Course Introduction

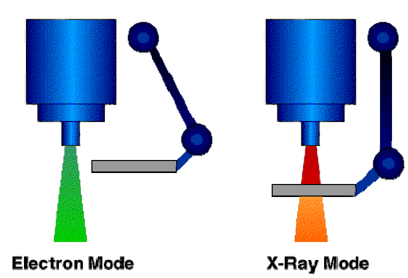
17-654/17-765

Analysis of Software Artifacts

Jonathan Aldrich

Software Disasters: Therac-25

- Delivered radiation treatment
- 2 modes
  - Electron: low power electrons
  - X-Ray: high power electrons converted to x-rays with shield
- Race condition
  - Operator specifies x-ray, then quickly corrects to electron mode
  - Dosage process doesn't see the update, delivers x-ray dose
  - Mode process sees update, removes shield
- Consequences
  - 3 deaths, 3 serious injuries from radiation overdose



from http://www.netcomp.monash.edu.au/cpe9001/assets/readings/HumanErrorTalk6.gif

source: Leveson and Turner, An Investigation of the Therac-25 Accidents, *IEEE Computer*, Vol. 26, No. 7, July 1993.

Software Disasters: Ariane 5

- $7 billion, 10 year rocket development
- Exploded on first launch
  - A numeric overflow occurred in an alignment system
  - Converting lateral velocity from a 64 to a 16-bit format
  - Guidance system shut down and reported diagnostic data
  - Diagnostic data was interpreted as real, led to explosion
- Irony: alignment system was *unnecessary* after launch and should have been shut off
- Double irony: overflow was in code reused from Ariane 4
  - Overflow impossible in Ariane 4
  - Decision to reuse Ariane 4 software, as developing new software was deemed too risky!

from http://www-user.tu-chemnitz.de/~uro/teaching/crashed-numeric/ariane5/

source: Ariane 501 Inquiry Board report

Software Disasters

Software Quality Challenges

[adapted from Scherlis]

- **Expense**
  - Testing and evaluation may consume more time and cost in the software engineering process than design and code development
- **Precision**
  - Almost impossible to completely succeed in testing and QA
    - “Very high quality” is rarely achieved, even for critical systems
  - Major gaps in testing and inspection
- **Consequences**
  - NIST report: $60B lost
  - Developers: Holding back features and new capabilities

Hardware Disasters: What’s Different?

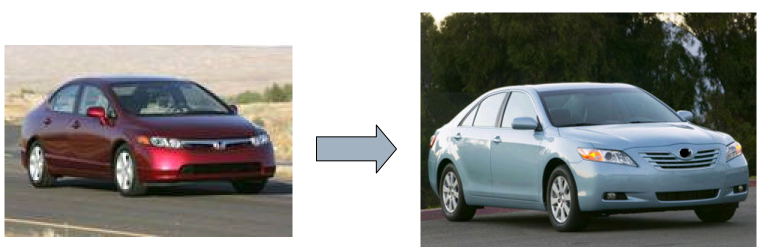
- Can be equally serious
- But don’t seem to be equally common

Why is Building Quality Software Hard?

- For other disciplines we do pretty well
  - Well-understood quality assurance techniques
  - Failures happen, but they are arguably rare
  - Engineers can measure and predict quality
- For software, we aren't doing well
  - How many cars get recalled for a patch once a month?
  - Failure is a daily or weekly occurrence
  - We have relatively poor techniques for measuring, predicting, and assuring quality

Software vs. other Engineering Disciplines

- Every software project is different
  - Classifications of engineering design
    - Routine design: specialize a well-known design to a specific context
      - Most common in engineering projects



Anyone recognize these cars?

Software vs. other Engineering Disciplines

- Every software project is different
  - Classifications of engineering design
    - Routine design: specialize a well-known design to a specific context
      - Most common in engineering projects
    - Innovative design: extend a well-known design to new parameter values
      - Sometimes risky – see Tacoma Narrows Bridge!

Software vs. other Engineering Disciplines

- Every software project is different
  - Classifications of engineering design
    - Routine design: specialize a well-known design to a specific context
      - Most common in engineering projects
    - Innovative design: extend a well-known design to new parameter values
    - Creative design: introduce new parameter values into the design space
      - Involves generating new prototypes
        - Variants of old prototypes, or completely new
      - Relatively unusual, and highly risky

Software vs. other Engineering Disciplines

- Every software project is different
  - Classifications of engineering design
    - Routine design: specialize a well-known design to a specific context
      - Most common in engineering projects
    - Innovative design: extend a well-known design to new parameter values
    - Creative design: introduce new parameter values into the design space
      - Involves generating new prototypes
        - Variants of old prototypes, or completely new
      - Relatively unusual, and highly risky
  - Software
    - Nearly all design is innovative or creative
    - As soon as design is routine, we put it in a library, language or tool!
      - "software manufacturing" will never happen

Software's Unmatched Complexity

- 50 Mloc = 1 million pages
  - What other man-made artifacts have designs this large?
  - We do because software is so flexible and powerful
  - We are limited only by complexity
    - As soon as we manage one level of complexity, the market will push us to add more!
- Worse: *every page matters*
  - Q: Could Windows crash because a third-party device driver has a bug?
  - A: Yes. In fact, that's the biggest cause of Windows crashes.
- *Why?*

Engineering Mathematics

- Continuous mathematics: calculus, etc.
  - Foundation of electrical, mechanical, civil, even chemical engineering
- Some quality strategies
  - Divide and conquer
    - Break a big problem into parts
      - Physical location: floor, room…
      - Conceptual system: frame, shell, wiring, plumbing…
    - Solve those parts separately
  - Overengineer
    - Build two so if one fails the other will work
    - Build twice as strong to allow for failure
  - Statistical analysis of quality
    - Relies on continuous domain
  - These work because the different parts of the system are independent
    - Never completely true, but true enough in practice

Software uses *Discrete* Mathematics

- Old quality strategies fail!
  - Divide and conquer
    - Butterfly effect: small bugs mushroom into big problems
  - Overengineering
    - Build two, and both will fail simultaneously
  - Statistical quality analysis
    - Most software has few meaningful statistical properties
- *Discrete math defeats conventional modularity*
  - Must *leverage discrete math* to analyze software
    - Choose concrete cases based on conceptual categories
      - Functional test coverage
      - Inspection checklists
      - Dynamic analysis
    - Construct proofs based on considering all abstract cases
      - Static analysis
      - Formal modeling
      - Program verification
    - Very *different* from analysis in other engineering disciplines

Questions for Analysis

- How can we ensure a system *does not behave badly?*
- How can we ensure a system *meets its specification?*
- How can we ensure a system *meets the needs of its users?*

Software Analysis, Defined

- *The systematic examination of a software artifact to determine its properties*
  - Systematic
    - Attempting to be comprehensive
      - Test coverage, inspection checklists, exhaustive model checking
  - Examination
    - Automated
      - Regression testing, static analysis, dynamic analysis
    - Manual
      - Manual testing, inspection, modeling
  - Artifact
    - Code, execution trace, test case, design or requirements document
  - Properties
    - Functional: code correctness
    - Quality attributes: evolvability, security, reliability, performance, …

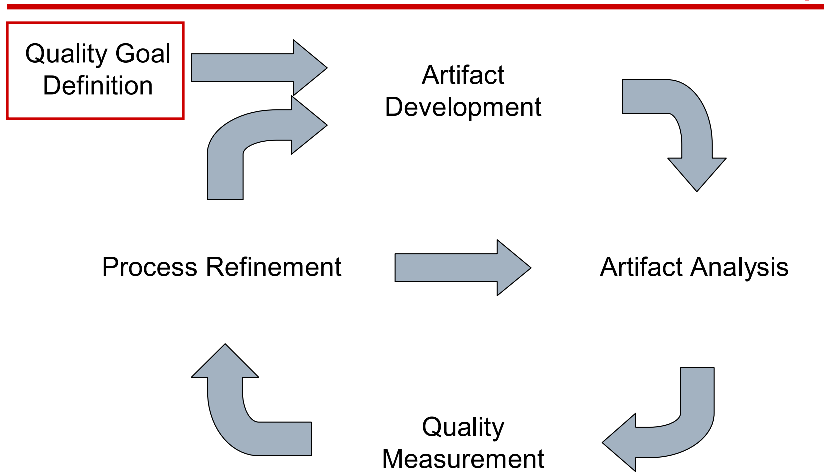
Verification and Validation

[adapted from Scherlis]

Two kinds of Analysis questions

- Verification
  - Does the system meet its specification?
    - i.e. *did we build the system right?*
  - Flaws in design or code
    - Incorrect design or implementation decisions
- Validation
  - Does the system meet the needs of users?
    - i.e. *did we build the right system?*
  - Flaws in specification
    - Incorrect requirements capture
- We will focus mostly on verification
  - Testing, inspection discussion will touch on validation
  - Other validation approaches beyond scope of course
    - prototyping, interviews, scenarios, user studies
    - A principal focus of the Methods course

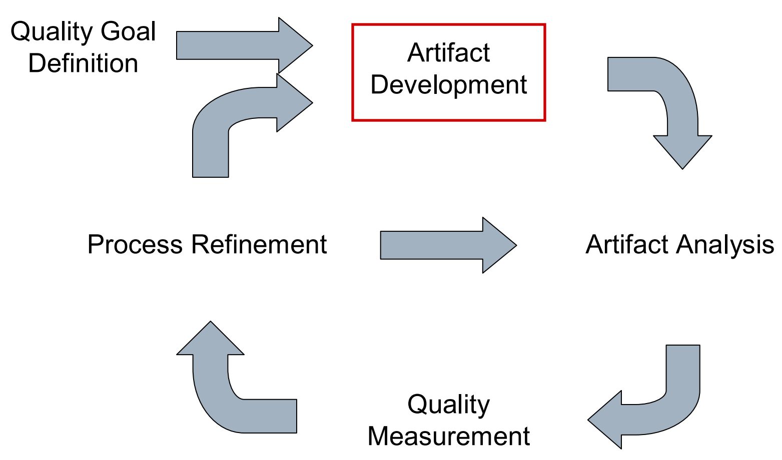
Analysis in a Process Context



Discussion: Quality Goals

- How might one define quality goals?
- # defects / kloc
- response time < 3 sec
- max down time
- mean time to failure
- free from buffer overflow
- # concurrent users
- safety (100%), security (0 security breaches)
- Generally depends on domain

Analysis in a Process Context



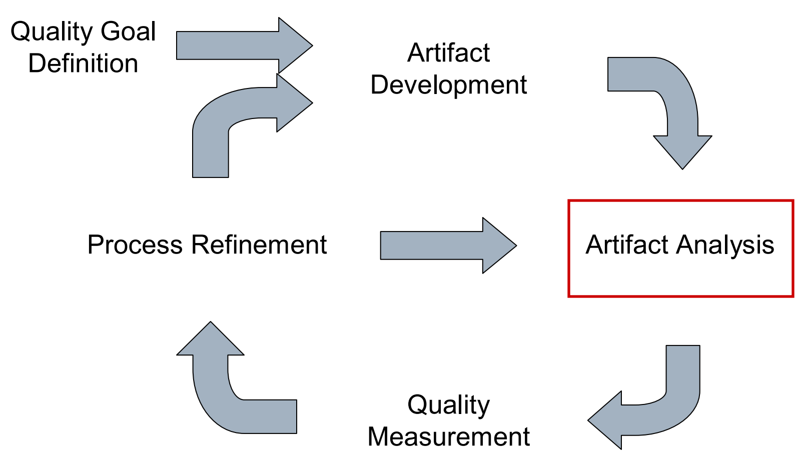
Prevention – Worth a Pound of Cure

- Requirements
  - Quality stakeholders
  - Non-functional attributes
- Process
  - Measurement and feedback
  - Testers and their role
    - E.g., S&S, agile
    - CMM, TSP, etc.
  - Risk mgmt
- Architecture
  - Robustness and self-healing
- Design
  - Robustness patterns
  - Safe APIs
  - Analysis
- Coding
  - Safe languages
  - Safe coding practices
  - Encapsulation / sandboxing
- Specific practices
  - Use of tools
  - Defect tracking
  - Root cause analysis

Evaluative techniques like testing are important—
but quality cannot be tested in!

[adapted from Scherlis]

Analysis in a Process Context



Discussion: Kinds of Analysis

Analysis Type	**What it's good for**
Testing	*Does it do what I want?*
Inspect to coding stand	*Identify bugs – esp. logical*
pair programming	*written to standards*
	avoid stupid mistakes
static analysis – e.g. beam	*locking / resource management*
	design/ requirement conform.
traceability matrix	*design errors*
model checking	*validation*
customer satisfaction	*UI requirement validation*
walkthrough	
inspection of other artifacts	*errors on requirements*

Principal Evaluative Techniques

[adapted from Scherlis]

- Testing
  - Direct execution of code on test data in a controlled environment
    - Functional and performance attributes
      - Component-level
      - System-level
  - Identify and locate faults – no assurance of complete coverage
- Inspection
  - Human evaluation of code, design documents (specs and models)
    - Structural attributes
      - Design and architecture
      - Coding practices
      - Algorithms and design elements
  - Creation and codification of understanding
- Dynamic analysis
  - Tools extracting data from test runs
    - Finding faults: memory errors
    - Gathering data: performance, invariants
  - Information is precise but does not cover all possible executions

Emerging Evaluative Techniques

[adapted from Scherlis]

- Modeling
  - Building and analyzing formal models of a system
    - Find design flaws
    - Predict system properties
  - Often tool-supported
- Static analysis
  - Tool-supported direct static evaluation of formal software artifacts
    - Mechanical errors
      - Null references
      - Unexpected exceptions
      - Memory usage
  - Can yield partial positive assurance
- Formal verification
  - Formal proof that a program meets its specification
    - Typical focus on functional attributes
  - Often tool-supported
  - Typically expensive

Quality Assurance is More Than Testing

Some quality attributes are difficult to test:

- Attributes that **cannot easily be measured externally**
  - Is a design evolvable? — Design Structure Matrices
  - Is a design secure? — Secure Development Lifecycle
  - Is a design technically sound? — Alloy; Model Checking
  - Does the code conform to a design? — Reflexion models
  - Where are the performance bottlenecks? — Performance analysis
  - Does the design meet the user's needs? — Usability analysis
- Attributes for which **tests are nondeterministic**
  - Real time constraints — Rate monotonic scheduling
  - Race conditions — Analysis of locking
- Attributes relating to the **absence of a property**
  - Absence of security exploits — Microsoft's PREfast
  - Absence of memory leaks — Cyclone, Purify
  - Absence of functional errors — Hoare Logic
  - Absence of non-termination — Termination analysis

Defect Types and Analysis

(List is incomplete)

- Functional errors: incorrect output
  - Testing, Inspection, Formal Verification
- Integration errors: misuse of APIs
  - Inspection – identify conflicts at design time
  - Integration testing – find errors at earliest opportunity
  - Types, analysis, and model checking – verify interface compatibility
- Mechanical defects: memory and concurrency errors
  - Static analysis – assure absence of mechanical defects
  - Dynamic analysis – identify at run time
- Robustness, security, evolvability errors
  - Security, robustness testing
  - Inspection of code and design
  - Static analysis – find security flaws, assure conformance to design
- Performance errors
  - Load testing, profiling – measure performance on realistic load
- Usability errors
  - Prototyping, user studies

Discussion: Criteria for Evaluating Techniques

- Cost – developer time taken
  - learning curve
  - money to buy a tool
- Benefit
- accuracy – does it pin down the line of code
  - ratio of true defects to false positives
- fitness for purpose, for artifact
- applicability to lifecycle
- % defects that reach the client
- coverage - functionality

[adapted from Scherlis]

Criteria for Evaluating Techniques

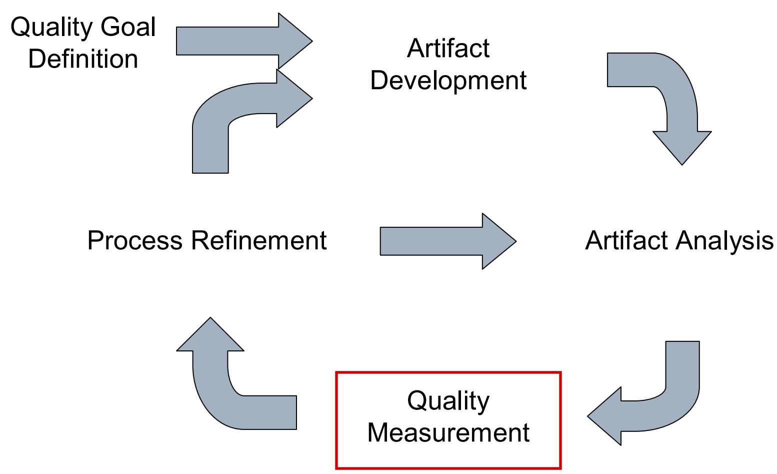
- Cost
  - Money, time to market
  - Sunk and recurring
- Timeliness
  - Design time
  - During coding
  - During testing
  - After deployment
- Accuracy
  - False positives
  - False negatives
- Development value
  - Is the information actionable?
    - e.g. enough information to fix a bug?
  - Risks of adoption
- Metrics: observability of outcomes
- Scope: What kinds of defects?
  - System scale and complexity
  - Error vs. fault focus
  - Non-functional attributes: performance, usability, security, safety, etc.
  - Functionality
- Integration and value during development
  - Defect prevention support
  - Architecture design
  - Code management
  - Modeling and design intent capture

[adapted from Scherlis]

Your Questions

- Q: In a slide, you said testing can locate a fault. Is that true?
- A: Sometimes!
  - Unit tests test a small unit of code (usually a procedure)
    - Errors found must be in that code
  - Tests that result in a crash show the error location
    - Which may or may not be the location of the fault (root cause)
  - System tests that just give the wrong answer aren't very informative
    - The fault in the code could be anywhere in the execution
    - But it's a concrete execution trace, so a debugger may help you isolate the problem
- What about basic course information?
- A: Sorry, didn't get to it, but I will today!

Analysis in a Process Context



Faults, Errors, Failures, Hazards

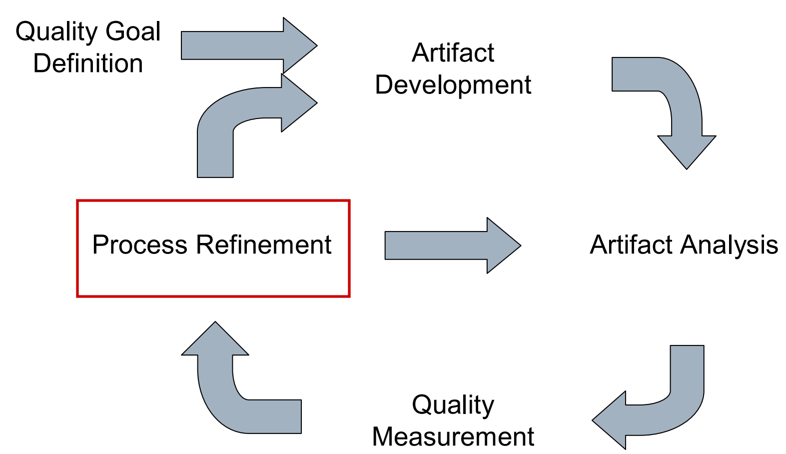
[adapted from Scherlis]

- **Fault**
  - **Type 1 – a flaw in an attached physical component**
    - Traditional notion of a fault in hardware reliability theory (physical parts wearing out)
  - **Type 2 – a static flaw in software code**
    - Syntactically local in code or structurally pervasive
    - Software faults cause errors only when triggered by use.
- **Error – incorrect state at execution time caused by a fault**
  - E.g., buffer overflow, race condition, deadlock, corrupted data
- **Failure – effect of an error on system capability**
  - E.g., program crashes, attacker gains control, program becomes unresponsive, incorrect output
- **Severity – cost of failure to stakeholders**
  - E.g., Loss of life, privacy compromise
- **Hazard – product of failure probability and severity**
  - Equivalent to risk exposure

Robustness / Fault Tolerance

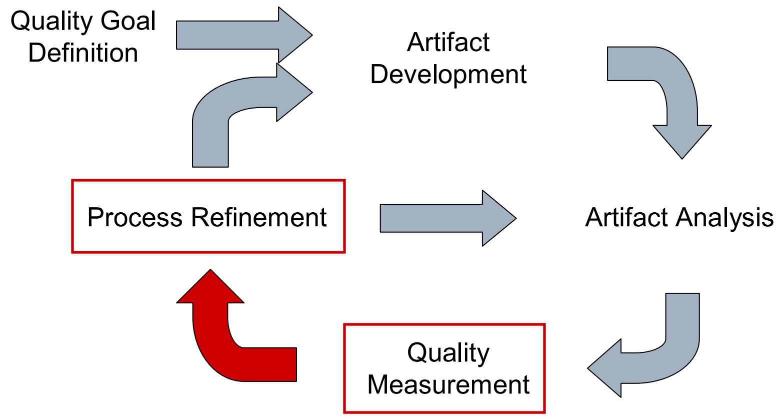
- How does the system behave in the presence of errors in the system or environment?
  - Hardware: memory parity errors, sensor failures, actuator anomalies
  - Software: buffer overflows, null dereferences, protocol violations
  - Environment: network faults, inputs out of range
- Robustness: diminishing the likelihood or severity of **failure** in response to the **fault**
  - Buffer overrun in C == ? in Java
- Strategies for robustness
  - Type systems
  - Run-time system checks
  - Rebooting components
  - Autonomic architectures
  - Self-healing data structures
  - Data validation
  - State estimators

Analysis in a Process Context



Too Many Defects: What to do?

- Real question: *Why?*



Measuring Quality

- Defects / kloc not enough
  - Break down by category, severity
  - Break down by phase introduced
  - Break down by phase detected
- Other metrics useful as well
  - case study: Microsoft [source: Manuvir Das]
    - code velocity / developer productivity
    - tool effectiveness (e.g. fix rate on warnings)
- Crucial information for quality analysis!

Discussion: Quality Problem Scenarios

- Many defects not found until system test
- Many defects introduced in design are found when coding
- Several similar security vulnerabilities are identified

Root Cause Analysis at Microsoft

- Gather data on failures
  - Every MSRC bulletin
  - Beta release feedback
  - Watson crash reports
  - Self host
  - Bug databases
- Understand important failures in a deep way
  - Understand why the defect was introduced
    - Not just the incorrect code
  - Understand why it was not caught earlier
    - Process failure
  - Identify patterns in defect data
- Design and adjust the engineering process to ensure that these failures are prevented
  - Developer education
  - Review checklists
  - New static analyses

source: Manuvir Das

Case study: QA at Microsoft

- Original process: manual code inspection
  - Effective when system and team are small
  - Too many paths to consider as system grew
- Early 1990s: add massive system and unit testing
  - Tests took weeks to run
    - Diversity of platforms and configurations
    - Sheer volume of tests
  - Inefficient detection of common patterns, security holes
    - Non-local, intermittent, uncommon path bugs
  - Was treading water in Longhorn/Vista release of Windows
- Early 2000s: add static analysis
  - Wide variety of tools
    - Test coverage, dependency violation, insufficient/bad design intent, integer overflow, allocation arithmetic, buffer overruns, memory errors, security issues
  - Enforced *automatically* at code check-in

Course Goals

- Understanding
  - Where different analyses are appropriate
  - Tradeoffs between analysis techniques
  - Theory sufficient to evaluate new analyses
  - Measurement and management of analysis
- Experience
  - Writing simple analyses
  - Applying analysis to software artifacts

Course Outline

- Introduction (today)
- Traditional analysis techniques
  - Testing: techniques, processes, tools
  - Inspection
- Program semantics and verification
  - Semantics and representations of code
  - Formal specification
  - Proving programs correct
- Static analysis
  - Program representations and bug finders
  - Dataflow analysis
  - Static analysis tools
- Analysis across the software lifecycle
  - Analyzing designs
  - Principles of security analysis; STRIDE
  - Performance analysis: profiling
  - Analyzing real-time and concurrent systems
  - Dynamic analysis, languages, and type systems
- Putting it all together
  - Quality in the organization
  - Case studies: Microsoft and eBay

Homeworks and Projects

- Find seeded defects using testing and inspection techniques
- Prove small programs correct with Hoare logic
- Check program correctness with the ESC/Java tool
- Design a dataflow analysis, and implement in in an analysis framework
- Analyze a design for evolvability, consistency
- Probe a software system for security violations
- Measure and tune system performance
- Assure synchronization in a concurrent system
- Run a commercial or research analysis tool on source code and report on the experience
- Develop a quality assurance plan for your studio

Evaluation

- 1-week assignments (~45%)
  - Basic understanding of analysis techniques
  - Write and apply custom analyses
  - Engineering tradeoffs
  - Most alone, some done in pairs
- 2-week group projects (~20%)
  - Evaluate analysis tools on studio or other project
    - Written reports and in-class presentations
  - Develop a quality assurance plan
    - For your Studio, Practicum, or other project
- Midterm and Final exam (~15% each)
  - Theory and engineering
- Class participation (~5%)
  - Discussion, presentations, participation sheets
- Schedule is on the web
  - Assignments due on Tuesdays

Ph.D. Projects

- Possible topics
  - Literature survey
    - Study techniques, put into framework, identify open problems
  - Comparative evaluation
    - Your experience with multiple techniques or tools
  - Development of a new analysis technique
  - Application of an analysis technique to a new problem domain
- Requirements
  - Written report
    - Length depends on nature of project
  - Research emphasis
  - Class presentation
- Details to be arranged with instructor

Policies

- Time Management
  - Keep track of time spent on each assignment
- Late Work
  - 5 free late days; use whenever you like
  - No other late work except under extraordinary circumstances
- Collaboration Policy
  - You may discuss the lectures and assignments with others, and help each other with technical problems
  - Your work must be your own. You may not look at other solutions before doing your own. If you discuss an assignment with others, throw away your notes and work from the beginning yourself.
  - You must cite sources if you use or paraphrase any material
  - If you have any questions, ask the instructor or TAs

Course Instructor and TAs

- Instructor
  - Jonathan Aldrich
    aldrich+ at cs.cmu.edu

- TAs
  - Ciera Christopher
    cchristo [at] cs.cmu.edu

  - Megha Jain
    meghajai [at] andrew.cmu.edu

Session Summary

- Achieving software quality is difficult
  - Design innovation, software complexity and discreteness
- Analysis defined
  - *The systematic examination of a software artifact to determine its properties*
- Diversity of analysis techniques
  - Testing, inspection, static and dynamic analysis, model checking, formal verification
  - Must know *when* and *how* to *use* and *measure*