Computer Science
UNIVERSITY OF TORONTO

lecture 22
course review

csc302h

winter 2014

course review

- hope the pizza is here by now!
- we covered a lot in this course!
- why do we need a course on *engineering large software systems*?
  - historically, humans have been pretty bad at it!
  - billions are wasted annually on failed or over-budget software projects

course review (2)

- 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

review – modeling

- we build models to help:
  - during design
  - to analyze existing systems (reverse engineer)
  - to help us communicate
- models are abstractions
  - help us focus on important aspects, not blinded by the details
  - decomposition, modularization, association

review – architecture

- avoid unnecessary coupling & cohesion
- if a layered approach, what are the layers? what goes in each?
  - following a pattern like MVC, MVP?
- modularize for reusability (well designed public interface)

review – architecture (2)

- Conway's law re. software structure & communication structure
- common architectures
  - layered:
    - open vs. closed, n-tier, partitioned
  - others: broker, client-server, event-based, repository (hub), MVC
- UML
  - package & component diagrams

review – software evolution

- E-type, S-type, P-type systems
- Lehman's laws of program evolution
  - continuing change, increasing complexity, self-regulating, conservation of; organizational stability, familiarity

- maintenance, rejuvenation, refactoring

review – sequence diagrams

- modeling software behavior with sequence diagrams
  - UML collaboration diagram captures control flow, sequence is a different rendering
  - emphasis on time. objects on top, arrows are "messages", time is vertical
  - interaction frames (alt, opt, loop, par, …)
- when to use sequence diagrams?
  - discussing design options
  - elaborating on use cases

review – use case diagrams

- capture system requirements
- show how users interact with a system
- short phrase to sum up a distinct piece of functionality
- “actors” (stick ppl) show a role that a user takes on during an interaction
- each use case has one or more actors
- relationships between use cases like «extends», «uses», «includes»
- reverse engineering use cases to describe a system

review – SDLC

- agile vs. [traditional | planning-based | sturdy | disciplined]
- what do they share? how are they different?
- which is better?
  - both & neither – depends on context
- SDLC models covered: waterfall, prototyping, phased, spiral, RUP, SCRUM, XP
- Gantt charts as a (bad) planning tool

review – risk management

- risk exposure:
  - RE = probability × consequences (loss)
- risk reduction leverage
  - RRL = (RE_{before} – RE_{after}) ÷ cost of mitigating action
- risk assessment
  - quantitative (if you can)
  - qualitative (risk exposure matrix)
- don't have independent V&V report to the development manager (conflict of interest)
- IV&V comes out of separate budget

review – requirements analysis

WHAT PROBLEM ARE
WE TRYING TO SOLVE?

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

- requirements change over time
- requirements can be incomplete
- therefore, requirements analysis is on-going and iterative

review – requirements analysis (2)

- so, how do we solicit requirements? where do they come from?
  - identify the stakeholders
  - identify the goals of the stakeholders
  - then think very hard about:

WHAT PROBLEM ARE

WE TRYING TO SOLVE?

  - it may not be what you were told

review – requirements analysis (3)

- from requirements to production

- D: consists of assumptions, or, truths
- R: are the wants, the things that solve the problem
- S: is the bridge, what must the system do to satisfy R
- P: is the program that satisfies S
- C: is easy, buy it, rent it, or it lives in the cloud, or…

- in general: S (given D) ⇒ R

review – robustness analysis

- bridge between use case and more technical things like sequence diagrams & code
- skipped if you don't need it
- used to:
  - analyze logic of a use case
  - ensure use cases are "robust" in that they really do represent the usage requirements
  - identify objects & responsibilities
  - visualize the things will build (i.e. code)
  - communicate (almost) technical stuff to stakeholders

review – V&V

- validation, or high-level testing, is usually preformed by doing "dynamic testing"
  - unit tests (less so, more for verif.), integration tests, system tests, acceptance tests (UAT)
- validation: "are we building the right thing?"
- verification: "are we building the thing right?"
- test cases are written for verification & run for validation.
- range of activities:
  - mission critical: may use formal methods (proofs)
  - latest fart app, probably not so much :)

review – testing

- multiple causes for defects: missing requirements, spec. error, bad design, bad algos, bad developers!
- defects (may) lead to failures, or go unnoticed.
- removing defects earlier is cheaper, sometimes by orders of magnitude!
- defect detection strategies & effectiveness:
  - formal design inspections & testing – 95% (ish)
  - agile informal review & regression – 90% (ish)
  - different costs, both useful depending on context

review – testing (2)

- some characteristics of good tests:
  - power: bug exists, test will find it
  - validity: no false-positives
  - non-redundancy: provides new information
  - repeatability: easy to re-run
  - etc. (don't memorize, but refer back when coming up with test plan)

review – testing (3)

- type of test, to what it applies, & what it is testing:
  - unit: unit of code, tested separately, generally applies to single use case or part of
  - integration: many (or all) units together, tests that code meets design specs.
  - Functional test: coverage of all inputs (inc. edge/ corner cases), tests functional req's.
  - performance: tests (one of the) quality requirements
  - acceptance: customer goals
  - installation: user environment (optional depending on context)
  - …

review – testing (4)

- structural testing (a.k.a. white-box testing)
  - should be called *"clear-box"* testing
  - based on structure of code
  - coverage == all paths through code tested
- functional testing (a.k.a. black-box testing)
  - can't see inside
  - test cases derived from use cases
- other types of testing:
  - dataflow, boundary, usability, acceptance, exploratory, interference, etc.

review – testing (5)

- test driven development (TDD)
  1. developer writes (initially failing) unit tests
  2. then, write minimum code to pass unit test
  3. then refactor (i.e. write more code) to meet full specification
- automated testing
- coverage
  - both functional and structural (& behavioral, inheritance)

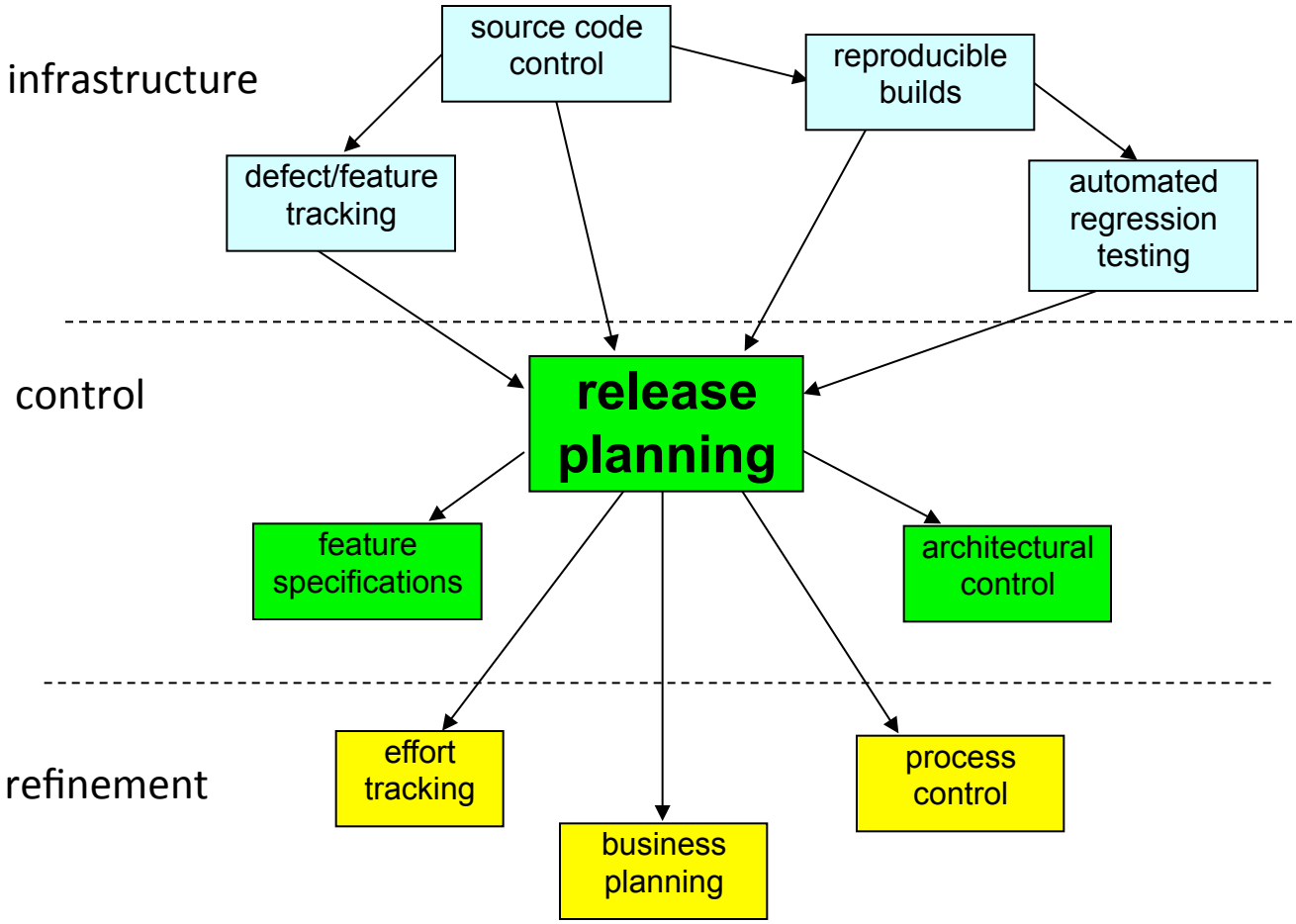
review – static analysis

- static (program) analysis refers to the analysis of a program's source code.
- various tools for various programming languages
- increasingly, the IDE is performing static analysis for us on the fly
- tools look for things like:
  - null dereference or null assignment
  - array index out of bounds
  - other runtime errors not caught by compiler
- lots of false positives & negatives

review – quality

- what is quality?
  - value to some person
  - fitness to purpose
  - exceeding the customer's expectations
  - …
- quality assurance focuses on the quality of the process (V&V on quality of product)
- quality frameworks
  - six sigma (6σ)
  - capability maturity model (CMM)
  - …

review – top-10 essential practices



review – release planning

- a.k.a. (software) development planning, agile horizon planning, etc.

What are we building?
By when will it be ready?
How many people do we have?

- can we do all 3 at once?

review – release planning (2)

- the essence of planning is uncertainty
  - react to changes; both internal & external
- what goes wrong if you don't plan
  - crossing the chasm
- why plan? – external pressures
- with good data, good managers can make good decisions.

review – release planning (3)

- requirements (or features = *F*)
  - prioritized potential requirements from wish list
    - can even do full cost / benefit analysis
  - estimate a size for each (planning poker) in ideal days (ECDs)

- calculate available resources (*N*)
  - pick a value for *T* (workdays until release date, end of sprint, horizon, …)
  - for each developer, determine availability, vacation, and then multiply by *w*

review – release tracking

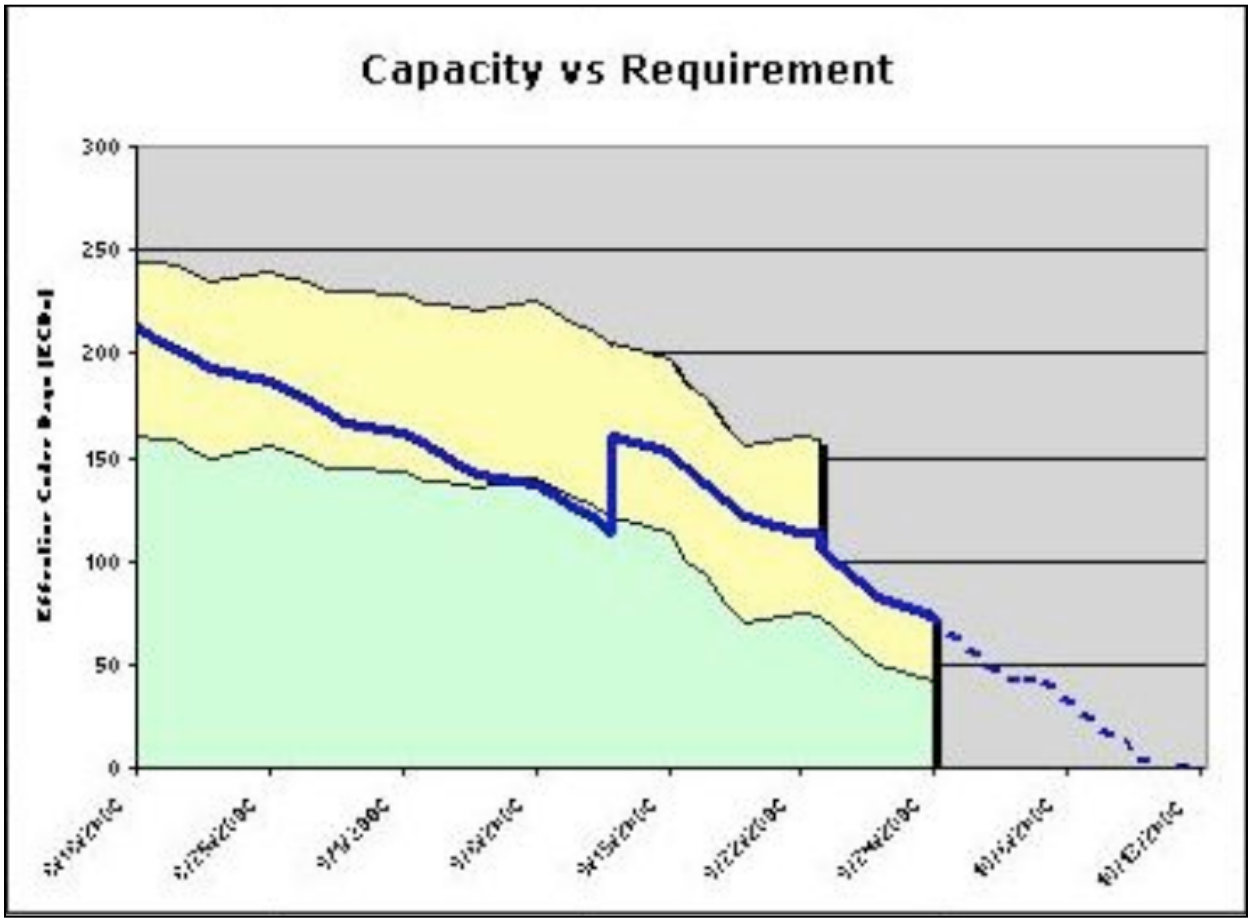
- the capacity constraint

$$F = N \times T$$

- plan must respect the capacity constraint
- keep plan up to date with most current estimates at all times
- dealing with overflow
  - move dates
  - cut features
  - Combination
  - adding developers is rarely help to current plan

review – release tracking (2)

- burndown charts to show velocity over time



review – estimation

- most important points:
  - estimates are not saying exactly how long you think something will take (by definition)
  - they are stochastic variables, we model them with a normal distribution
  - we can use confidence intervals to determine how likely we are to meet deadlines
    - estimate with a given confidence interval in mind (ex. 80%)

Fin