The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

csc444h: software engineering I

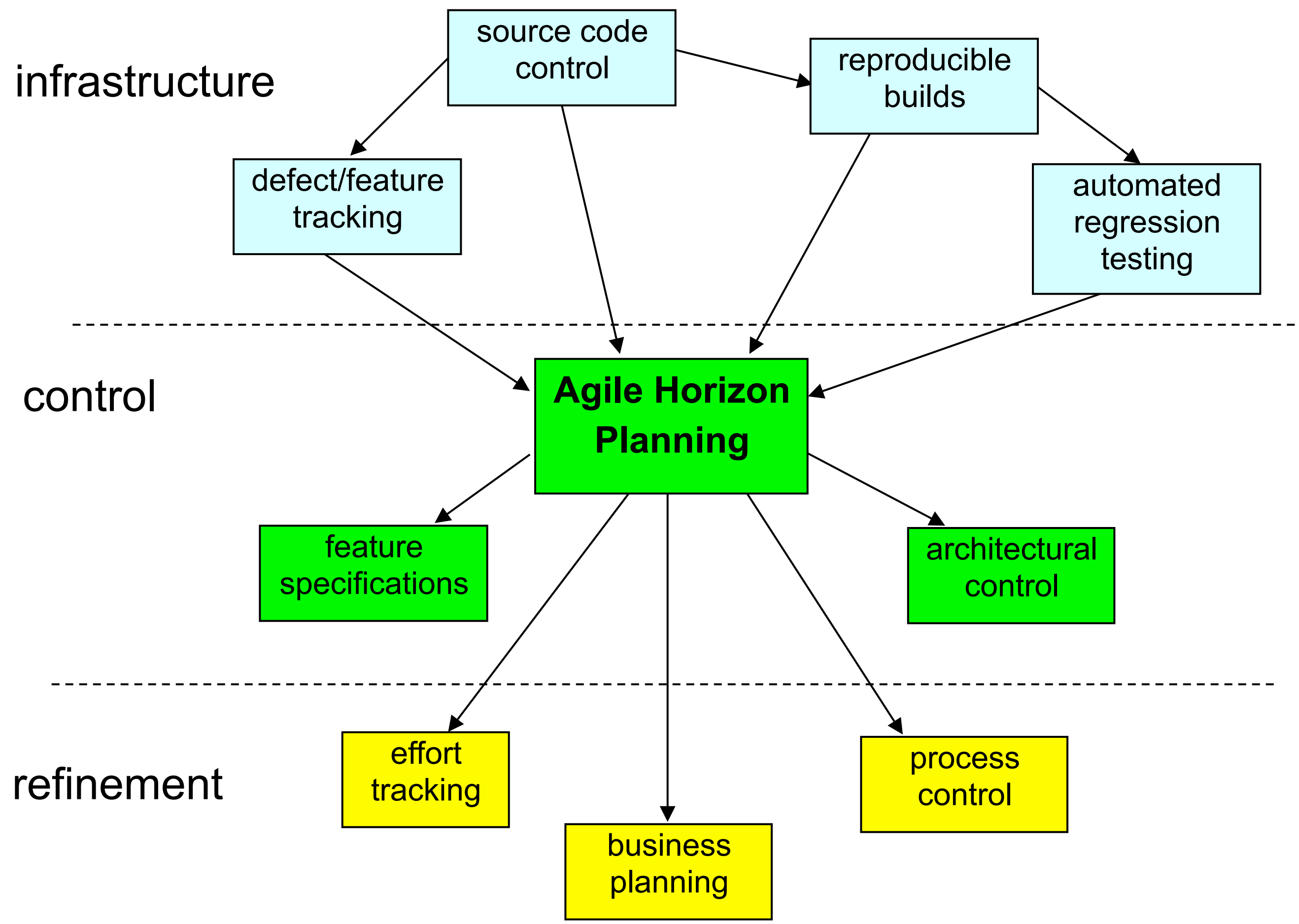
matt medland
matt@cs.utoronto.ca
http://www.cs.utoronto.ca/~matt/csc444

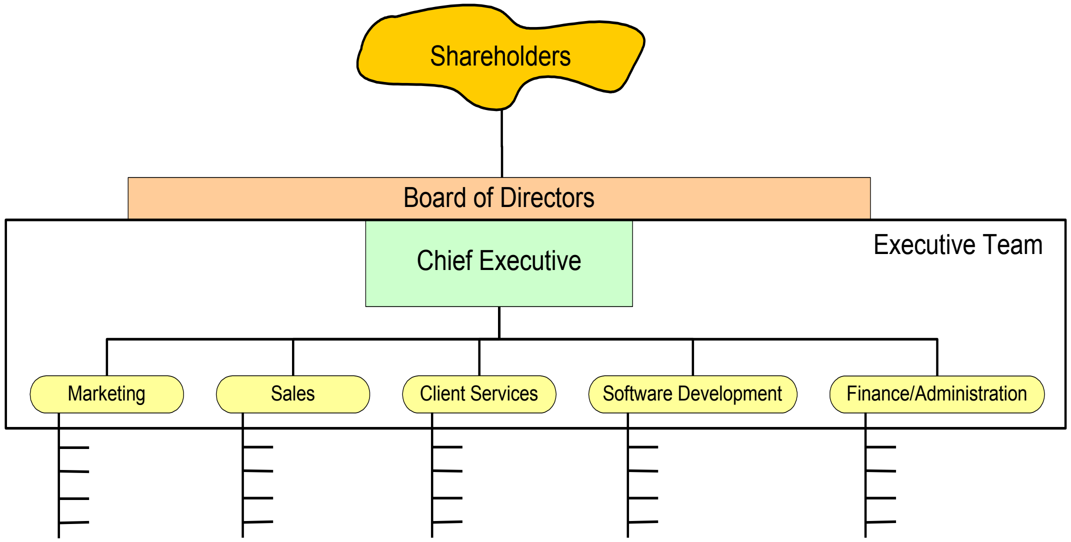
summary

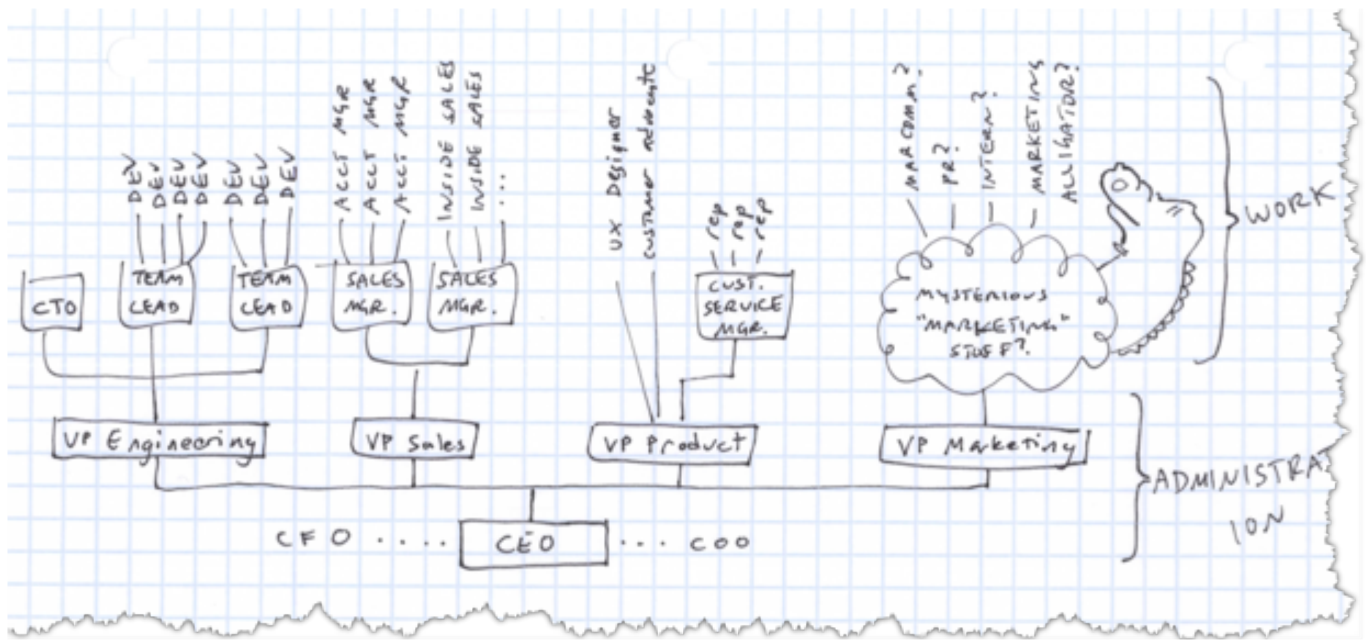
- why do we need a course on software engineering (of large systems)?
  - historically, humans are pretty bad at software engineering
  - lots of spectacular failure examples
  - billions wasted annually

- we discussed what it means for a software system to be considered "*large*"
  - lots of possible choices for metrics
  - chose another definition without metrics:

for our purposes, "*large*" means anything non-trivial that benefits from proper planning and tools, and will be used by someone other than the developer


infrastructure
source code control
reproducible builds
defect/feature tracking
automated regression testing
control
Agile Horizon Planning
feature specifications
architectural control
refinement
effort tracking
business planning
process control


Shareholders
Board of Directors
Executive Team
Chief Executive
Marketing
Sales
Client Services
Software Development
Finance/Administration


DEV
DEV
DEV
DEV
DEV
DEV
DEV
ACCT MGR
ACCT MGR
ACCT MGR
INSIDE SALES
INSIDE SALES
...
UX Designer
customer advocate
rep
rep
rep
MARCOMM?
PR?
INTERN?
MARKETING ALLIGATOR?
CTO
TEAM LEAD
TEAM LEAD
SALES MGR.
SALES MGR.
CUST. SERVICE MGR.
MYSTERIOUS "MARKETING" STUFF?
WORK
VP Engineering
VP Sales
VP Product
VP Marketing
ADMINISTRATION
CFO ....
CEO
... COO

modeling

- models are abstractions
  - often with many details removed
  - used in reverse & forward engineering

- diagrams as models
- UML (& others)
  - structural:
    - class, package, object, component
  - behaviuoral:
    - use case, sequence, state chart

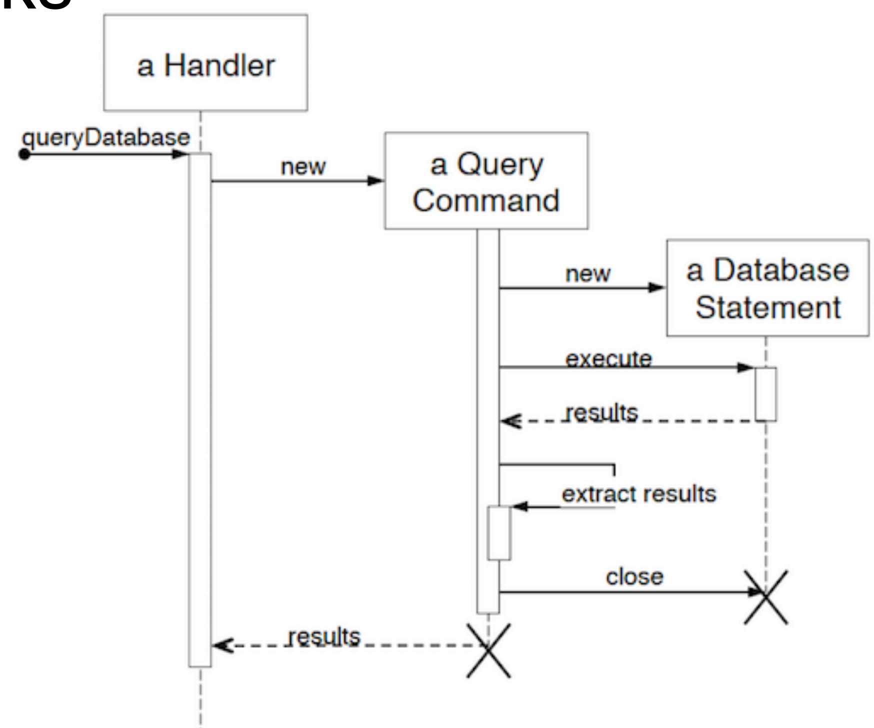
modeling – class diagrams

- standard notation

- visibility
- generalization
- aggregation/composition
- association (& multiplicity)

modeling – sequence diagrams

- used to elaborate use cases
  - good at modeling the tricky bits
  - event ordering & object creation/deletion
- comparing design choices
- assessing bottlenecks



modeling – sequence diagrams (2)

- interaction frames:

Operator	Meaning
alt	Alternative; only the frame whose guard is true will execute
opt	Optional; only executes if the guard is true
par	Parallel; frames execute in parallel
loop	Frame executes multiple times, guard indicates how many
region	Critical region; only one thread can execute this frame at a time
neg	Negative; frame shows an invalid interaction
ref	Reference; refers to a sequence shown on another diagram
sd	Sequence Diagram; used to surround the whole diagram (optional)

modeling – use cases

- use cases
  - flow of events, written from users p.o.v.
  - describes functionality system must provide
  - user stories

- detailed written use case:
  - how the use case starts & ends
  - normal flow of events
  - alternate & exceptional (separate) flow of events

modeling – use cases (2)

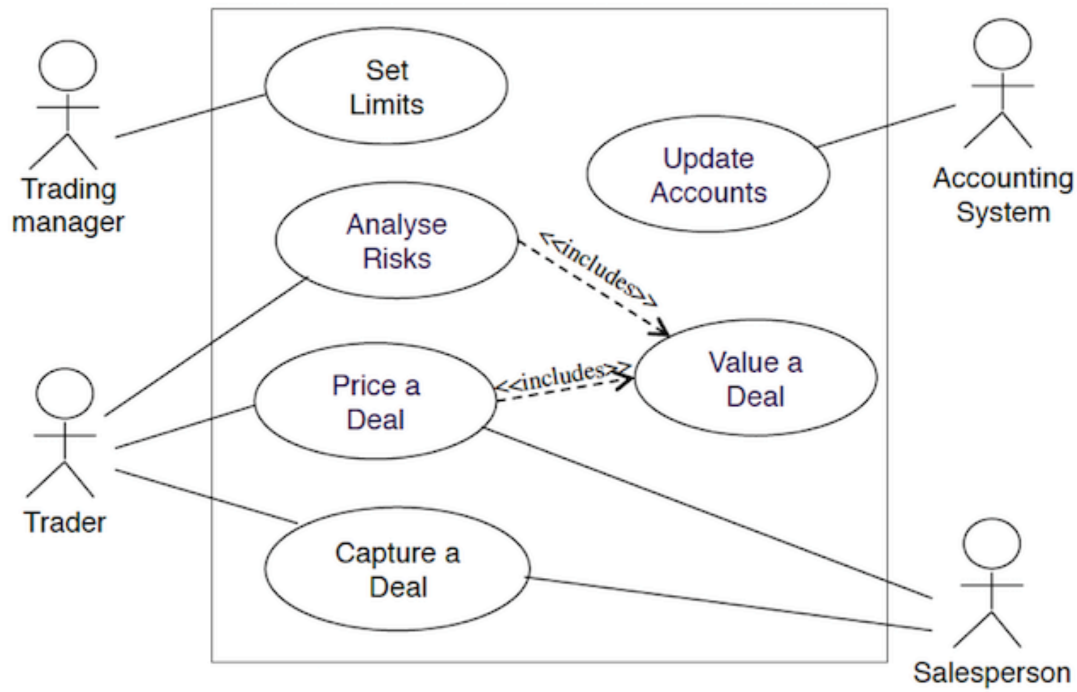
- example

Buy a Product

Main Success Scenario:
1. Customer browses catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address, next-day or 3-day delivery)
4. System presents full pricing information
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming email to customer

Extensions:
3a: Customer is Regular Customer
 .1 System displays current shipping, pricing and billing information
 .2 Customer may accept or override these defaults, returns to MSS at step 6
6a: System fails to authorize credit card
 .1 Customer may reenter credit card information or may cancel

modeling – use cases (2)

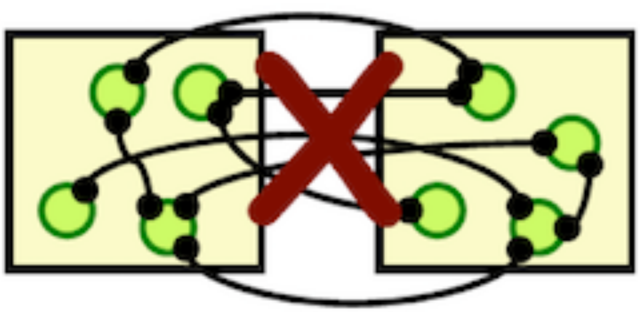


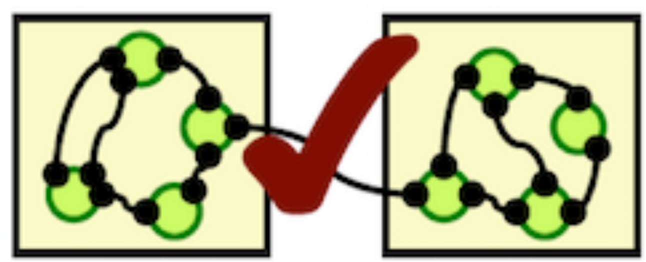
- diagrams
  - actors
  - classes
  - relationships
    - ex. <<uses>>, <<extends>>
    - generalizations

architecture

- components often represented by UML package diagrams

- coupling
  - try to minimize interfaces between modules
  - makes changes, or swap outs, easier
- cohesion
  - strongly interrelated subcomponents





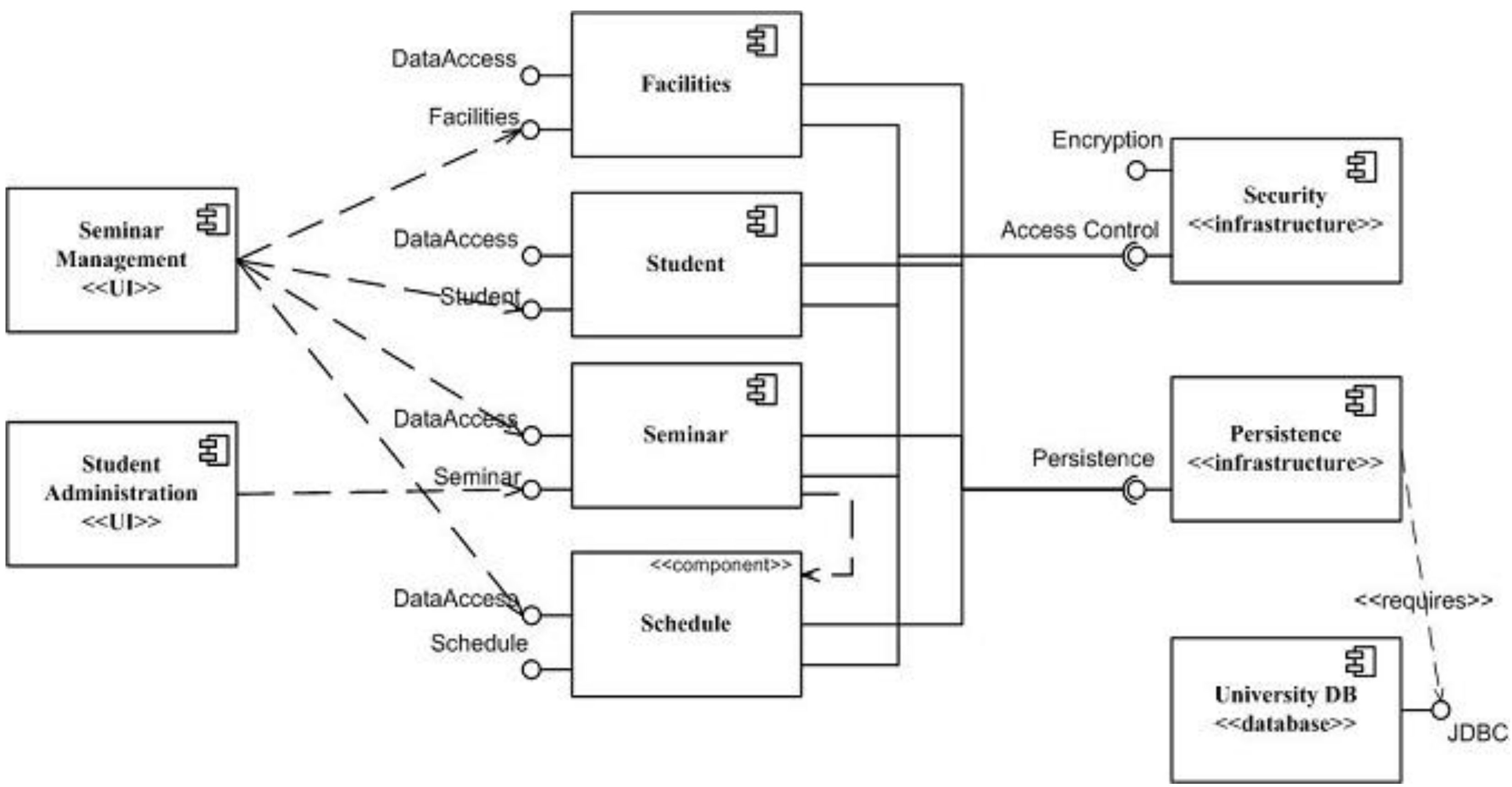
architecture – Conway's law

Conway's law

"The structure of a software system reflects the structure of the organization that built it"

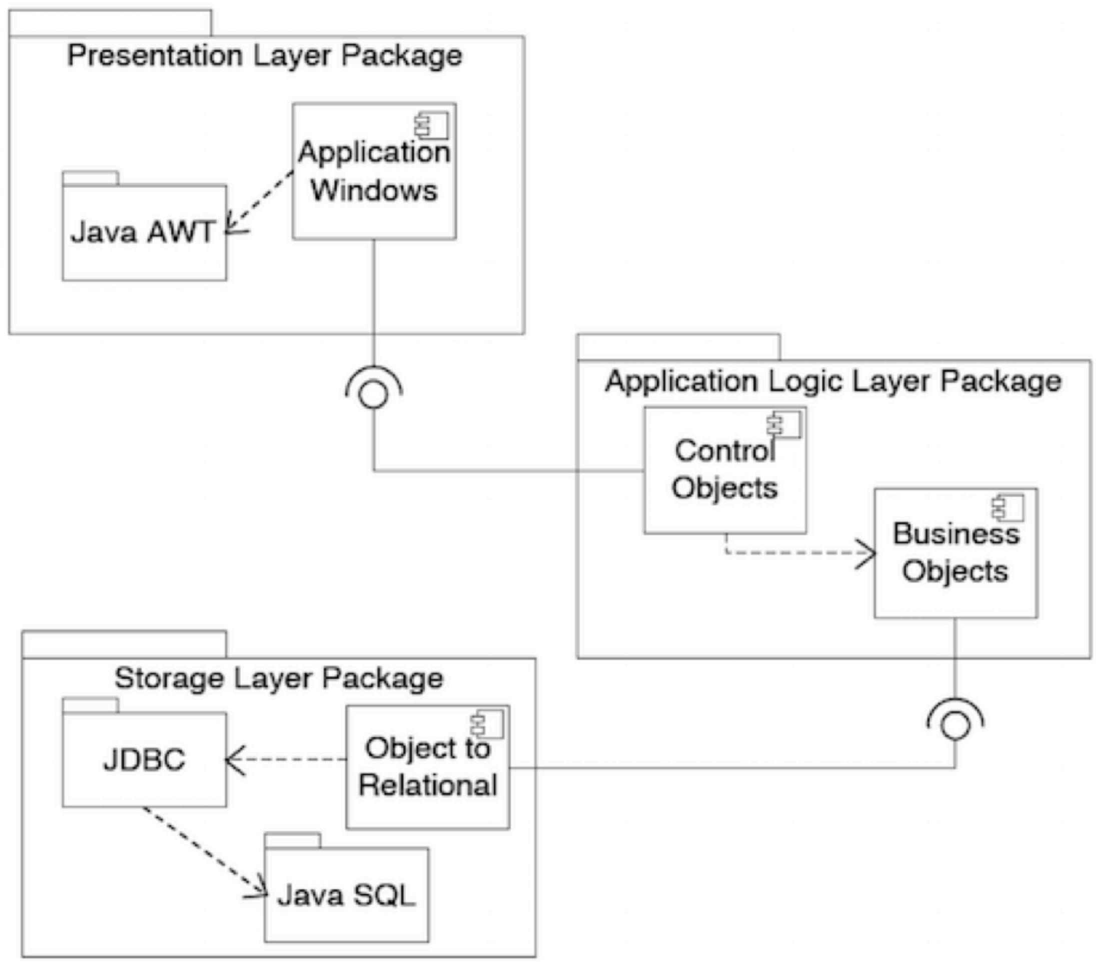
architecture – component diagrams

- alternative to package diagrams



architecture – component diagrams (2)

- or, in combination



architecture – types

- layered:
  - multiple tiers
  - open vs. closed
  - partitioned layers
- pipe & filter
- event-based
- repositories
- MVC (architectural pattern)
- …

reverse engineering

- some reasons software tends to deteriorate over time:
  - not kept up to date with changing needs
  - legacy technology
  - documentation becomes obsolete
  - changing requirements
  - not properly architected for adaptability

reverse engineering (2)

- corrective actions
  - re-documentation
  - design (re)discovery
  - refactoring and reimplementation
- tools range from command-line, to UML diagram auto-generation

SDLC

- what's the goal of a good SDLC?
  - passes all the tests (external quality attributes)
  - good design/architecture (internal)
  - good user experience (quality in use)
  - process quality (can process help ensure product quality)

SDLC (2)

- two main flavors:
  - traditional
    - more rigid
    - little user involvement after spec
    - big-bang releases
  - agile
    - continuous (or frequent) deployment
    - react quickly to changing requirements
    - manifesto & 12 principles

SDLC – agile manifesto

http://agilemanifesto.org/

we are uncovering better ways of developing software by doing it and helping others do it.
through this work we have come to value:

individuals and interactions over processes and tools
working software over comprehensive documentation
customer collaboration over contract negotiation
responding to change over following a plan

that is, while there is value in the items on the right, we value the items on the left more

SDLC – agile dangers

- committing only next sprint
  - doesn't work well for rest of company
  - planning horizon includes multiple sprints
- eliminating comprehensive testing
  - still need a solid testing strategy
- points don't mean much
  - "points" are cute, but meaningless outside R&D
- may find yourself in "cowboy country"
  - may pride yourself on responsiveness to customers, but really just fighting fires

planning

- planning is required when external pressures come to bear on feature availability dates

- common flaws regarding planning
  - making no plans!
  - make a plan, but don't track it
  - attempt to track the plan with inadequate tools

planning (2)

What are we building?

By when will it be ready?

How many people do we have?

- answer these questions, and nothing more
  - not "who will be doing what?"
  - not "what are the detailed tasks required?"
  - not "in what order must the tasks be performed?"

planning

What are we building?

By when will it be ready?

How many people do we have?

the difficult question is:

can we do all 3 at once?

planning – balance sheet

Dates: Coding phase: Jul.1—Oct.1
Beta availability: Nov.1
General availability: Dec.1

Capacity: *days available*

Fred	**31** ecd
Lorna	**33** ecd
…	…
Bill	**21** ecd
total	***317 ecd***

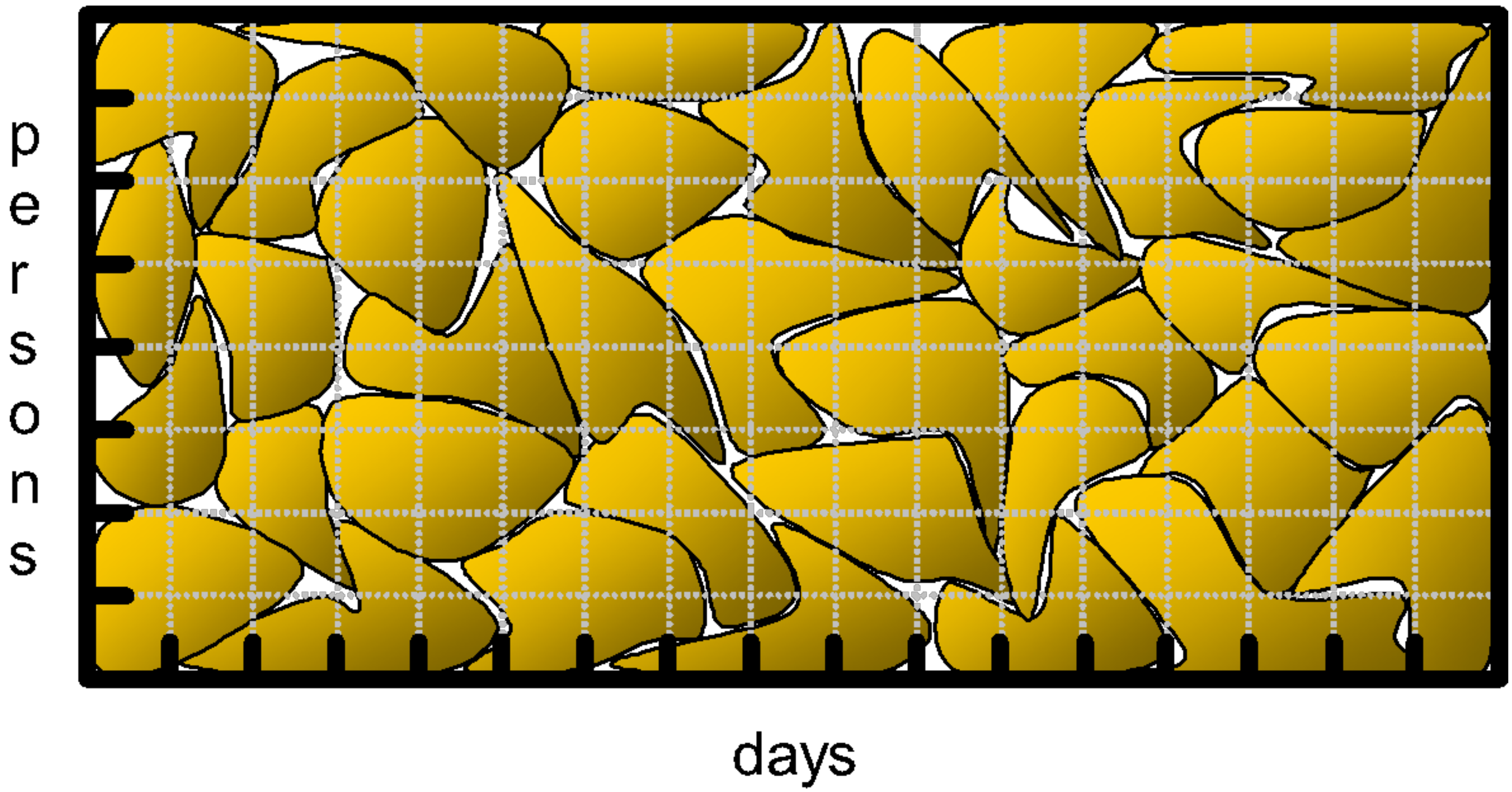
Requirement: *days required*

AR report	**14** ecd
Dialog re-design	**22** ecd
…	…
Thread support	**87** ecd
total	***317 ecd***

Status:

Capacity:	**317 effective coder-days**
Requirement:	**317 effective coder-days**
Delta:	**0 effective coder days**

planning – geometry



everything must fit!

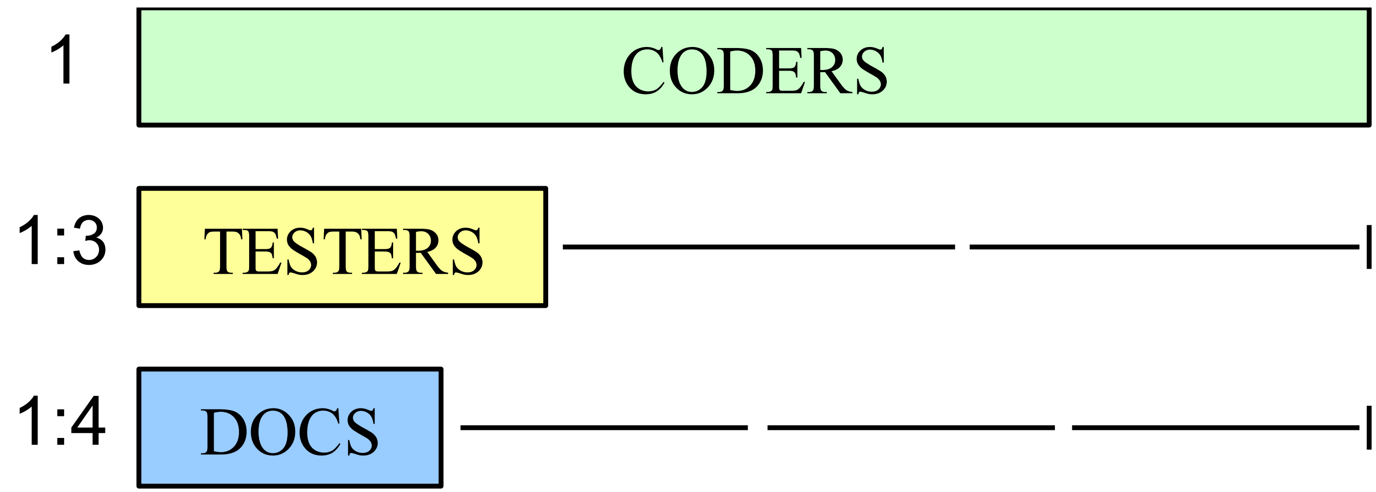
planning – capacity constraint

- what are we building? **F**
- when will it be ready? **T**
- how many developers? **N**

$$\mathbf{F \leq N \times T}$$

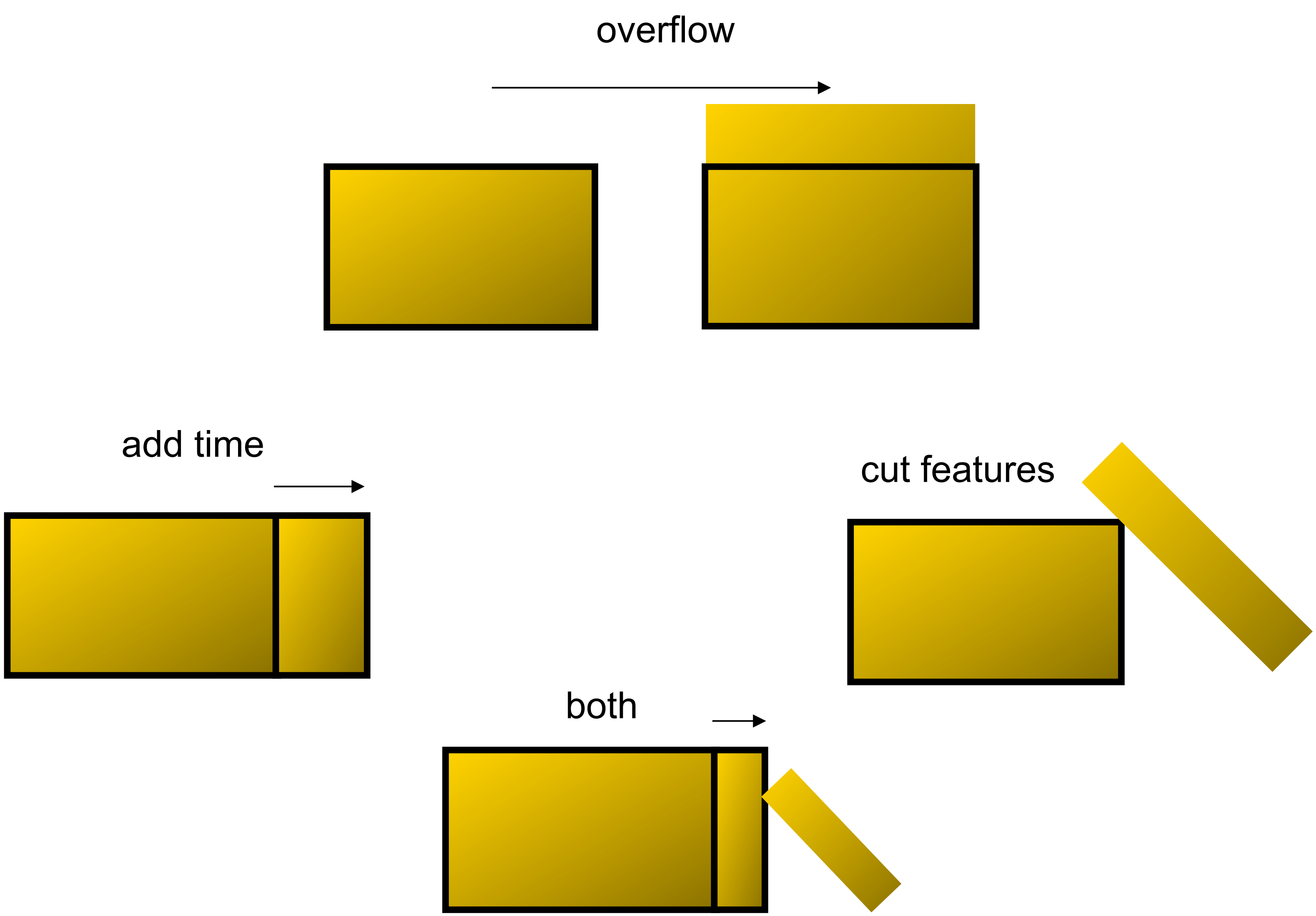
- plan <u>must</u> respect the capacity constraint
- must continuously update the plan to maintain this property

planning – ratios



- typical ratios used in horizon planning
- adjust as necessary
- assumes availability throughout the (overlapping) release cycle.

planning – overflow



risk management

- about risk
  - risk is the possibility of suffering loss
  - risk itself is not bad, it is essential to progress
  - the challenge is to manage the amount of risk
- two parts:
  - risk assessment
  - risk control
- useful concepts:
  - for each risk: Risk Exposure

    RE = p(unsatisfactory outcome) × loss(unsatisfactory outcome)
  - for each mitigation action: Risk Reduction Leverage

    RRL = (RE_{before} – RE_{after}) ÷ cost of mitigating action

risk mgmt. – quantative

- **RRL > 1:** good ROI, do it if you have the money
- **RRL = 1:** the reduction in risk exposure equals the cost of the mitigating action. could pay the cost to fix instead (always?)
- **0 < RRL < 1:** costs more than you save. still improves the situation, but losing $$
- **RRL < 0:** mitigating action actually made things worse! don't do it!

risk mgmt. – qualitative

- risk exposure matrix:

		Likelihood of Occurrence		
		Very likely	Possible	Unlikely
Undesirable outcome	(5) Loss of Life	**Catastrophic**	**Catastrophic**	**Severe**
	(4) Loss of Spacecraft	**Catastrophic**	**Severe**	**Severe**
	(3) Loss of Mission	**Severe**	**Severe**	**High**
	(2) Degraded Mission	**High**	**Moderate**	**Low**
	(1) Inconvenience	**Moderate**	**Low**	**Low**

releases

- releases are expensive
  - marketing collateral
  - launch events
  - training
  - …
- biggest cost is supporting different versions
  - maintenance releases are less costly

releases (2)

- usually need to support 2 or 3 releases
  - try to limit it as much as possible
  - don't do release per customer if at all possible
    - product vs. service, scalability
  - opportunity cost of developers
- time between releases can be important
  - tradeoff: new features vs. costly maintenance
  - never put features in a maintenance release!
    - may result in increase in bug count

releases (3)

- release proliferation
  - buggy releases cause some customers to not upgrade quickly
    - leads to many releases in the field
- if all else fails, and features go into maintenance release, or custom version, or…
  - a really solid regression system may be the only hope

versions

- versions and releases are different
  - versions are different variants of the same software
    - may be very small differences
    - doesn't't apply as much to SaaS, except for client
  - versions have their own maintenance release streams
  - lots of reasons for different versions
    - multiple os support, demos, different hardware, …

versions (2)

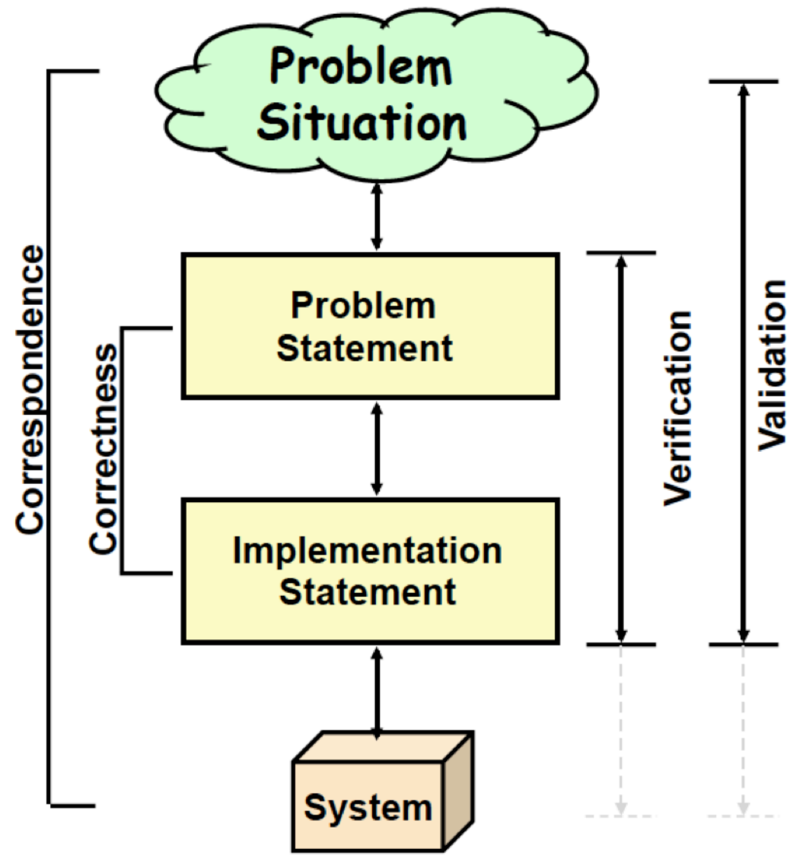
- watch out for version proliferation
  - really need bberry version?
- develop common code, and minimize version-specific code
- custom versions
  - minimize by scripting, configuration, customization, user API, etc.

requirements analysis

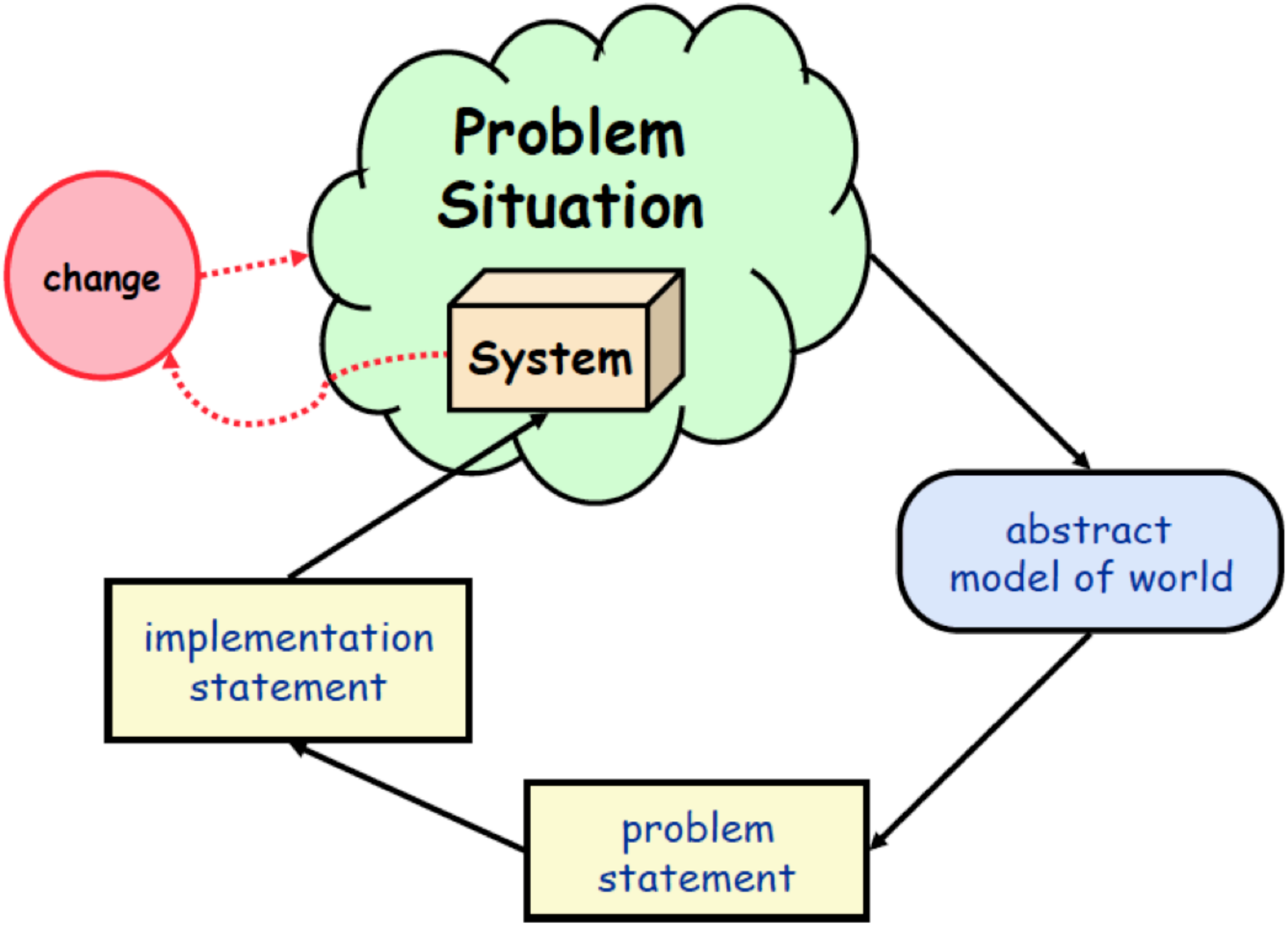
- quality = fitness for purpose
- software is designed for a purpose
  - if it doesn't work, designer got the purpose wrong
- the purpose is found in human activities
- what is the goal of the design?
  - new components, algorithms, interfaces, etc.
  - make activities more efficient, effective, enjoyable
- usually many stakeholders and complex (or conflicting) problem statements
  - may never totally capture spec
  - user participation is essential

requirements analysis (2)

- separate problem desc is useful
  - can be discussed with stakeholders
  - used to eval design choices
  - good source of test cases
  - note: most obvious problem might not be right one to solve
- still need to check:
  - soln correctly solves the problem (verification)
  - problem stmt corresponds to stakeholder need (validation)

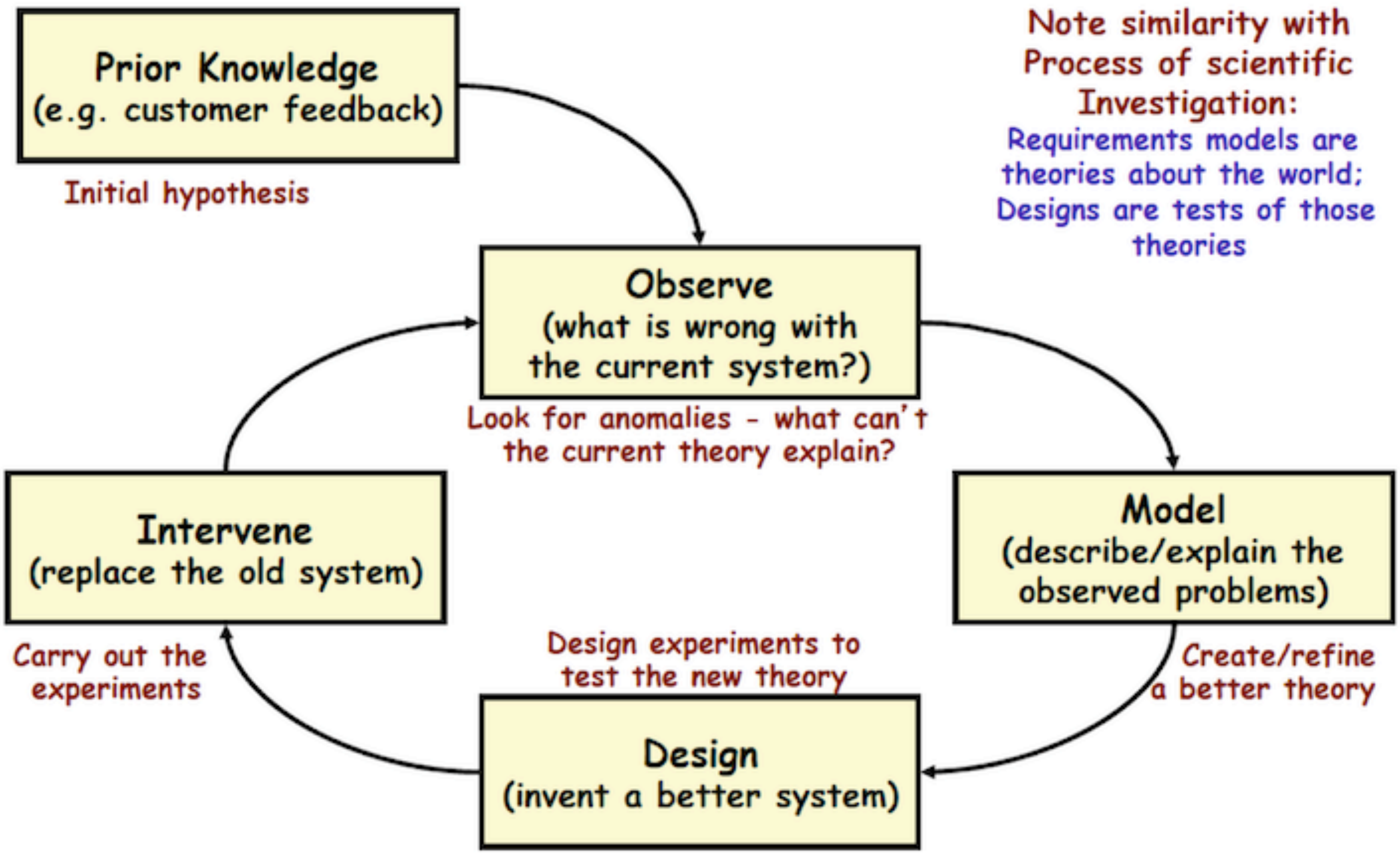


requirements analysis (3)



requirements analysis (4)

- requirements as theories



requirements analysis (5)

- domain properties (assumptions):
  - things in domain that are true regardless if system built
- (system) requirements:
  - things in the application domain we wish to be made true by building proposed system
    - may involve things which the machine can't access
- a (software) specification:
  - a description of behaviours that the program must have to meet the requirements
    - can only be written in terms of the shared phenomena
- S, D $\Rightarrow$ R

requirements to design

- requirements analysis:
  - It's all about (correctly) identifying the purpose

what problem are we trying to solve?

  - answer this wrong and you'll have a quality fail (and all it's associated nastiness)

requirements to design (2)

- what requirements analysts do:
  - which problem needs to be solved? (boundaries)
  - where is the problem? (understand context/domain)
  - whose problem is it? (identify all stakeholders)
  - why does it need solving? (stakeholder goals)
  - when does it need to be solved? (identify development constraints)
  - what might prevent the solution? (feasibility and risk)
  - how might a software system help (collect use cases)

defect tracking

- where it was found
  - product, release, version, hardware, os, drivers, general area
- who found it
  - customer, internal, when
- description of the defect
  - summary, description, how to reproduce, associated data
  - links to related defects or features
- triage
  - severity, likelihood → priority
- audit trail
  - all changes to the defect data, by whom, when
- state
  - state, owner

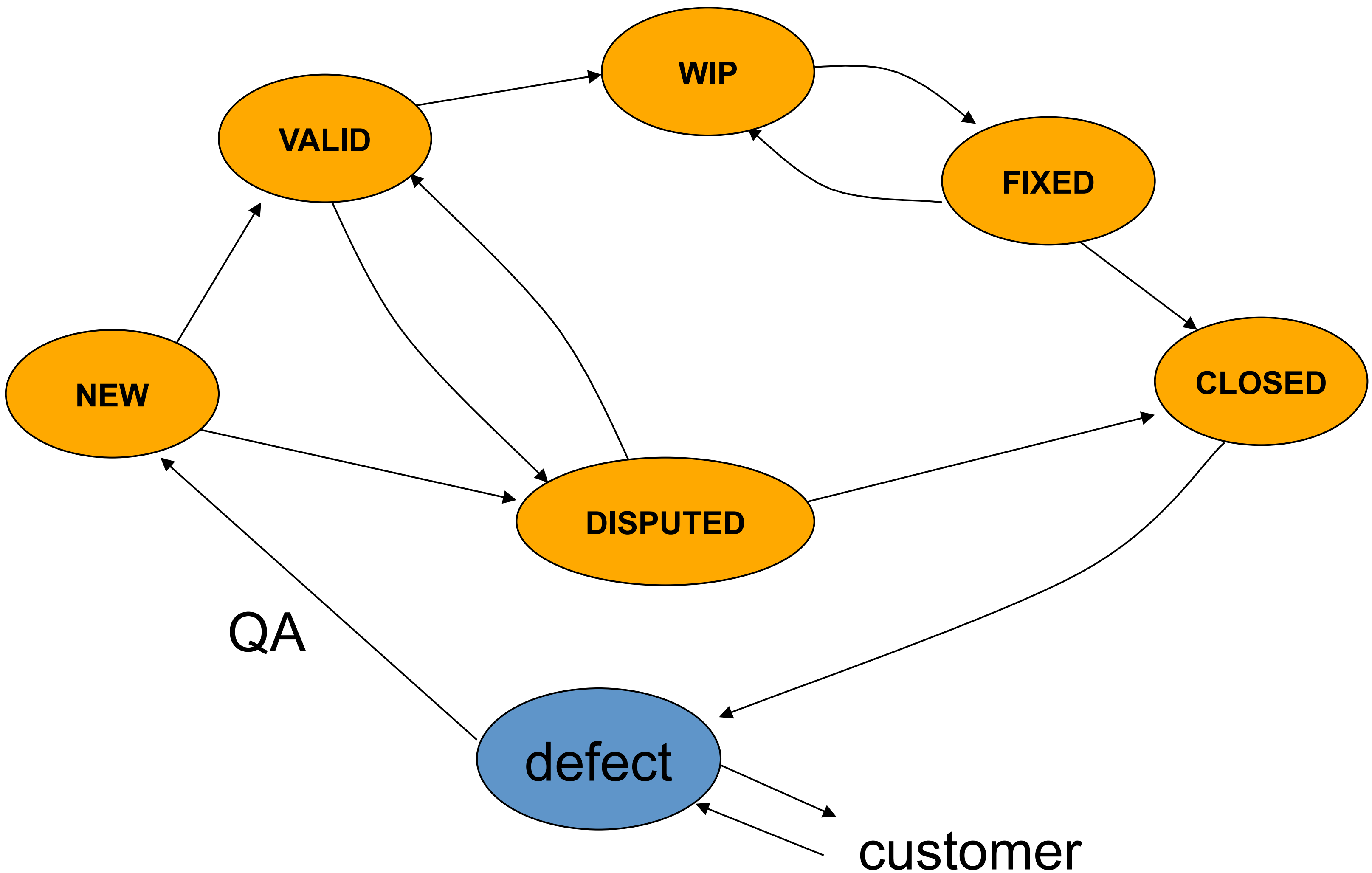
defect tracking (2)

likelihood

priority	low	medium	high
crash, bad data	2	1	1
work around	5	3	2
cosmetic	5	4	3

severity

defect tracking (3)



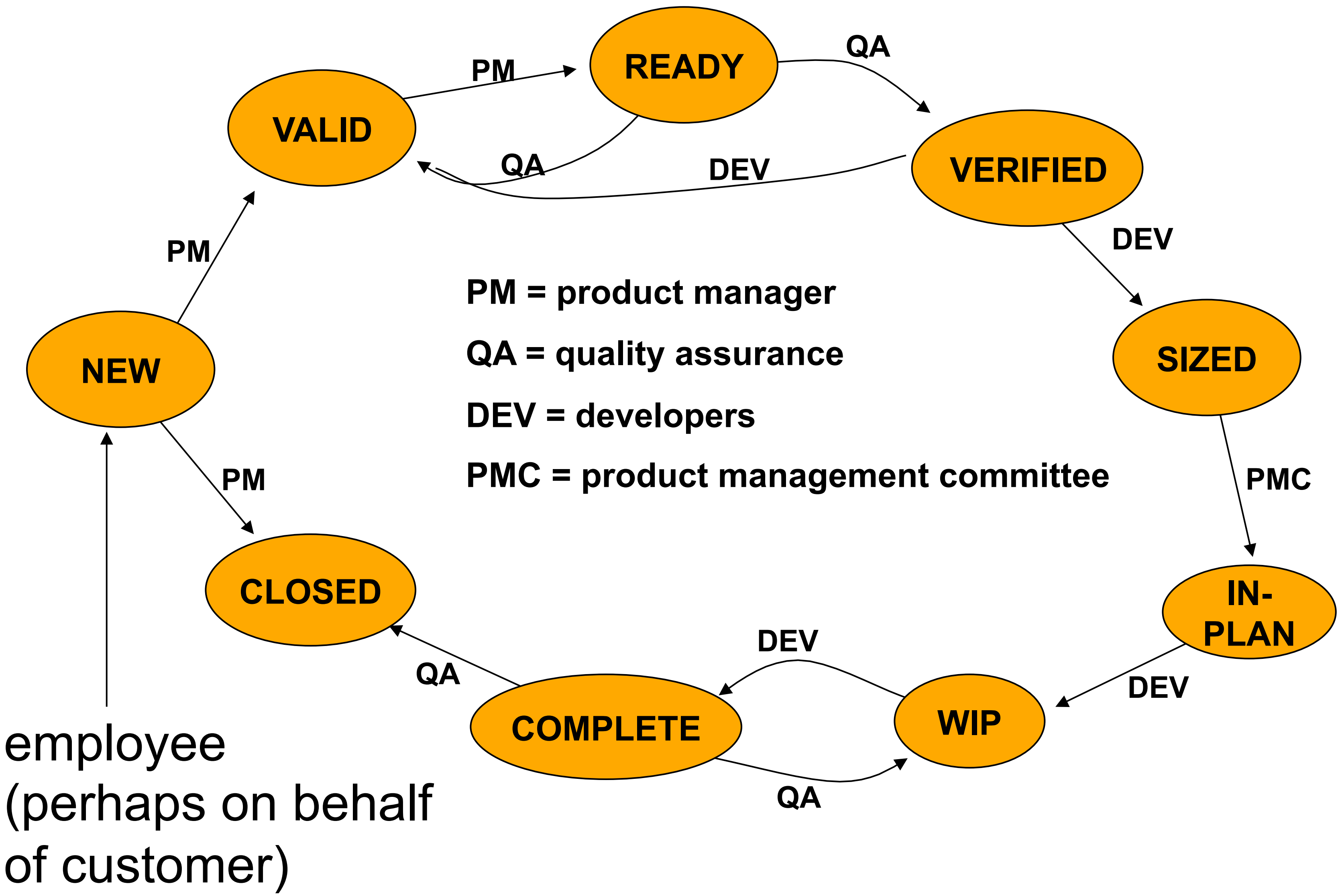
defect tracking (4)

- auto-assigned to developer, or devs pick
- developers can exchange defects
- R&D management needs to:
  - review all defects to:
    - ensure correct priority
    - ensure properly assigned and worked on
    - track trends – arrivals & departures
- system connected to source control
  - helps with attribution
- automated patching to correct severe defects in field

feature tracking

- description
  - one phrase summary, one-paragraph description
  - which product, which area of the product, targeted at which segment?
- who requested it
  - customer, internal, when
  - internal champion
- priority
  - customer desired priority
  - company assigned priority
- target release
  - set once in a release plan
  - set if decided definitely not in the next release
- effort
  - # of ECDs required to implement the feature
- attached documents
  - specification, design, review results, ...
- working notes
  - time stamped notes forming a discussion thread
- process tracking
  - spec required? spec done? spec reviewed? ...

feature tracking (2)



feature tracking (3)

- R&D meets to discuss features in the "in-plan" state
  - specifications written for complicated features
  - UML diagrams for use cases
  - UML sequence diagrams for clarity
  - UML state chart diagrams for clarity
- reviews:
  - specification review before dev starts
  - feature demo meetings
  - design review
  - code review

feature tracking (4)

- effort tracking attached to each feature record
- management reports:
  - features in-plan, spec done, code complete, demo done, acceptance test done, etc.
  - ecds and burn-down charts
    - velocity, ecd delta, expected delay, etc.

testing

- humans are fallible
  - infeasible to completely fix the humans
  - need to double and triple check their work to find the problems
- testing
  - running the software to see if it works the way it is supposed to.
    - works according to specifications
    - ensures specifications are reasonable (that they solve the intended problem)
- correctness proofs

testing (2)

- unit tests
  - performed by developers
  - save and automate for regression
- functional test (black box)
  - performed by QA on single features
  - starts before feature complete
- integration test
  - after all features have been finished
  - whole system works together
  - problems here are logged as defects

testing (3)

- test-driven development (TDD)
  - before feature is written devise all test cases
  - implement all tests with whatever automated tool you are using
    - tests will all fail because the feature code is not written yet
  - write the feature code
  - check that all tests now pass
  - unit tests developed in first step are saved as regression system and run automatically

testing (4)

- performance regression
  - keep performance statistics on the regression run for trending
  - functionality may be fine, but performance not

- memory leak regressions
  - specialized software can check
  - less important in managed code (with gc)
    - even harder to correct in this scenario, usually a runtime system bug

regression testing

- locks-in quality
  - once you achieve quality, you don't backslide
  - everybody focuses on new features and forgets the old
- finding defects sooner
  - finds the defect nearest the point in time it was injected
  - freshest in the coder's mind
  - least expensive time to fix it
- development aid
  - can work on complex, central bits of the code without fear of breaking something major on not finding out
- releasing
  - if need a last minute critical defect fix to release
  - if no/poor automated regression, might have to delay until re-tested

regression testing (2)

- coverage is a measure of how much of the system is exercised by the regression tests
  - all functions
  - every line of code
  - all conditions
  - overridden and overriding methods
  - …

- GUI regression testing is hard
  - tools can help
  - minor layout changes can mess it up
  - can use an API to simulate as close to UI as possible

effort estimation

- estimates are imprecise
  - optimistic? pessimistic? some confidence level?
- many techniques
  - three-point estimates, function points, etc.
- confidence intervals
  - ***T*** is fixed, ***F*** & ***N*** are stochastic variables
  - $D(T) = N \times T - F$ (is the delta)
  - compute normal curve for ***D(T)*** and select ***T*** such that desired confidence is achieved
    - repeat with different feature set ***F*** if ***T*** is fixed
  - shortcut is to estimate at 80%, and 50% (average), then fit normal and predict $P(D(T)) < 0$

the end

good luck on the exam!