CSE584: Software Engineering

Lecture 8 (May 27, 1997)

David Notkin
Dept. of Computer Science & Engineering
University of Washington
www.cs.washington.edu/homes/notkin

UW CSE
CSE584: Software Engineering

Notkin (c) 1997 1

Lecture 8, Outline [approximate minutes]

- Approaches for quality assurance [15]
- Proofs of correctness [30]
- Reviews, inspections, etc. [30]
- Break [10]
- Statistically-based approaches [45]
- Wrap-up [10]

- Next week
  - PSP, CMM, ISO 9000
  - Testing
  - What's hot in software engineering research

Notkin (c) 1997 2

Approaches to quality assurance

- Testing
- Proofs of correctness
- Process improvement
  - CMM, ISO 9000, reviews, inspections, …
- Statistical measures
  - Reliability, Cleanroom, etc.
- Software safety (fault tree analysis, etc.)
- Others?

Notkin (c) 1997 3

Proofs of program correctness

- Given a precise specification and an implementation, show that the implementation satisfies the specification
  - Distinct from proving properties about specifications
- Caveats
  - Not generally practical
  - Can provide some useful insights for programming

Notkin (c) 1997 4

Pre- and post-conditions

- Pre-condition
  - Predicate describing state before a program executes
- Post-condition
  - Predicate describing state after a program executes

Notkin (c) 1997 5

Hoare triples

- `{P} S {Q}`
  - A predicate that is true if and only if
    - » when pre-condition `P` is true
    - » and then `S` is executed
    - » then post-condition `Q` will be true
- Strong correctness requires `S` to terminate for `{P} S {Q}` to be true
- Weak correctness does not

Notkin (c) 1997 6

Example

- `{ x = X ^ y = Y }`
  `t := x; x := y; y := t`
  `{ x = Y ^ y = X }`
- Compute intermediate assertions
  - ```
    { x = X ^ y = Y }
    t := x;
    { x = X ^ y = Y ^ t = X }
    x := y
    { x = Y ^ y = Y ^ t = X }
    y := t
    { x = Y ^ y = X ^ t = X }
    ```
- `{ x = Y ^ y = X ^ t = X } => { x = Y ^ y = X }`

UW CSE
CSE584: Software Engineering

Notkin (c) 1997 7

Semantics of statements

- In the example, I appealed to your intuition about what := does
- But a more precise definition is needed
- This is often done using weakest preconditions (wp's) [Dijkstra]
  - wp(S,Q) is the weakest condition that must be true beforehand so that S terminates in a state such that Q is true
- When S is a language construct, wp defines its semantics

UW CSE
CSE584: Software Engineering

Notkin (c) 1997 8

Assignment wp

- `wp( x := E,Q(x)) = Q(E)`
  - For a condition `Q(X)` to be true after execution of `x := E`, the condition `Q(E)` must hold beforehand
- `wp( j := j+1, j <= 1) = (j <= 0)`
- What about side effects?

UW CSE
CSE584: Software Engineering

Notkin (c) 1997 9

Hoare triples and wp's

- A nice relationship holds between Hoare triples and wp's
- To prove {P}S{Q}
  - First compute wp(S,Q)
  - And then prove P => wp(S,Q)
- In essence, this computes backwards from the desired post-condition, through each statement, to the original pre-condition

UW CSE
CSE584: Software Engineering

Notkin (c) 1997 10

Sequencing wp

- `wp((S1;S2),Q)= wp(S1,wp(S2,Q))`
- ```
  wp( j:=j+2;k:=k-2, j+k=1) =
    wp(j:=j+2,wp(k:=k-2,j+k=1)) =
    wp(j:=j+2,j+k-2=1) =
    (j+2+k-2=1) =
    j+k=1
  ```

UW CSE
CSE584: Software Engineering

Notkin (c) 1997 11

Example

- `wp(t:=x;x:=y;y:=t,x=X^y=Y)`
- `wp(t:=x;x:=y,wp(y:=t,x=X^y=Y))`
- `wp(t:=x;x:=y,x=X^t=Y)`
- `wp(t:=x,wp(x:=y,x=X^t=Y))`
- `wp(t:=x,y=X^t=Y)`
- `y=X^x=Y`

UW CSE
CSE584: Software Engineering

Notkin (c) 1997 12

Conditional wp's

- `wp(if C then S1 else S2,Q)`
  - Have to cover both parts of the conditional
- `(C => wp(S1,Q))^(~C => wp(S2,Q)))`

Notkin (c) 1997 13

Example

- ```
  {x <> 0}
  if x < 0 then x := -x else x := x-1
  {x >= 0}
  ```

Notkin (c) 1997 14

Loops

- Weakest pre-conditions on loops are problematic since they need not terminate
- So instead we approximate the wp of a loop with a loop invariant
- A loop invariant differs from a weakest pre-condition
  - Does not imply termination
  - May be stronger than is strictly necessary

Notkin (c) 1997 15

Loop invariants

- What role must the loop invariant `I` of `while B do S` play to ensure post-condition `Q` holds afterwards?
- We need
  - `{B^I}S{I}`
  - `~B ^ I => Q`
- That is, the loop maintains the invariant and on termination, the post-condition holds

Notkin (c) 1997 16

Example

- ```
  {T}
  j:=1; s:=b[0];
  while j < 11 do
      j:=j+1; s:=b[j]
  od
  ```
  $\{s=\Sigma_{k=0..10}\ b[k]\}$
- What's an appropriate loop invariant?

Notkin (c) 1997 17

Termination

- Loop invariants don't address termination
- If termination is material, a separate proof is used
- These proofs generally use well-founded sets
  - Essentially, one finds a value that is monotonically increasing (decreasing) towards a fixed bound
  - In the last example, j monotonically approaches 11

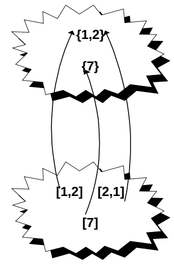
Notkin (c) 1997 18

Miscellaneous

- Recursion
- Side effects
- Procedures and functions
  - Parameter passing mechanisms
- Abstract data types

Correctness of ADTs [Hoare]

- Need to define pre- and post-conditions on both the abstract state and also the concrete state
- Relate these through a (many to one) representation function [Liskov & Guttag]



Reviews, etc.

- Reviews, walkthroughs, and inspections are all in a family of activities where an artifact (specification, code, etc.) is studied by a peer group to improve the artifact's quality
- There is a large and increasing literature that demonstrates the effectiveness (although not always the cost-effectiveness) of these approaches

Reviews, etc.

- N-heads are better than one
- Intended to
  - identify defects
  - identify needed improvements
  - encourage uniformity and conformance to standards
  - enforce subjective rules

Purposes

- Increase quality through peer review
- Provide management visibility
- Encourage preparation

- Explicit non-purpose
  - Assessment of individual abilities for promotion, pay increases, ranking, etc.
  - Management usually not permitted at reviews

Walkthrough

- A formal activity
- A programmer (designer) presents a program (design)
- Values of sample data are traced
- Peers evaluate technical aspects of the design

Inspections [Sommerville]

- Formal approach to code review
- Intended explicitly for defect *detection* (not correction)
- Defects include logical errors, anomalies in the code (such as uninitialized variables), non-complicance with standards, etc.

Inspection requirements

- A precise specification must be available
- Peers must be knowledgeable about organizational standards
- Code should be syntactically correct and basic tests passed
- Error checklist must be provided

Inspection process

- Plan
- Overview
- Individual preparation
  - Code, documentation distributed in advance
- Meeting
- Rework
- Follow-up

Inspection teams

- Four or more members
- Author of code
- Reader of code (reads to team)
- Inspector of code
- Moderator chairs meeting, takes notes, etc.

Inspection checklists

- Checklist of common errors drives inspection
- Checklist dependent on programming language
  - Weaker type systems usually imply longer checklists
- Examples
  - Initialization, loop termination, array bounds, ...

Inspection rate

- 500 statements/hour during overview
- 125 statements/hour during individual prep
- 90-125 statements/hour during review
- Inspecting 500 statements can take 40 person-hours
  - For 1MLOC, this would be about 40 person-years of effort

Issues in inspections

- Can groupware technology significantly improve inspections?
- Can you have inspections without meetings?
  - Since meetings are expensive to hold and schedule
  - Since the preparation may catch more defects than the meetings
- See Adam Porter's talk on Thursday

UW CSE CSE584: Software Engineering Notkin (c) 1997 31

Statistical approaches

- There are a number of approaches to quality assurance that are (in varying senses) based on statistics
  - Software reliability
  - N-version programming
  - Cleanroom

UW CSE CSE584: Software Engineering Notkin (c) 1997 32

Software reliability [RST]

- The probability that software will provide failure-free operation in a fixed environment for a fixed interval of time
  - A system might have reliability 0.96 when used for a one week period by an expert user
- Mean-time-to-failure is the average interval of time between failures
- One common use of software reliability models is to decide when it's OK to ship a product

UW CSE CSE584: Software Engineering Notkin (c) 1997 33

Operational profiles

- An accurate operational profile is needed
  - Frequency of application of specific operations for the program being studied
- An operational profile is the probability density function (over the entire input space) that best represents how the inputs would be selected during the life-time of the software

UW CSE CSE584: Software Engineering Notkin (c) 1997 34

Understood domains

- In industries such as telecommunications, operational profiles can be fairly easily gathered
- The phone company has records of virtually every call made in the last 20 years
  - Phones are used in pretty consistent ways

UW CSE CSE584: Software Engineering Notkin (c) 1997 35

Less understood domains

- But for shrink-wrapped software products, operational profiles are harder to divine
- How will different users using different products with different features behave?
  - CPA's vs. college students using a spreadsheet?
- Will usage change over time?
  - More or less than the phone system?

UW CSE CSE584: Software Engineering Notkin (c) 1997 36

Cost

- To assess reliabilities past the 3rd or 4th decimal place can require an enormous amount of testing
- Should all failures be considered equally bad?
  - Showstoppers vs. “wrong color”
- Oracles of “correctness” aren’t always easy
- Monitoring phone switches is relatively easy; monitoring shrinkwrap isn’t

UW CSE CSE584: Software Engineering Notkin (c) 1997 37

Applying reliability models

- There is extensive real use of models in this style
- There is also a lot of theoretical work that is never validated
  - Variants on models never compared to reality
- There are courses, books, etc. about how to apply reliability modeling in practice

UW CSE CSE584: Software Engineering Notkin (c) 1997 38

N-version programming

- The idea of N-version (multi-version) programming comes from a common hardware reliability approach--replication
- The basic notion is simple
  - Have N independent teams write N versions of a program
  - Run them all simultaneously and have them vote at specified points

UW CSE CSE584: Software Engineering Notkin (c) 1997 39

Objective

- Since the programs are built independently, the objective is to improve the quality by a multiplicative factor
  - A bug only hurts if it also appears in another (N/2)+1 versions
- This idea indeed works pretty well in hardware
- The cost issue in software is different, though
  - Not a matter of producing multiple chips, but of producing multiple implementations

UW CSE CSE584: Software Engineering Notkin (c) 1997 40

Assumption

- But there is an underlying assumption at work
  - The implementations will fail independently
    - » Like the chips in hardware that fail based on physical structures
  - Otherwise, a multiplicative factor will not be gained
- Do independently built implementations of the same specification fail independently?

UW CSE CSE584: Software Engineering Notkin (c) 1997 41

Probably not

- Knight and Leveson did some experiments that showed that this assumption is probably false
  - In particular, they showed that similar errors often arise in independently implemented versions of the same specifications
- An additive benefit may arise from N-version programming, but not a multiplicative one

UW CSE CSE584: Software Engineering Notkin (c) 1997 42

Why?

- Errors are often in the specification
- Errors are often made at boundary conditions
- The complexity of a program is often in a small piece or two, which each group has trouble with
- The background and training of people in an organization are often similar

And now...

- N-version advocates are still out there in an aggressive way
  - I believe some organizations still require contracted software to be built this way
- There are some experiments showing independence
- There are attempts to introduce variety explicitly
  - Different specs, different languages, etc.
- I'm still completely opposed to this approach based on the Knight/Leveson experiments

Cleanroom [Harlan Mills]

- Cleanroom combines managerial and technical activities into a process intended to lead to very high quality software
  - Combines formal methods with statistical testing for reliability with incremental development
  - Does *not* allow unit execution or testing
- Effectiveness is a controversial issue

Basics: five points

- Formal specification
  - "Required system behavior and architecture"
  - Black box stimulus-response specification
- Incremental development
  - Partitioned into "user-function increments" that "accumulate into the final product"
- Structured programming
  - Limited use of control and data abstraction constructs; stepwise refinement of specification

Basics: five points (con't)

- Static verification
  - Components statically verified using mathematic correctness arguments
  - Individual components *neither* executed *nor* tested
    - » No white box testing, no black box testing, no coverage analysis
- Statistical testing
  - Each increment is tested statistically based on operational profile

Three teams

- Specification team
- Development team
  - Codes
  - Statically verifies using inspections
- Certification team
  - Develops and applies statistical tests
  - Reliability models used to decide when to ship

Claims

- Very aggressive positive claims
  - About 20-30 systems (all under 500KLOC)
- 100KLOC systems with (well) under 10 errors in the field in the first year or two
- Finds 1-4 errors/KLOC during statistical testing
- Some projects claim 70% improvement in development productivity

UW CSE
CSE584: Software Engineering

Notkin (c) 1997 49

Counterclaims [Beizer]

- Several (related) questions raised
  - Are comparisons to other methods fair?
  - Why eliminate unit testing?
  - Why trust software reliability modeling so much?
    - » Especially hard to get good operational models
- Claim is that unless Cleanroom embraces modern testing approaches, it will fail to be used broadly

UW CSE
CSE584: Software Engineering

Notkin (c) 1997 50