brooks**, in his book **The Design of Design**

Harlan Mills's system [1971] of incremental development and iterative delivery
is the best way to stay quite close to users right from the very start of the project.

One builds a minimal-function version that works;
then one gives it to users to use, or at least to test-drive.

Even products being built for a mass market can be tested on a sample of users.

"No Silver Bullet: Essence and Accidents of Software Engineering",

TR86-020 September **1986** Frederick P. Brooks, Jr., UNC Chapel Hill
CF https://www.cs.unc.edu/techreports/86-020.pdf

2. Does It Have to be Hard? Essential Difficulties

Aristotle view of difficulties
- o- 2 kinds: Essence & Accident

Essence = intrinsic to S/W (innate, essence, built-in, necessary)
What are a S/W pgm's necessary parts?
Conceptual Parts:
Declarative Parts (**no timing, no sequencing**)
- o- data sets
- o- relationships among data items – invariants (eg age & current date & birthday go together)

Procedural Parts (steps, sequencing, timing, "dynamics")
- o- algorithms (adequate) // process steps
- o- functions (sub-algo modularity+API) & calls to em
- **NOT Referring To: any specific representations/langs/implementations**

Keys for the S/W Conceptual Construct (before the 'Build' Phase):
- o- **Spec/Req'ts** (What user needs, not about S/W internals)
- o- **Design** (Arch/Model -- parts to deliver the what is need by the user)
- o- **Testing/Prototype** (Does the Arch/Model/Design deliver user needs &/or likes? Validate, not Verify)

(*)** Can we find out before we "staff up" and build the full system, that our Design will be liked by the user?
Not: build/test specific implementation – not even a no-frills implementation
(Brooks cares about this because of all the failed S/W projects that were delivered but were unsuitable.)

Keys, Irreducible Essence – (Irreducible == stuff you can't avoid dealing with)
- o-**2.1 Complexity** – it scales exponentially; comprehension overwhelmed
  - o-- data sub-sets (objects) can influence each other
  - o-- It's why Low Coupling/Thin Links are used to reduce it

- o-**2.2 Conformity** -- Conform to Weird Outside Stuff
  - o- Big pgms always must conform to environment/interfaces to other systems
    - o-- Use of Gof patterns: **Proxy,Facade,Mediator, Adapter, Strategy**
  - o- Esp. Expected user data/processing needs

- o-**2.3 Changeability** – (of Reqts **during the time spent constructing** the S/W product)
  - o- First Understanding (of the Pbm) is **Never Correct**; (hence Agile)
    Once program is in user hands:
    - o-- devrs find **mis-understandings**
    - o-- users **realize poor reqts**
    - o-- users **imagine new better reqts**
  - o- Always **Biz pressure** to change:
    - o-- For Mktg/Compete, Mktg/Sales, PR, Tech, Laws/Regs/Certifications
    - o-- Hence, shorter dev-to-ship 'cycle' ==> faster recovery from changes

o-**2.4 Invisibility/Emergence – (The Inability to See the Complexity All at Once, & Understand it)**
o- **S/W = Construct of Interlocking Concepts**
o- **Visualization invaluable**: Blueprints, Maps, Tour guides, Trees, Graphs, UML, "Cheat-sheets"
o-- Easier to see pbms/bugs/omissions (from that visual viewpoint)
o- S/W is **too complex to capture** in such simple formats

*"As soon as we attempt to **diagram software structure**, we find it to Constitute not one, but **several, general directed graphs**, **superimposed** one upon another. The several graphs may represent the flow of control, the flow of data, patterns of dependency, **TIME SEQUENCE**, name-space relationships. These are usually **not even planar**, much less hierarchical."*

o- (*) Traditional Sol'n: **Abstract/Model**
o-- Simplify a view **by ignoring parts**
o** Can't find bugs **involving parts that are ignored**
(*) Remains inherently un-visualizable

3. Break-thrus on Accidental Difficulties
o-**3.1 High-Level Languages**
o- Better abstractions for essence, plus adds new accidental stuff
o-**3.2 Time-Sharing** -- 1 (virtual) computer per devr
o- Shorter feedback loop on all pgm bugs
o-**3.3 Unified Programming Environments (IDEs)**
o- Auto-help finding & ID very simple details

4. Hopes for the Silver (based on existing tech + some improvements) – Other's attacks on the Essence
o-**4.1 High-Level Language (HLL -nox) Advances**
o- Con: Over-rich feature set, hard to learn
o- Pro: Programming in a working subset if it helps
o- Con: Just another HLL; up from Asm bugs to Stmt-Step bugs

o-**4.2 OOP & other Technical Fads**
o- 2 Idea Kinds:
o-- ADTs nox (op algebra rules; self-contained) -- helps avoid usage bugs
o-- Class hier -- avoids copy-paste bugs; but needs **SOLID/D**!

o-**4.3 AI**
o- **AI-1**: Human problem solving tech: ML, GA, ANN
o- AI has a Slippery def
o- Not close to Human-level, weak & pbm-specific & buggy
o-- NB, ANN-Deep-Learning latest useful fad
o- **AI-2**: Expert Systems (see sec 4.4)

o-**4.4 Expert Systems**
o-- Pick SME nox (Subj Matter Expert) brains for Domain rules & impl an RBS
o-- Used extensively in simple domains (eg, loan qualifying)
o-- Scale up reqs Big Knowledge tech, failed for 30 yrs
o- Best Help: prompt & double-check on newbies (new trainees)
o-- Not in general use for this yet

o-**4.5 "Automatic" Programming** (AP) (AKA Formal Mds)
o- Idea: Write spec, push button, out plops pgm verified – (type I, m->p)
o- Pbm, how is spec lang diff from HLL? No diff, just "higher" maybe.
(*) Parnas: view of "AP" *Glamorous, not meaningful*
o- Doesn't spec pbm, but the pgm!; Not the Need/What but the Mech/How.
Exceptions that Work (using AP): Pbm has
o- Few knobs o- Many known soln mechs o- Exist std rules for soln

o-**4.6 Graphical Pgmg**
o- Idea: Write visual diagram(s), push button, out plops pgm verified
o-- VB framework provides std GUI **wrapper** code
o-- UML tools provides std general **wrapper** code, but harder
CON:
o-- Screens still too small !! (to see all of the complex diagrams needed)
o-- Needs far too many diagrams w/ linkages that are too complex

o-**4.7 Pgm Verif** (AKA Formal Mds) (type II, m<-p/model)
o- Verif Design/Model before detailed design/code/test
o- Prover only says model verified; **doesn't prove spec correct (aka validate)**
o- Prover can have bugs
o- pgm still needs built

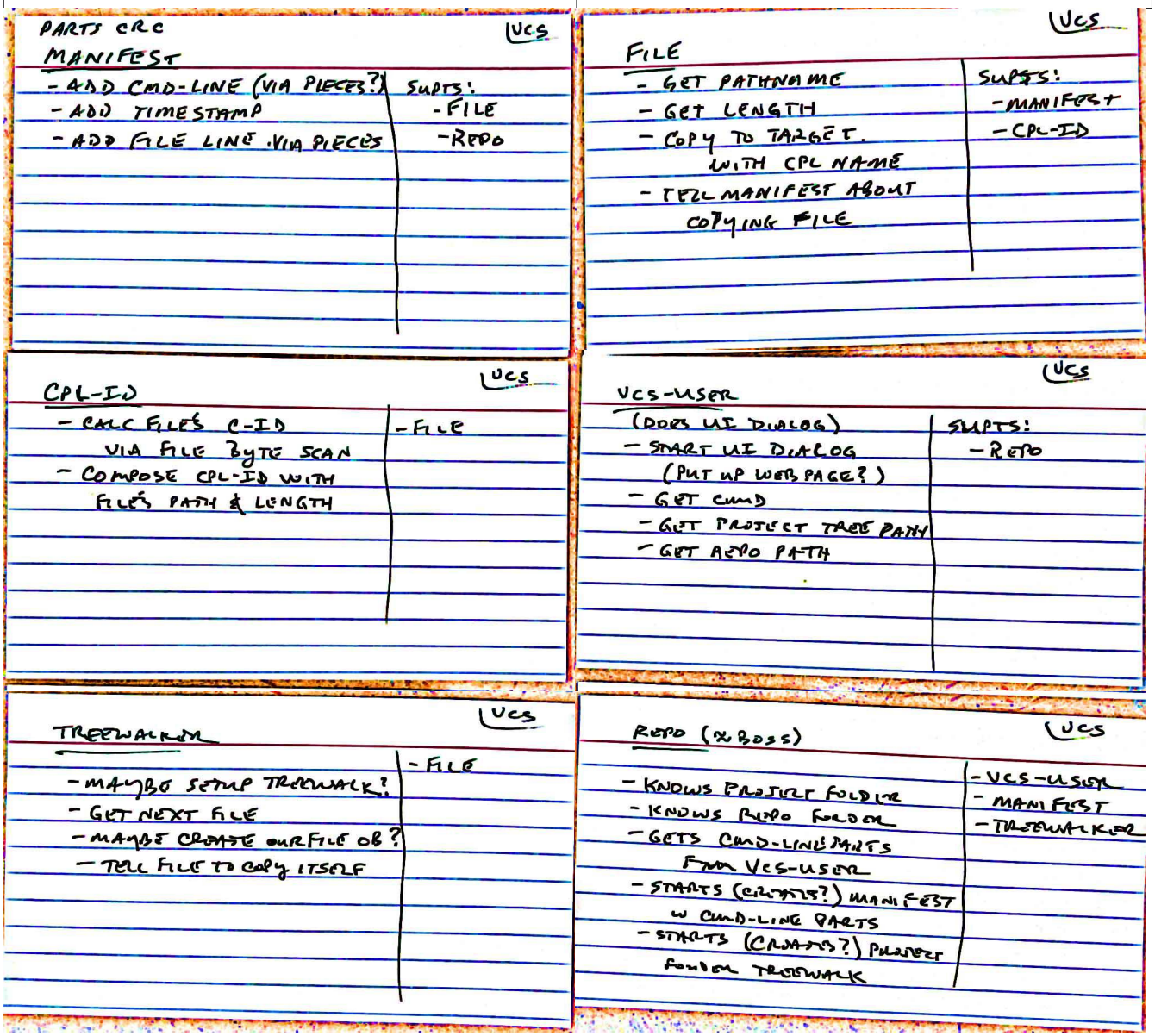
o-**4.8 Environments and Tools**; (Eg, ~Unified Process, UML, IDEs)
o- Easiest stuff already done

o-**4.9 Workstations** (AKA Desktop 'personal' computers)
o- Now, 35yrs, have laptops & 2-,3-head screens, tablets, smart phones, but no improvements to help SWE

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?



Mini-SWE Rules

January 19, 2021

Prelims

** **Reasonable Person Std:** (Mgrs like this – cuz "transparency" – clears "fog of dev" a bit)
o-Submit what you've got on time
o-Ask for help early – not at the last minute

** **Smart Person Std** == Always be ready to show your boss visible progress.
o- The Next Hour; The Next 5 minutes

(*)(*) **S/W's Most Important Thing: Morale --** Maintain Your Morale

Rule #0 (Fast): Get to working S/W Fast
o- **Brute Force Simple:** no frills, use what you know well, go ugly early (and clean once it works)
o- Once you get to working SW, you can easily make it faster – or find out it isn't suitable
o- Always turn in clean-ish code – so others don't think you produce terrible code (unmaintainable)

Rule #1 (Optim): Never Pre-optimize (from Don Knuth, Turing Awd)
o- Don't do things for the future; wait till things are required to take the next small step
o- Always prove that you need to optimization some specific piece of code

Rule #2 (Hunts): Kill Bug Hunts (stop doing SWE that produces hard-to-find RT bugs)
o- 90% of typical dev effort is in **run-time (RT)** bugs
o- WANT: Make RT bugs look like CT bugs (cuz they take < 5 mins to find/fix, usually)
o- HOW: Do Add-a-Trick – always make Trick's results visible on screen (GUI or console log)
o-- Add a trick ("small" piece of code) at a time – then compile, run, test, see it works correctly
o-- And **takes < 5 min to fix any RT bug** – this is how "small"

Rule #3 (Story): The 2-Story Story
o- Top-Story is written in the **U**ser's **Problem Domain Language** (**U**Lang)
o- Bottom-Story is Comp-Sci stuff – user's don't understand this stuff
How? Via **CRC Hand-Sim**
o1. **User-Scenario**: One user role; rambling paragraphs on what user does (tasks) in that role
o2. **Use-Case**: one role-one task: ID agents (user-in-role & others), actions, interaction messages
o3. Write **CRC Card** per Agent: **Class**=agent; **Responsibilities**=actions, **Collaborators**=msg-buddies
o4. **Simulate CRC** Architecture: one card per person, sim the task kickoff, interactions, & results

Rule #4 (EIO): Use Sample EIO – BEFORE you code
o- Write **Expected Input-Output** pairs to drive your detailed designs; based on CRC Sim architecture

Rule #5 (Half-Day): Publish (Half-Day) Goal for your **entire** planned day's effort
o- gives you 100% overrun leeway (Overrun == you under-estimated what you needed)
o- gives you **massive estimation** practice

Rule #6 (Clean): Clean the Page – BEFORE you let your code become visible to others
o- Don't get a reputation for turning in poor quality code
o- Avoid **Lehman's "Mummy" Laws** of S/W Development Life Cycle (**SDLC**) atrophy

Mini-SWE Rules

January 19, 2021

Prelims

** **Reasonable Person Std:** (Mgrs like this – cuz "transparency" – clears "fog of dev" a bit)
o-Submit what you've got on time
o-Ask for help early – not at the last minute

** **Smart Person Std** == Always be ready to show your boss visible progress.
o- The Next Hour; The Next 5 minutes

(*)(*) **S/W's Most Important Thing: Morale --** Maintain Your Morale

Rule #0 (Fast): Get to working S/W Fast
o- **Brute Force Simple:** no frills, use what you know well, go ugly early (and clean once it works)
o- Once you get to working SW, you can easily make it faster – or find out it isn't suitable
o- Always turn in clean-ish code – so others don't think you produce terrible code (unmaintainable)

Rule #1 (Optim): Never Pre-optimize (from Don Knuth, Turing Awd)
o- Don't do things for the future; wait till things are required to take the next small step
o- Always prove that you need to optimization some specific piece of code

Rule #2 (Hunts): Kill Bug Hunts (stop doing SWE that produces hard-to-find RT bugs)
o- 90% of typical dev effort is in **run-time (RT)** bugs
o- WANT: Make RT bugs look like CT bugs (cuz they take < 5 mins to find/fix, usually)
o- HOW: Do Add-a-Trick – always make Trick's results visible on screen (GUI or console log)
o-- Add a trick ("small" piece of code) at a time – then compile, run, test, see it works correctly
o-- And **takes < 5 min to fix any RT bug** – this is how "small"

Rule #3 (Story): The 2-Story Story
o- Top-Story is written in the **U**ser's **Problem Domain Language** (**U**Lang)
o- Bottom-Story is Comp-Sci stuff – user's don't understand this stuff
How? Via **CRC Hand-Sim**
o1. **User-Scenario**: One user role; rambling paragraphs on what user does (tasks) in that role
o2. **Use-Case**: one role-one task: ID agents (user-in-role & others), actions, interaction messages
o3. Write **CRC Card** per Agent: **Class**=agent; **Responsibilities**=actions, **Collaborators**=msg-buddies
o4. **Simulate CRC** Architecture: one card per person, sim the task kickoff, interactions, & results

Rule #4 (EIO): Use Sample EIO – BEFORE you code
o- Write **Expected Input-Output** pairs to drive your detailed designs; based on CRC Sim architecture

Rule #5 (Half-Day): Publish (Half-Day) Goal for your **entire** planned day's effort
o- gives you 100% overrun leeway (Overrun == you under-estimated what you needed)
o- gives you **massive estimation** practice

Rule #6 (Clean): Clean the Page – BEFORE you let your code become visible to others
o- Don't get a reputation for turning in poor quality code
o- Avoid **Lehman's "Mummy" Laws** of S/W Development Life Cycle (**SDLC**) atrophy