# CPSC 310 – Software Engineering

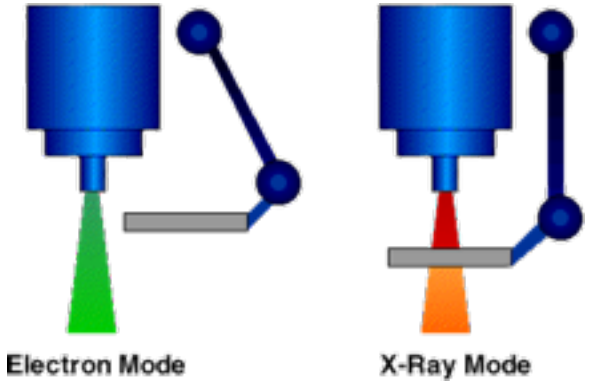
# Quality

# Learning Goals

By the end of this unit, you will be able to:

- Describe aspects that affect software quality other than code quality
- Explain the benefits of high quality code
- Explain why we can't sufficiently measure code quality with testing alone
- Describe mechanisms for improving code quality (code reviews, pair programming, refactoring, software metrics)

# Therac-25




- Computerized radiation therapy machine
  - Shallow tissue: direct electron beam
  - Deeper tissue: electron beam converted into X-ray photons
- accidents occurred when high-energy electron-beam was activated without target having been rotated into place; machine's software did not detect this
- First case in 1984: lawsuit but manufacturer refused to believe in a malfunction of Therac-25
- Second case in 1985: display indicated "no dose" so operator repeated 5 times; patient died 3 months later
- Overall six accidents with ~100 times the intended does between 1985 and 1987; 3 patients died

See more at http://courses.cs.vt.edu/cs3604/lib/Therac_25/Therac_1.html

# Therac-25: some problems

- The design did not have any hardware interlocks to prevent the electron-beam from operating in its high-energy mode without the target in place.
- The engineer had reused software from older models. These models had hardware interlocks and were therefore not as vulnerable to the software defects.
- The hardware provided no way for the software to verify that sensors were working correctly.
- The equipment control task did not properly synchronize with the operator interface task, so that race conditions occurred if the operator changed the setup too quickly. This was evidently missed during testing, since it took some practice before operators were able to work quickly enough for the problem to occur.
- The software set a flag variable by incrementing it. Occasionally an arithmetic overflow occurred, causing the software to bypass safety checks.

from: Stephen Dannelly

# Therac-25

Many factors:

- Programming errors / race conditions
- No independent review of software
- Inadequate risk assessment  together with overconfidence in software
- Therac-25 software and hardware combination never tested until assembled at the hospital
- poor human computer interaction design
- a lax culture of safety in the manufacturing organization
- management inadequacies and lack of procedures for following through on all reported incidents

# What is Software Quality?

According to IEEE

- The degree to which a system, component or process meets the ***specified requirements***.
- The degree to which a system, component or process meets the ***customer or user needs and expectations***.

# What is Software Quality?

According to Roger Pressman

- Conformance to explicitly stated functional and performance ***requirements***, explicitly documented ***development standards***, and ***implicit characteristics*** that are expected of all professionally developed software.

# Software Quality Attributes

- ISO9126:
  - Functionality: the ability of the system to do the work for which it was intended, incl Security.
  - Reliability: can it maintain performance?
  - Maintainability: can it be modified?
  - Efficiency: performance and resource consumption.
  - Usability: effort needed to use the system.
  - Portability: can the system move to other environments?
- Quality can be process, internal, external, or 'in-use'

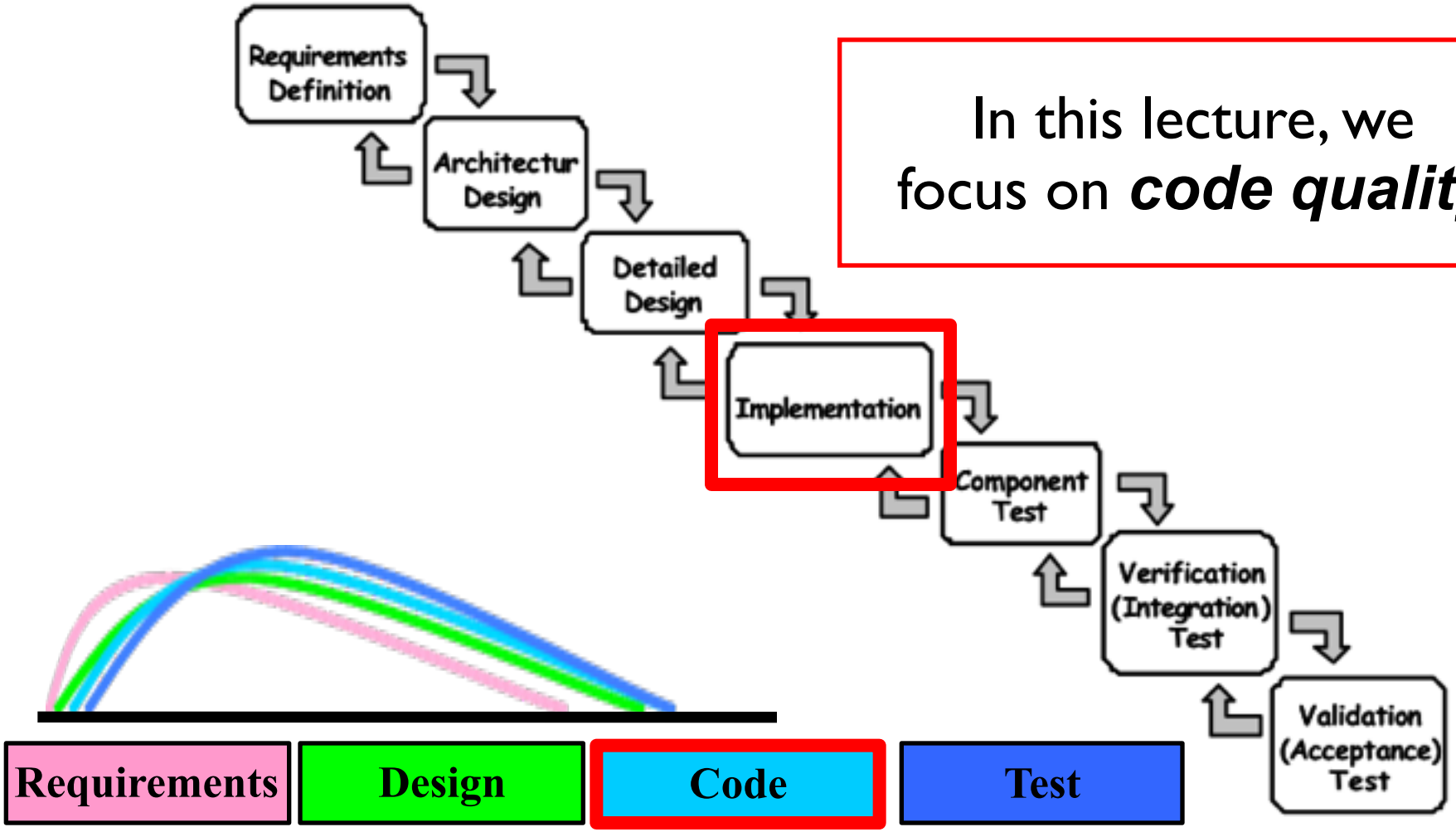
# Overall Quality

Quality is a chain: good process → good internal quality → good external quality → happy customer

Assessing quality:

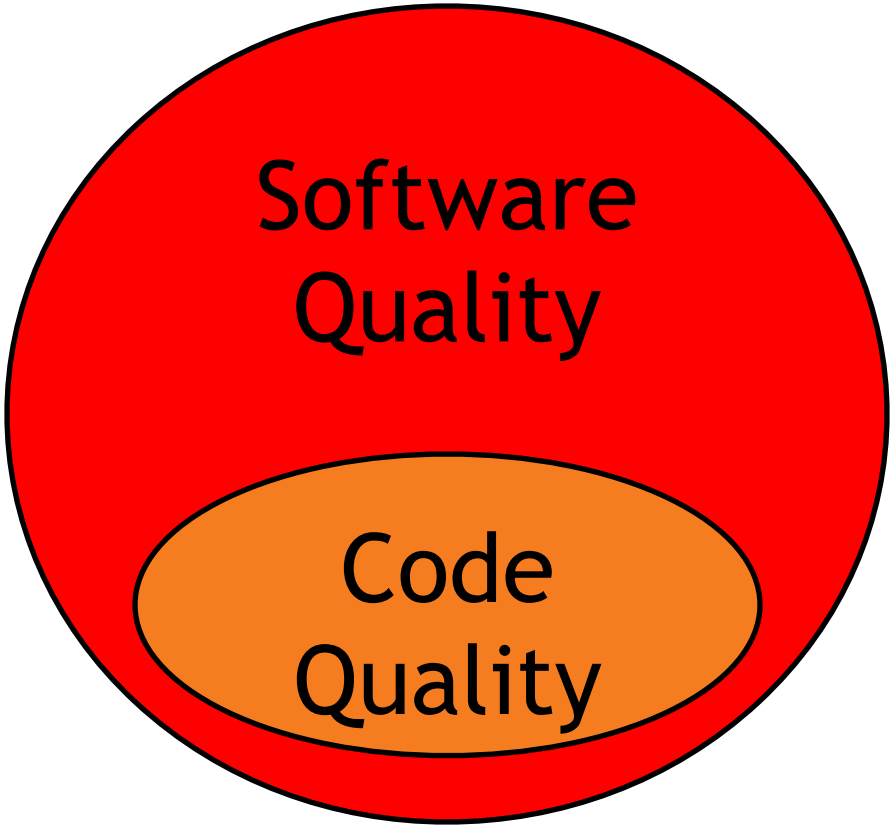
- Quality Assurance (QA): test the process quality (CMM, ISO9000, TQM, etc)
- (Independent) V&V
  - Verification: did we build it right? *internal*
  - Validation: did we meet requirements? *external*

# Code Quality




In this lecture, we focus on ***code quality***

# Not the only element of Software Quality



# Other elements of Software Quality

- Faulty definition of requirements
- Client-developer communication failures
- Logical design errors
- Shortcomings of the testing process
- Procedure errors
- Time management problems
- …

# Joel Test: 12 steps to better code

1. Do you use source control?
2. Can you make a build in one step?
3. Do you make daily builds?
4. Do you have a bug database?
5. Do you fix bugs before writing new code?
6. Do you have an up-to-date schedule?
7. Do you have a spec?
8. Do programmers have quiet working conditions?
9. Do you use the best tools money can buy?
10. Do you have testers?
11. Do new candidates write code during their interview?
12. Do you do hallway usability testing?

see http://www.joelonsoftware.com/articles/fog0000000043.html

# An Example

```
char b[2][10000],*s,*t=b,*d,*e=b+1,**p;main(int
c,char**v)
{int n=atoi(v[1]);strcpy(b,v[2]);while(n--)
{for(s=t,d=e;*s; s++){for(p=v+3;*p;p++)if(**p==*s)
{strcpy(d,*p+2);d+=strlen( d); goto x;}*d++=*s;x:}
s=t;t=e;e=s;*d++=0;}puts(t);}
```

Is there anything wrong with this code?

# Ingredients in low-quality code

What could you do to make sure your code was bad?

# Recipe for a Disaster

- Ignore what the customers say they want – the developers surely must know better.
- Put in all the features that could potentially ever be useful.
- ***Do not worry about quality aspects (and ignore the related practices) until the deadline approaches.***
- Do not waste time on design or documentation – after all, code is the most important thing and time is already too short to do all that needs to be done.

# Some of the Major Mechanisms for Quality Code

- Cultural mechanisms
  - Teamwork / Team-Building
  - Organizational Values
- Human mechanisms
  - Code Reviews
  - Refactoring
- Automatic mechanisms
  - Style checkers
  - Quality Metrics

# Cultural Mechanisms

- Teamwork / Team-Building
- Organizational Values

# Teamwork / Team Building

- "No matter what the problem is, it's always a people problem." - *Jerry Weinberg*
- Techniques
  - Ice-breaker
  - Personality test
  - Casual meetings
  - Inclusive teams
    - Open communication
    - Transparent decision making
  - Foosball?

# Organizational Values

- "The structure of a computer program reflects the structure of the organization that built it."
  - *Conway's Law*
- Rigid hierarchical structure
  - Decisions are handed down, no ability to dispute
  - Less input into each decision, less motivation?
  - Less discussions could lead to faster decisions (although…)
- Flexible, collaborative, team-based structure
  - Better decisions through collaboration
  - Different people focus on different issues, cover all bases

http://research.microsoft.com/apps/pubs/default.aspx?id=70535

# Human Mechanisms

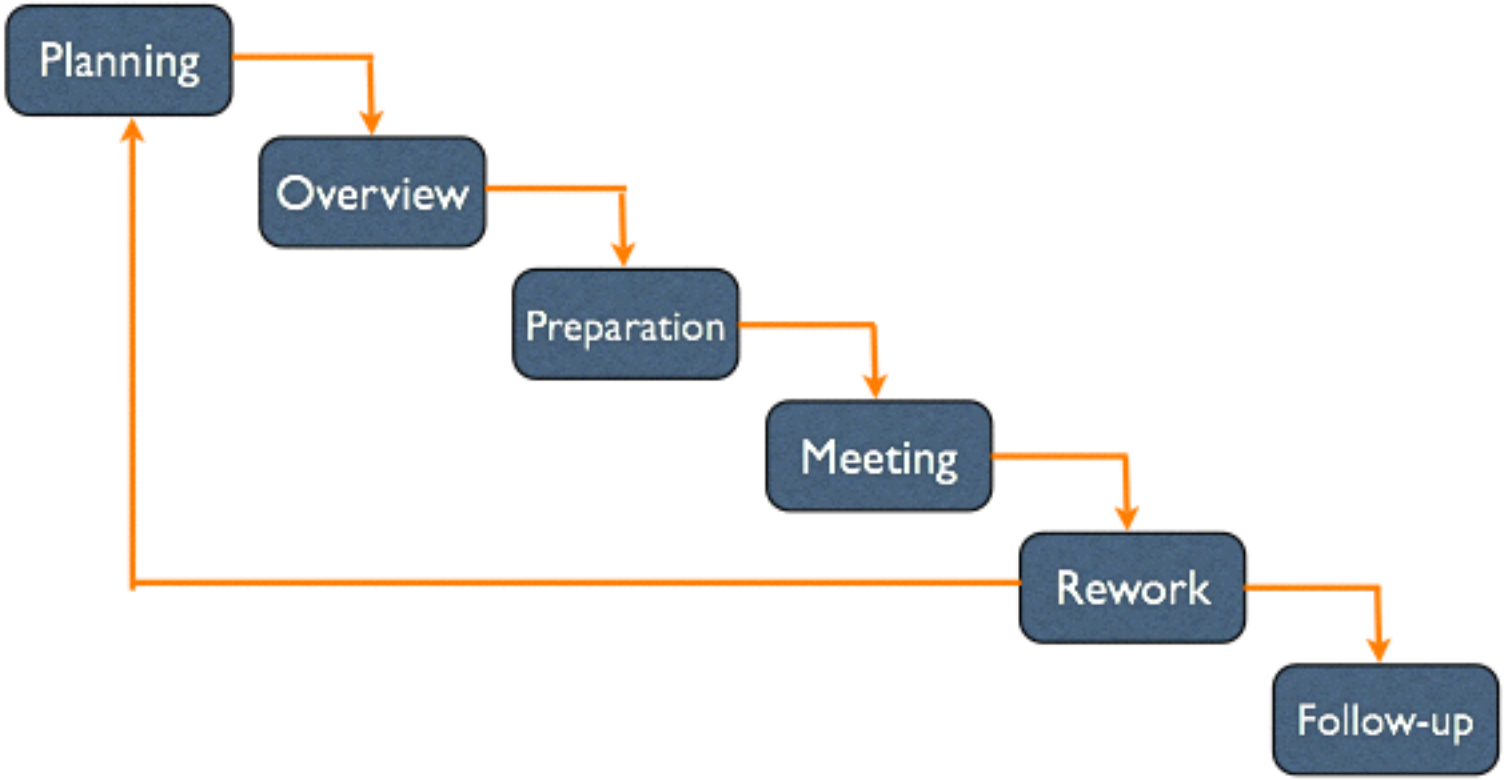
- Code Reviews
- Refactoring

# Code Reviews

- Formal code review meetings
  - Well defined, specific participant roles and responsibilities, documented review procedure, reporting of process…
- Lighter weight methods of code reviews
  - Tool-assisted code review
  - Ad-hoc review (over-the-shoulder)
  - Peer deskcheck / Email pass-around
  - Pair programming

See more at
http://www.atlassian.com/software/crucible/learn/codereviewwhitepaper.pdf

# Formal Review Meetings



# Formal Reviews – Reviewee (Author)

- Be quiet while you listen to the entire criticism/ question
- Deliver defense in term of the problem you were trying to solve
- Your code is on trial, not you!

# Formal Reviews – Reviewer

- Criticize the code, not the developer
- Before declaring a piece of code wrong, ask why it was done the way it was
- Remember: this is your colleague and s/he will be reviewing you in the future

# Formal Reviews – Moderator

- Keep review flowing
- Keep people on topic
- Break infinite loops

# Formal Reviews – Recorder

- Take notes describing the defects that were detected

# Formal Reviews – Praise!

- Make sure to notice something unique or elegant
- Acknowledge when a developer is trail blazing

# Formal Reviews – Problems

- Real problems are interpersonal
- Watch for:
  - Personal instead of code criticism
  - Axe grinding
  - Stylistic criticism

# Lighter weight methods of code reviews

- Tool-assisted code review: Authors and reviewers use specialized tools designed for peer code review.
- Ad-hoc review (over-the-shoulder): One developer looks over the author's shoulder as the latter walks through the code.
- Peer deskcheck: (Only) one person besides the developer reviews the code.
- Email pass-around: Multiple developers may be involved in a concurrent, online deskcheck or source code management system emails code to reviewers automatically after a check-in
- Pair Programming: Two authors develop code together at the same workstation such as is common in Extreme Programming.

# Tool-assisted code review

There are many examples of tools you can use for code reviews

e.g.

- ReviewBoard (http://www.reviewboard.org)
- Code Collaborator (http://smartbear.com/products/software-development/code-review)

# Pair Programming (1)

- ***Increased discipline.*** Pairing partners are more likely to "do the right thing" and are less likely to take long breaks.
- ***Better code.*** Pairing partners are less likely to produce a bad design due to their immersion, and tend to come up with higher quality designs.
- ***Multiple developers contributing to design.*** If pairs are rotated frequently, several people will be involved in developing a particular feature. This can help create better solutions, particularly when a pair gets stuck on a tricky problem.
- ***Improved morale.*** Pair programming can be more enjoyable for some engineers than programming alone.

# Pair Programming (2)

- ***Collective code ownership.*** When everyone on a project is pair programming, and pairs rotate frequently, everybody gains a working knowledge of the entire codebase.
- ***Mentoring.*** Everyone, even junior programmers, possess knowledge that others don't. Pair programming is a painless way of spreading that knowledge.
- ***Team cohesion.*** People get to know each other more quickly when pair programming. Pair programming may encourage team gelling.

# Refactoring

- Quality of code decays over time
  - Need to spend time cleaning up
- Common problems:
  - Duplicated code
  - Hard-to-read code
  - Long methods

- No refactoring = lazy programming

# Software metrics

Variety of measures proposed for assessing software quality and complexity:

- Function points, cyclomatic complexity, fan-in/fan-out
- All of them are highly correlated with LOC.

Metrics are highly context-sensitive.

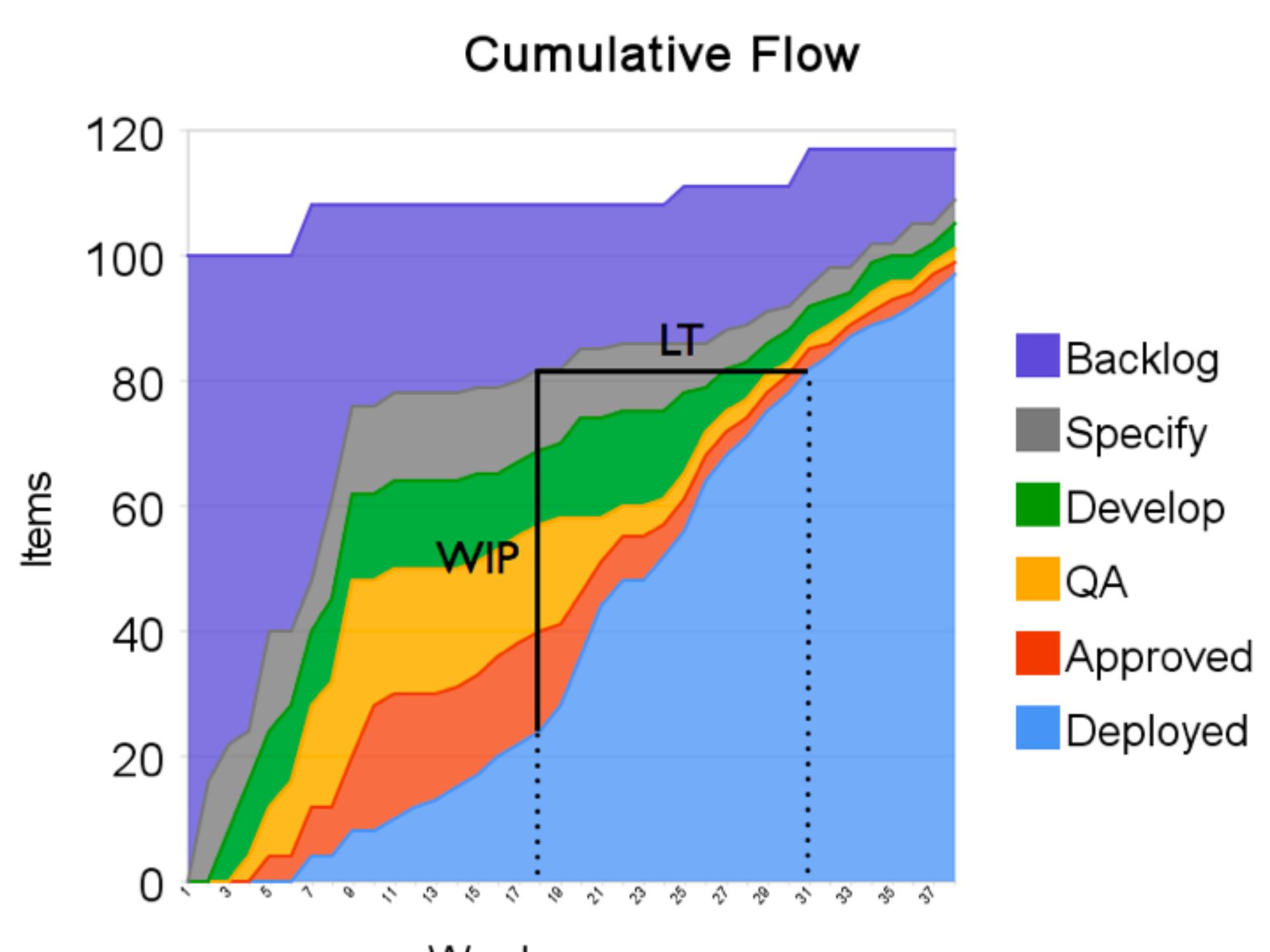
Most substantial effort: COCOMO and COQUALMO models from USC/Barry Boehm

More objective metrics come from dynamic analysis (profiling)


http://en.wikipedia.org/wiki/Project_management_triangle

# Process metrics

- Velocity and burndown charts:
  - How many stories are left?
  - How many story points are we finishing per day (throughput)
- Lead time:
  - What is time between task creation and task close?
- Work in progress:
  - How many items are we still working on?
- Capture these last two measures using a Cumulative Flow Diagram to measure throughput.
  - One characteristic of a Kanban approach to organizational change.


Cumulative Flow
120
100
80
60
40
20
0
Items
LT
WIP
Backlog
Specify
Develop
QA
Approved
Deployed

# Summary

- Software Quality is a large problem
  - Code quality is an important part of it
- Code quality is difficult to assess directly
  - Usually associated to process quality
- Good mechanisms for these processes
  - Cultural, Human, Automatic
  - Pair programming, software metrics
- Good design → Good code
- In a future lecture: Refactoring and code smells