Fostering the Next Generation of Software Test Architects: Reappraising Software Testing 2

Cem Kaner, J.D., Ph.D.

Executive Vice-President, Association for Software Testing

Professor of Software Engineering, Florida Institute of Technology

Data Analysis / Presentation

by Nawwar Kabbani

February 5, 2010

These notes are partially based on research supported by NSF Grant CCLI-0717613 “Adaptation & Implementation of an Activity-Based Online or Hybrid Course in Software Testing.” Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Copyright (c) Cem Kaner

Underlying Objective

Foster the next generation of test architects.

UNDERLYING CONTRAST: COMMODITY-LEVEL SOFTWARE TESTING

- You are a commodity if:
  - your client perceives you as equivalent to the other members of your class
- Commodity testers:
  - have standardized skills / knowledge
  - are easily replaced
  - are cheaply outsourced
  - add relatively little to the project

Commodities

There are green bananas
and ripe bananas
and rotten bananas
and big bananas
and little bananas.
But by and large,
a banana is a banana.

Commodity testers have little on-the-job control over their pay, status, job security, opportunity for professional growth or the focus of their work.

What level are you working at? (Some Examples)

CHECKING	• Testing for UI implementation weakness (e.g. boundary tests) • Straightforward nonconformance testing • Verification should be thought of as the handmaiden to validation
BASIC EXPLORATION	• Quicktests • Straightforward tours to determine the basics of the product, the platform, the market, the risks, etc. • Here, we are on the road to validation (but might not be there yet)
SYSTEMATIC VARIATION	• Conscious, efficiently-run sampling strategy for testing compatibility with big pool of devices / interoperable products / data-sharing partners, etc. • Conscious, efficiently-run strategy for assessing data quality, improving coverage (by intentionally-defined criteria)
BUSINESS VALUE	• Assess the extent to which the product provides the value for which it was designed, e.g. via exploratory scenario testing
EXPERT INVESTIGATION	• Expose root causes of hard to replicate problems • Model-building for challenging circumstances (e.g. skilled performance testing) • Vulnerabilities that require deep technical knowledge (some security testing) • Extent to which the product solves vital but hard-to-solve business problems

GUI-Level Regression Testing: Commodity-Level Test Automation

- addresses a narrow subset of the universe of testing tasks
- re-use existing tests
  - a collection of tests that have one thing in common: the program has passed all of them
  - provide little new information about the product under test
  - tests are rarely revised to become harsher as the product gets more stable, so the suite is either too harsh for early testing or too simplistic / unrealistic for later testing
  - tests often address issues (e.g. boundary tests) that would be cheaper and better tested at unit level
- enormous maintenance costs
  - several basic frameworks for reducing GUI regression maintenance are well understood

The Tasks of Test Automation

- **Theory of error**
  What kinds of errors do we hope to expose?
- **Input data**
  How will we select and generate input data and conditions?
- **Sequential dependence**
  Should tests be independent? If not, what info should persist or drive sequence from test N to N+1?
- **Execution**
  How well are test suites run, especially in case of individual test failures?
- **Output data**
  Observe which outputs, and what dimensions of them?
- **Comparison data**
  IF detection is via comparison to oracle data, where do we get the data?

- **Detection**
  What heuristics/rules tell us there **might** be a problem?
- **Evaluation**
  How **decide** whether X is a problem or not?
- **Troubleshooting support**
  Failure triggers what further data collection?
- **Notification**
  How/when is failure reported?
- **Retention**
  In general, what data do we keep?
- **Maintenance**
  How are tests / suites updated / replaced?
- **Relevant contexts**
  Under what circumstances is this approach relevant/desirable?

We can automate any subset of these

Implementing the Agenda

- Next generation of test architects
  - Programming skills
  - Test skills
  - Understand simulators (including simulation-support tools)
  - Troubleshooting skills (loggers, performance analyzers, pattern recognition, investigative thinking)
  - Requirements analysis and a willingness to design tools to be (successfully and happily) used by nonprogrammers
  - Flexible understanding of the range of system-level test automation possibilities

History

- Planning for this course started in 2002
  - XP was in full swing
  - Web tutorials on nUnit were in development, and web discussion/coaching was easy to join
  - Books were in draft (Beck's was out and several others were coming)
  - I taught the course on my own, co-taught with Andy Tinkham and Pat McGee. Pat Bond also taught the course. We all had comparable results

Implementation-Level Testing

- We want students to understand the state of their code as they develop it
  - TDD
  - Eclipse
  - JUnit
  - Debugger
  - Coverage monitoring (ECL/Emma)
  - Checkstyle
  - Subversion
- Several black box tests become unnecessary if adequate implementation-level testing is done
- We use Java tools for now, but would be at least as happy with C# or Python if appropriate learning support materials were available.

Agile-Biased Course Content

- TDD (test-and-calibrate / code / test / fix / … / refactor / commit)
- Basic dev tools that support TDD
- Maintenance project (possible some years, not others)
- Unit test design (such as theories of coverage, testing Booleans, testing common control-flow structures, testing across different data structures)
- System-level test architecture (some examples)
  - Comparison-based (e.g. test against oracle)
  - Theory-of-error-based (e.g. test for manipulation of voting)
  - Sequence-based (e.g. long-sequence regression)

Student Profiles: Undergrads

- Florida Tech undergrads
  - Required course for SE B.Sc.
  - High GPA's / seniors / most are adequate programmers
- Many of the undergrads are used to tightly-defined problems (down to the variable names, class/method names and file structures) and find it difficult to work from scratch
- The two most successful instances were almost entirely undergrad. Includes 2009. Cannot generalize from these to predictions for next year.

Student Profiles: M.Sc.

- Local industry (aerospace) – rocket science mgrs are smart but no longer remember how to write code
- International. Highly variable educational backgrounds. Many are non-programmers despite near-perfect CS grades on their transcripts.
- Enormous resistance:
  - Refusal to even attempt TDD
  - Google code / cut-and-paste fails with TDD
  - Collaboration via delegation (you do this, I'll do that…)
  - Open disbelief that a testing course would involve programming
  - Disbelief that this approach is taken in industry
  - Belief that traditional development process are the One True Way

Challenges in Teaching Implementation-Level Test Design

- Unit test design (theories of coverage, testing Booleans, testing common control-flow structures, testing across different data structures)
- Challenges in teaching unit test design:
  - My expertise is in black box analysis (I am personally profoundly more interested in helping testers assess the quality of the product than the adequacy of the implementation (or the purity of the development process)).
  - **That means I rely on textbooks for inspiration (of me and my students). My focus will be on what the book credibly and intelligibly supports, IF I can find a book worth using.**

LEARNING OBJECTS (FROM MY NOTES TO WTST)

- Agile books focus on the tool and the process, not on the testing. Minimal design advice, and much of what is there is bad.
- The new university textbooks present testing almost as applied discrete math. Lots of theory. Lots of references to formal literature.
- I need a way to connect test ideas to tasks that programmers might actually do or be asked to do in real jobs (as distinct from MS research, IBM Research, etc.). These books are not helping me.
- I need materials that engineers (as opposed to theorists) can understand and not immediately lose patience with. I need books that help me with the credibility of what they teach. That is, specific technical advice, with notes on real-life applications in circumstances that normal humans are likely to encounter in companies where they might actually work.
- The credibility must be built up through the technical presentation, not through the "quality process" discussions that (at best) are likely to bounce off of students who are studying to become individual practitioners.
- At this time, I have not found a book worth adopting. (WTST participants skimmed 2 large tables worth of books reviewed for this course.)
- If there are new books coming that fill this gap, we were unaware of them.

Implementation of the Course

- Easy introductory material to introduce students to
  - Unit test tools, eclipse, source control
  - Problem decomposition, calibration
  - This has evolved to dominate the course. Other faculty were unsurprised that this would be a major issue without a decent text, because we are reteaching students how to program.
- Mid-level programming of increasingly complex problems
  - Astels provided good training wheels tasks. No current artefacts available
- Maintenance (no longer part of the course)
  - Open source tool 1000 statements (ideal) with comprehensible code.
- Final exam
  - Complex problem, takehome, open book, 2 – to – 4 weeks for the exam (hard deadline)
  - Build a test tool that does X
  - Review exam drafts in advance

In-class Labs

1. Testing basic code structures (e.g. conditionals, loops)
2. Intro. to refactoring
3. Problem decomposition, insertion sort as an example.
4. Debugging
5. Testing common data structures (e.g. arrays, lists, maps)
6. Refactoring existing code (two labs).

Homework

- 5 homeworks
  - Hello world!
  - Ticket Machine (conditionals)
  - Date Arithmetic (loops, Boolean expressions, coverage, oracles)
  - Files I/O (files, exceptions, oracles)
  - Parameterized tests (JUnit parameterized tests, data-driven tests, files)
- 2 assignments.
  - Random Numbers Generator
  - Refactoring and code Smells

FINAL EXAM

- Take home individual project.
- Tuned down of the 2008 version.
- Grading addressed mainly the following points:
  - ➢ Completeness (how many requirement items were implemented)
  - ➢ Course objectives
    - Correct using of TDD
    - Industry standard tools (checkstyle, branch coverage, svn)
    - Problem decomposition
    - Technical communication
    - Glass box techniques
    - Refactoring
    - Good test design
    - Well designed code

RESULTS

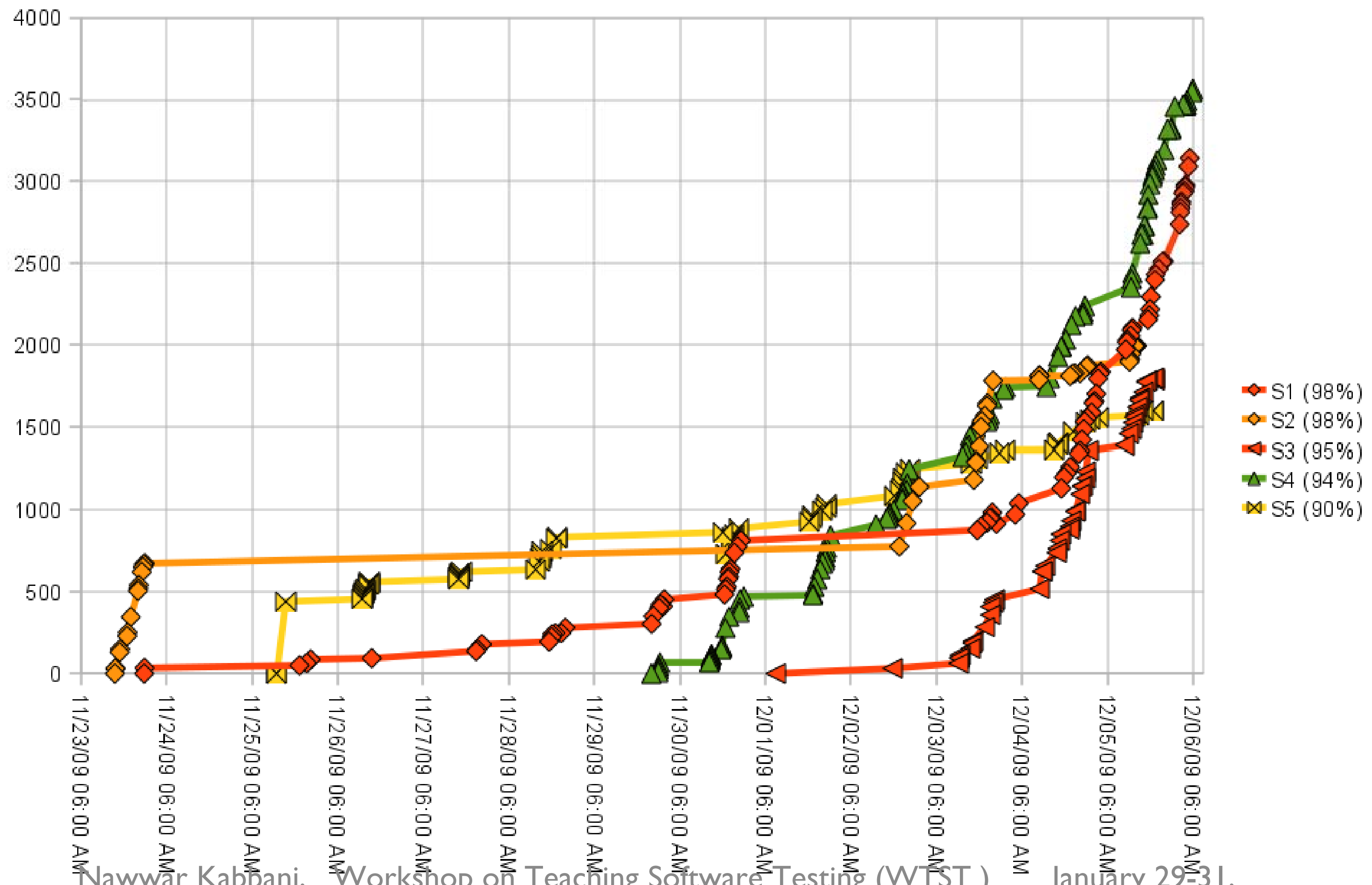
- Several students are fully successful
  - ➢ Good jobs at good pay after (this, unfortunately, attracts other unsuitable students)
- Many students are resistant
- Many students are inadequate programmers and attempt to stick to development strategies that will not work in this course
  - ➢ We review the source code repository for each assignment, to assess refactoring and evolution. Students who buy code or cut-and-paste code fail b/c of no TDD
- Stunningly many students start at the last minute despite midterm exam and assignment experiences

EXAM RESULTS

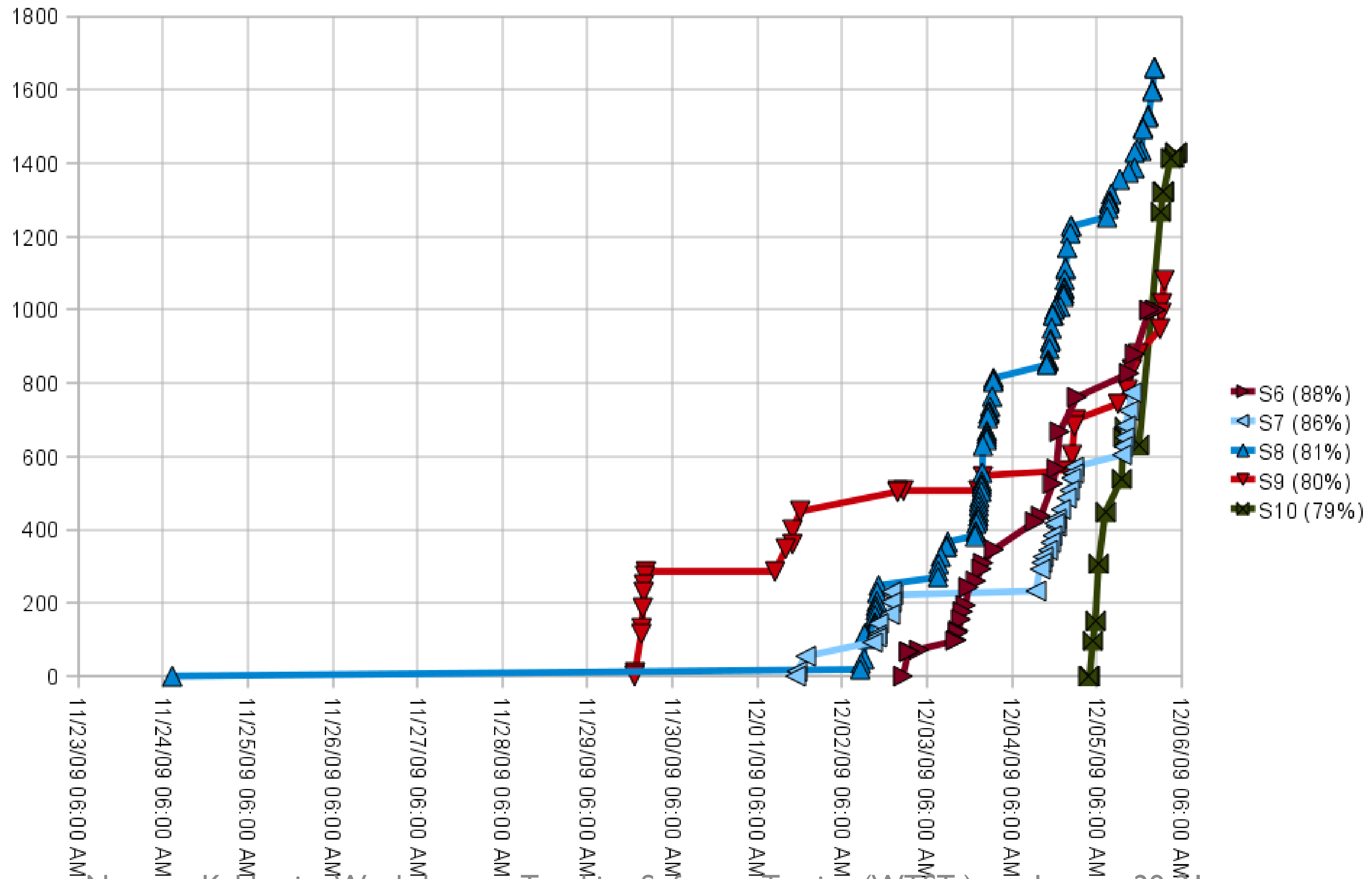
Student (score%)	SVN Check-ins (commits)	LOC (non-blank)	LOC / Commit
S1 (98)	83	3143	37.87
S2 (98)	52	1996	38.38
S3 (95)	50	1785	35.7
S4 (94)	116	3567	30.75
S5 (90)	61	1600	26.23
S6 (88)	28	999	35.68
S7 (86)	34	774	22.76
S8 (81)	105	1660	15.81
S9 (80)	33	1080	32.73
S10 (79)	19	1429	75.21
S11 (52)	21	884	42.1
S12 (52)	21	639	30.43
S13 (52)	73	415	5.68
S14 (30)	57	918	16.11
S15 (28)	18	385	21.39
S16 (16)	8	489	61.13

NB: 2 students didn't submit the final

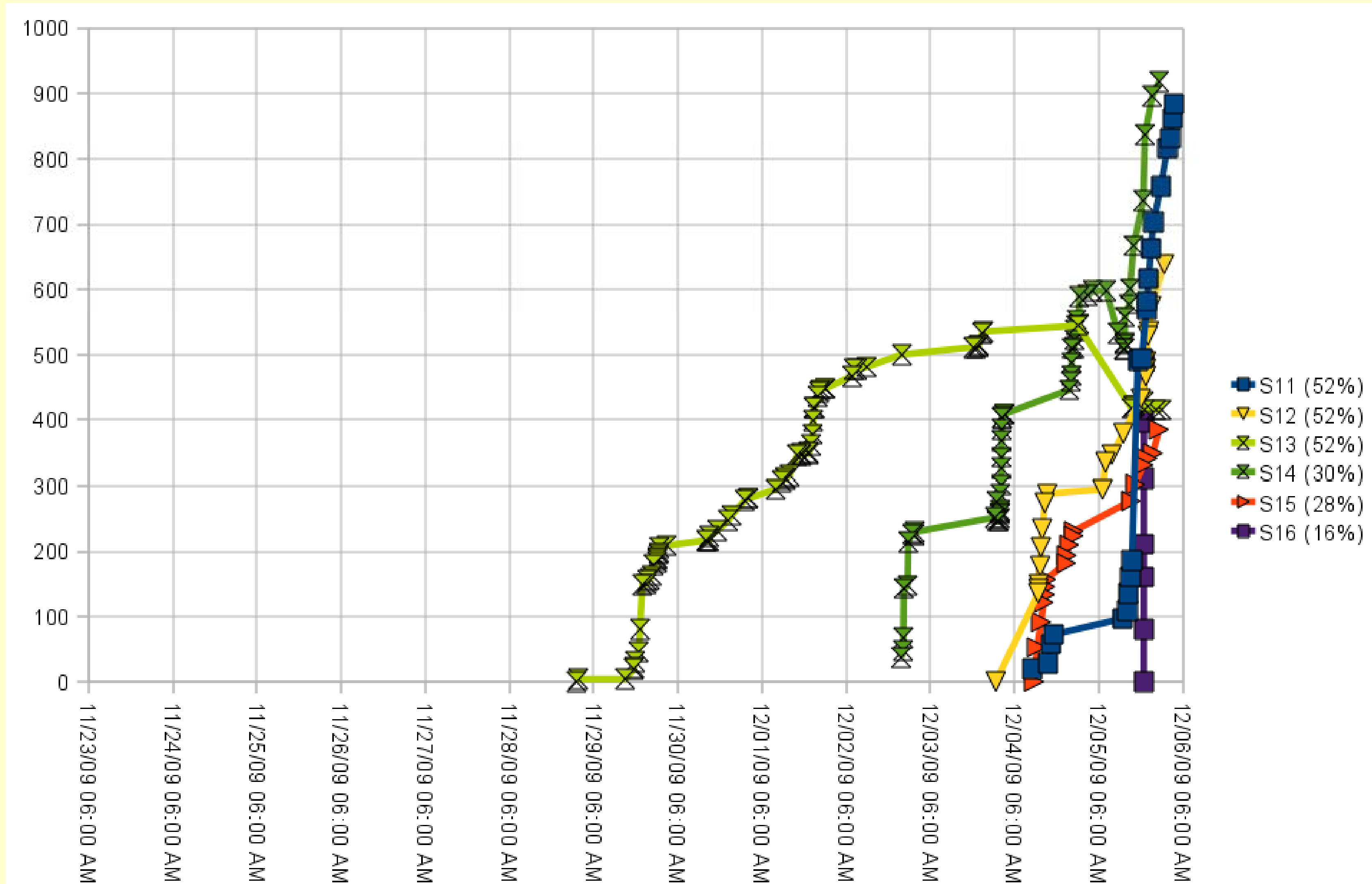
Subversion commits over time – top 5 scores



Subversion commits over time – middle scores



Subversion commits over time – bottom 6 scores



What do we need? (Presentation at WTST)

- Instructional support
  - Textbook for 1st year students in Java, C# or Python
  - Textbook for somewhat-more experienced programmers (2nd or 3rd year students), but it cannot assume detailed knowledge of programming concepts (profile of returning students makes this unwise)
  - Workbooks to support commercial or online instruction
- Learning objects
  - Risk analyses for common programming language constructs (how can they fail?) This need not be language-specific
  - Kept-up-to-date examples (video demonstration) of test-driven development of sample programs
  - Better examples (with videos) of refactorings, without assumptions of strong familiarity with the programming constructs or the authors' favorite patterns
- Testimonials targeted to students instead of business clients and funding agencies

Where do we go from here?

- Drop emphasis on TDD,
  - refocus on implementation level tests in context of student-written test tools
  - Change textbook focus (we will still have big problems)
- Many of the problems with student work, especially M.Sc. work, will not go away by changing dev process
- We will need a new control method to detect cheating
- Your thoughts are welcome

Copyright (c) Cem Kaner