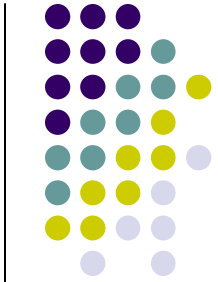
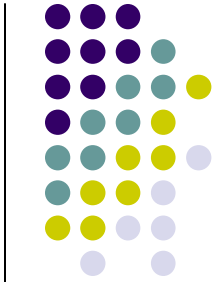


Automated Unit Testing and the TSP

presented by
Noopur Davis
and
Larry Maccherone

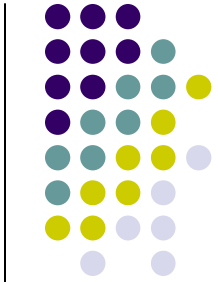
TSP Symposium
September 2007

Outline



- Background
- Changes to TSP
- Results from TSP teams
- Industry results
- Research topics

Unit Testing (UT)



IEEE definition of unit testing

Testing of individual

- *hardware or software units*
- *or groups of related units*

Common understanding of unit testing

- done by developers
- done on very small units of code
- goal is to ensure that isolated units of work are functioning correctly

Automated Unit Testing (AUT)

Automates the task of unit testing

- tests are usually written in the same language as production code.

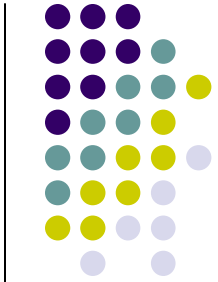
Tests are written to exercise units of code

- in procedural languages, these could be functions and procedures
- in object-oriented languages, these are frequently methods and classes.

Test Driven Development (TDD)

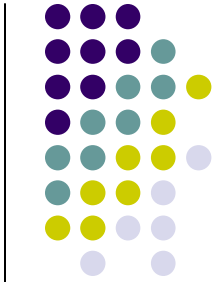
- What is TDD?
  - A strategy for software development where you write the tests before writing any production code
  - You expect the tests to fail the first time they are run
  - Tests serve as requirements or design artifacts
  - Force one to think about functionality and API **before** thinking about implementation
- TDD requires automated unit testing, but...
  - Not everyone doing automated unit testing is doing TDD

Why the Buzz?



- Most Agile methods strongly support automated unit tests, and some explicitly call out for TDD.
- Neither AUT nor TDD are new
  - The Agile community, in their own words, has "rediscovered" these
- "It is desirable to develop the tests before you write the code". *A Discipline for Software Engineering, page 370*

Testing Frameworks

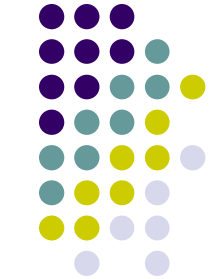


Automated unit tests are supported by testing frameworks that help with

- setup and teardown
- method and class-level testing
- family of assertions and generation of exceptions
- ability to extend the framework

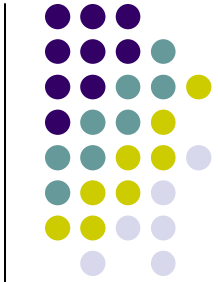
The most popular family of frameworks is the x-Unit family (junit, nunit, cunit, phpunit, flexunit, etc..)

Junit Example



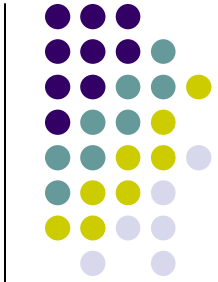
```
// derived from example provided by Frank P. Coyle, PhD (http://engr.smu.edu/~coyle)
public class LibraryTest extends TestCase {

   private Library library;

   public void setUp( ) throws Exception {
      library = new Library( );
      library.addBook(new Book( "Cosmos", "Carl Sagan" ));
      library.addBook(new Book( "Contact", "Carl Sagan" ));
      library.addBook(new Book( “Contact", “Jena Malone" ));
   }

   public void tearDown( ) { library = null; }

   public void testGetBooksByTitle( ) {
      Vector books = library.getBooksByTitle( “Contact" );
      assertEquals( "wrong number of books found", 2, books.size( ) );
   }

   public void testGetBooksByAuthor( ) {
      Vector books = library.getBooksByAuthor( "Carl Sagan" );
      assertEquals( "2 books not found", 2, books.size( ) );
   }
    // Junit also provides assertTrue, assertFalse, assertNull, and a few more
}
```

Adding AUT to TSP



- Considerations for
  - Process framework
  - Planning framework
  - Quality framework
  - Measurement framework

Operational Details



Automated unit-tests are written along with production code in very tight increments

1. Write a couple of lines of production code*
2. Write a couple of lines of test code
3. Build and execute (most testing frameworks do this automatically)
4. Refactor both test and code if needed
5. Repeat

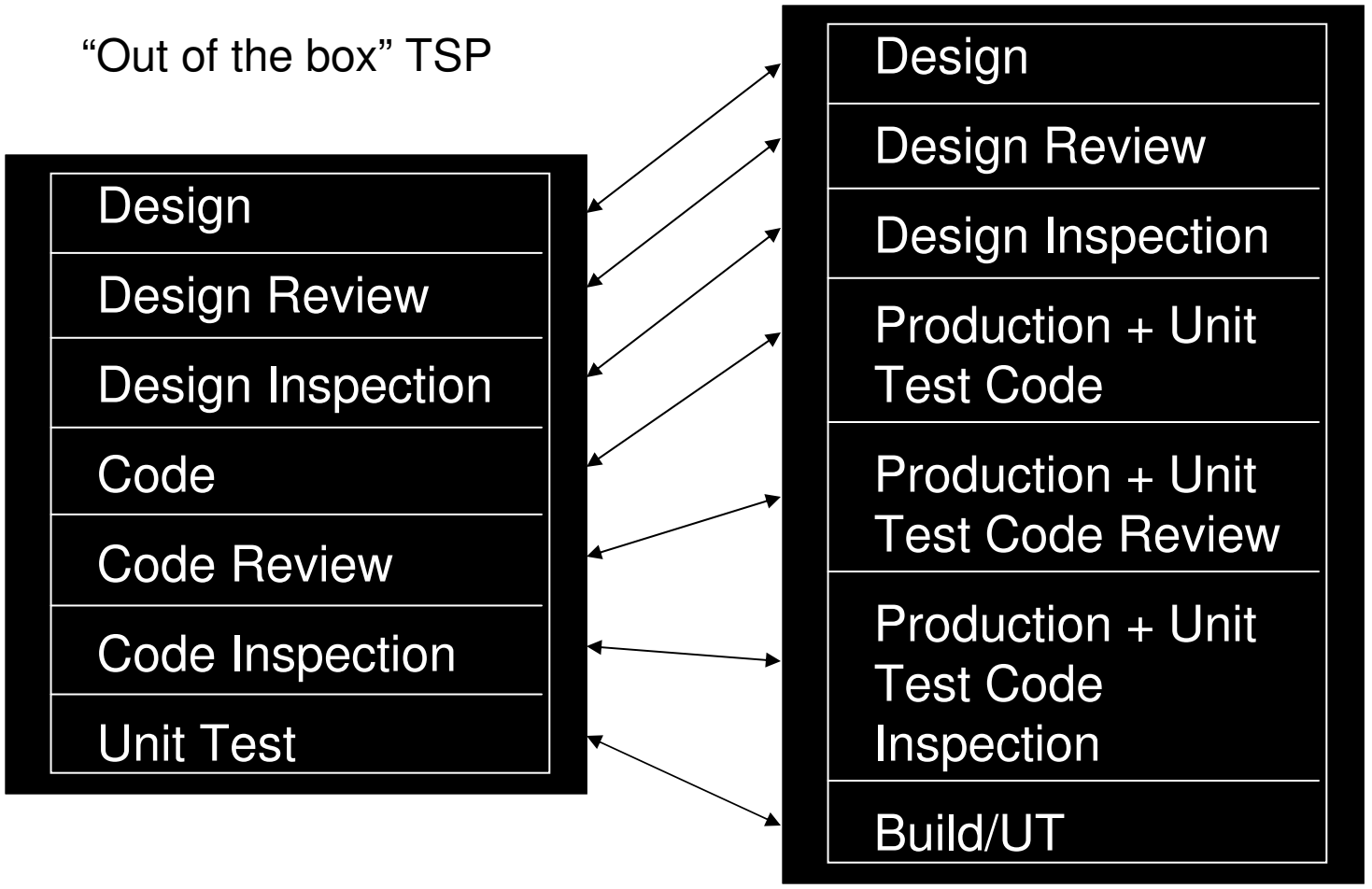
*for TDD, the order would be 2, 3, 1, 3, 4, 5

AUTs and Builds

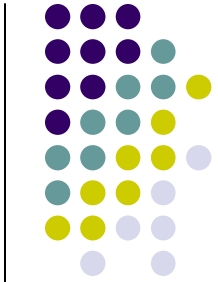
AUTs must almost always be coupled with a build system that automatically builds and executes all unit tests (regression)

- Continuous builds
- Multiple builds a day

Process Considerations

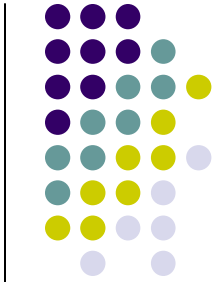


TSP Teams Data - LOC



Name	Prod. LOC	UT LOC	UT Loc/Prod LOC
Subsystem 1	1388	2126	1.53
Subsystem 2	1634	940	0.58
Subsystem 3	1863	1208	0.65
Subsystem 4	2009	1794	0.89
Subsystem 5	2667	781	0.29
Subsystem 6	3022	1442	0.48
Subsystem 7	3520	1851	0.53
Subsystem 8	4789	2197	0.46
Subsystem 9	7609	6125	0.80
Subsystem 10	12990	8481	0.65
Subsystem 11	17490	16233	0.93
Subsystem 12	55602	72269	1.30
		Average	0.76
		Max	1.53
		Min	0.29

TSP Teams Data - Coverage

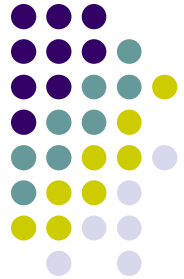


	Conditionals	*Statements*	*Methods*	*Total*
Subsystem 1	**97.6%**	**98.3%**	**100%**	**98.2%**
Subsystem 2	**50%**	**84.6%**	**95.3%**	**84.9%**
Subsystem 3	**66.9%**	**88.6%**	**91.5%**	**83.9%**
Subsystem 4	**62%**	**75.7%**	**89.6%**	**76%**
Subsystem 5	**40.7%**	**65.9%**	**80.5%**	**66.2%**
Subsystem 6	**66.6%**	**77%**	**83.5%**	**76.4%**
Subsystem 7	**60%**	**67.4%**	**63.9%**	**66%**
Subsystem 8	**66.7%**	**72.2%**	**100%**	**73.1%**
Subsystem 9	**76.7%**	**80.2%**	**100%**	**81.2%**

Industry Data – Microsoft TDD Case Study[1]

	Windows	MSN
Test LOC/Source LOC	0.66	0.89
Block coverage	79%	88%
Development time (person months)	24	46
Team size	2	12
Decrease in Defects/LOC	38%	24%
Increase in dev time	25-35%	15%

Industry Data – IBM Case Study[2]



	Test LOC/Prod LOC
Device driver 1	.54
Device driver 2	.09
Device driver 3	.59
Device driver 4	.22
Device driver 5	.76
Device driver 6	.63
Device driver 7	.88
Device driver 8	1.12
Device driver 9	.13
Device driver 10	.43

Results

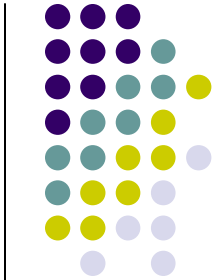
- For TSP teams, the results as measured by improved system test defect density are inconclusive
  - The results are the “best in class” for TSP teams
  - They are not significantly better than other best in class teams that are not using AUT.
- The largest set of industry results from 19 case studies, controlled experiments, simulation, and artifact analysis shows*
  - Productivity decreased by 19% (-27% to 90%)
  - Quality improved by 25%

*Some data is based on “perception”

Lessons Learned

- Almost all serious testing efforts end up extending the test framework
- Not all tests can be automated
- Create a new test whenever a defect is detected that escaped the test suite
- Must have testable designs
- Hard to add to legacy
- AUTs result in “better” APIs, help document the code better, and do seem to help in code maintenance.

Planning Considerations



Size estimation

- Rule of thumb: plan to write as much unit test code as production code

Productivity

- Rule of thumb: unit test code is about 4 times faster to write than production code
- Plan for chunks of time to
  - Setup and learn test frameworks
  - Integrate AUTs into build system
  - Major re-factoring of tests every few iterations

Time-in-phase distribution

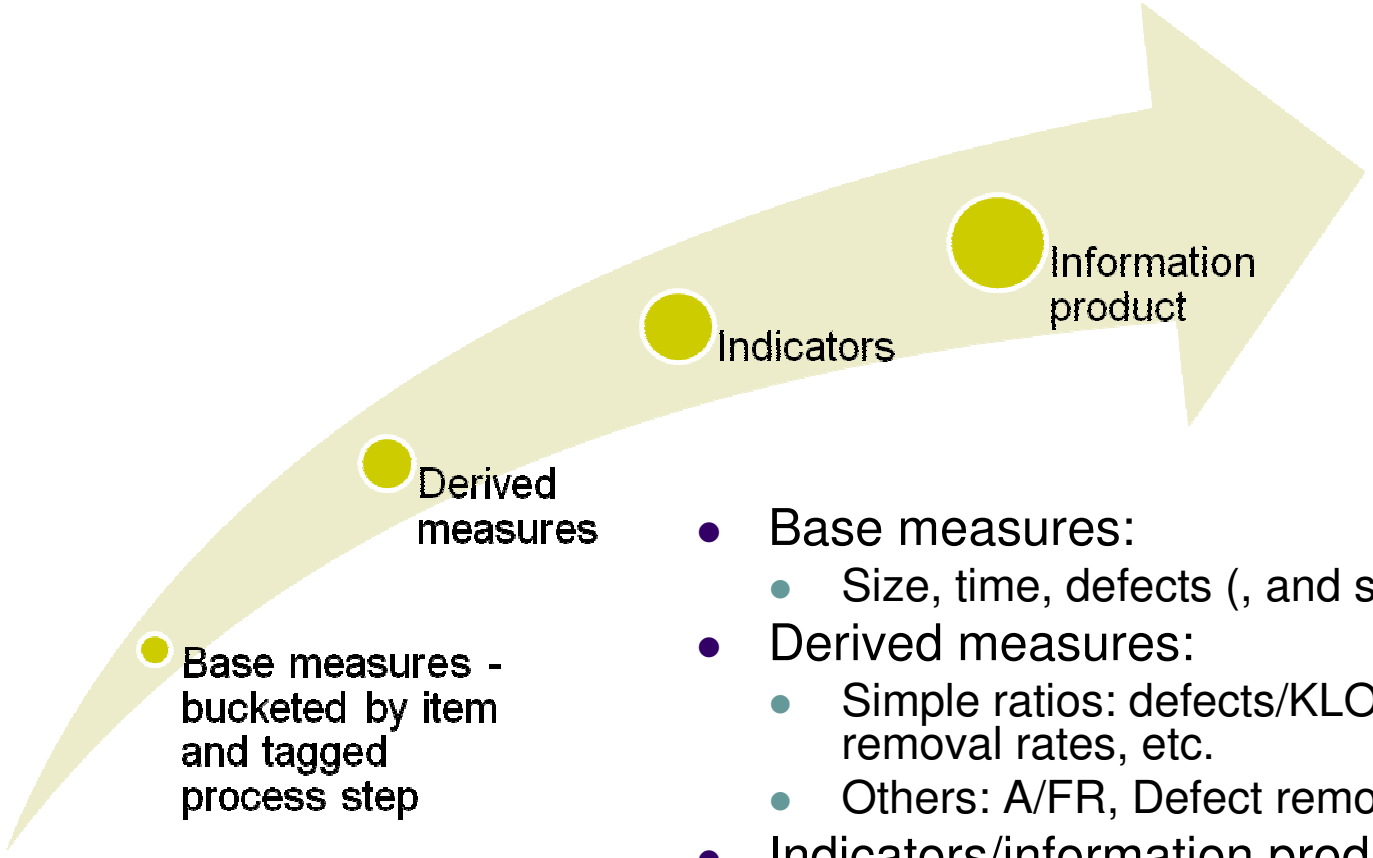
- Increase time in code phase
- Decrease time in unit test phase

Code coverage

- Most teams are aiming for 80%

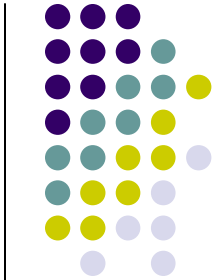
How can we get more empirical than "rules of thumb"? Larry will talk about this next.

TSP
Measurement Information Model



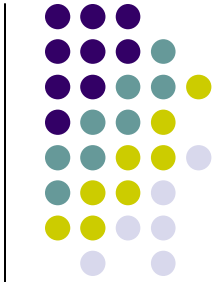
- Base measures:
  - Size, time, defects (, and schedule)
- Derived measures:
  - Simple ratios: defects/KLOC, LOC/hr, defect removal rates, etc.
  - Others: A/FR, Defect removal leverage, PQI, etc.
- Indicators/information product:
  - CR more efficient than UT at removing defects
  - Enough time spent on team inspection
  - Will (not) finish by planned completion date

Current TSP Information Model



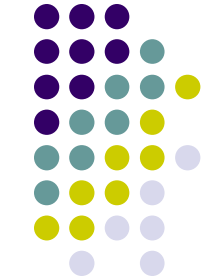
- Historically effective
  - Encourages culture of review/inspection
  - Earned value tracking provides industry best progress feedback
- Adaptable to
  - Changes in process definition
  - Product as well as non-product activities
- Limitations with respect to emerging technology
  - Time spent writing automated unit tests should not be considered “failure”
  - Awkward or even misleading cost of quality formula
  - No ROI for future benefit

Unanswered Questions

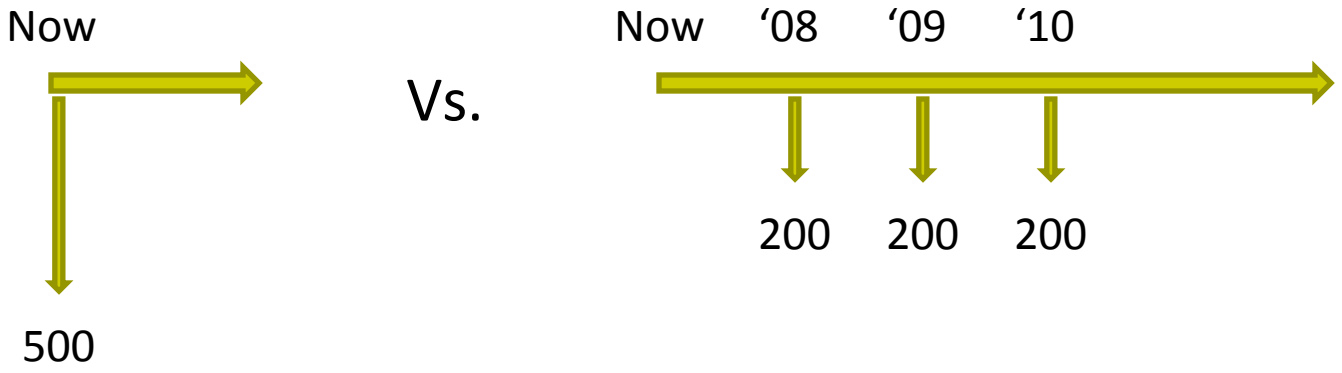


- Project questions:
  - Cost/benefit of automated unit testing?
  - Effectiveness (or ineffectiveness) of automated code analysis?
  - How much automated unit testing and analysis to do?
  - Value of refactoring?
  - Reduce other appraisal activities in the presence of these? How much?
- Process questions:
  - New base measures needed? Or are simple bucketing and tagging changes sufficient?
  - What derived measures and indicators are necessary?
  - Do these proposed changes accommodate other new practices and technologies that are on the horizon?

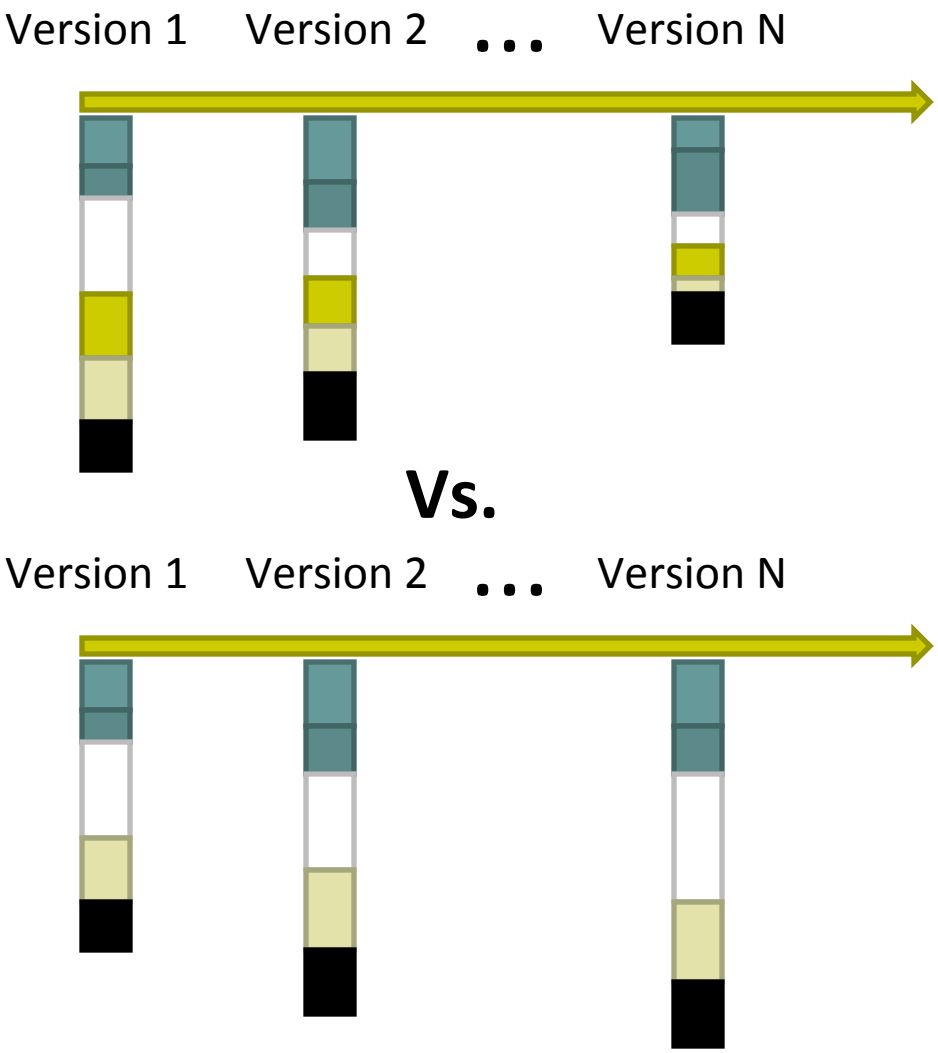
Delayed Gratification



- Currently cost/benefit in TSP is only hinted at
  - Defects removed per time exerted in the current iteration
- A true ROI for defect prevention activities would compare two effort flows the same way we'd compare two cash flows in Economics 101.

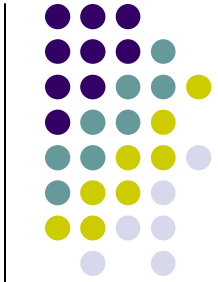


Comparing Effort Flows



Design
Reviews
Prod. Code
Test Code
Unit test
Other

Relationships



Time invested in this	Should save time in this
Design	Code
Design	Fixing future defects
Review/inspection activities	Fixing future defects
Writing automated unit tests	Fixing future defects
Creating custom analysis rules	Fixing future defects
Writing automated unit tests and custom analysis rules	Refactoring
Refactoring	Adding new features

← The Agile message

How to Conduct Experiments?

1. Controlled experiment(s)
   - Have two groups (or more) develop the same thing in iterations. One with AUT, one without.
2. Longitudinally in a single project
   - Treat different parts of the code (but of same type) as separate efforts
   - Track effort in future iterations modifying or consuming those parts
   - This will require a very high level of traceability and automated data gathering from SCM, IDE, Build tools, etc.

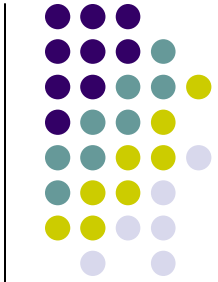
Questions We Hope to Answer

- Production:Test code ratio:
  - What is the “ideal” ratio? 3:1? 1:1?
  - Does it matter what type of part you are building?
  - How do “time value of effort” calculations change the picture?
  - Is “ideal” different when Quality (as opposed to long-term cost/benefit) is of paramount concern?
- Coverage:
  - What is ideal coverage? 80%? Higher?
  - What characteristics indicate the need for higher or lower coverage?

Looking Ahead

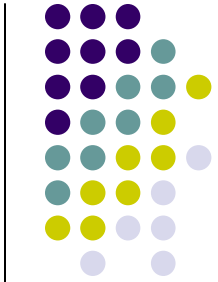
- Automated gathering of data from IDEs, build tools, unit test output, analysis tool output
- Configurable tagging and historical relationship calculation
- Perspectives
  - Q: Do you count test writing as “Coding” or as “Testing” or what?
  - A: Count it as “Coding” if the current iteration is for writing tests. Count it as a Cost of Quality activity from the perspective of the entire project.
- Easy to use tools for teams to do flexible ex-post-facto analysis

References



1. T. Bhat and N. Nagappan, "*Evaluating the efficacy of test-driven development: industrial case studies*" *ACM/IEEE international symposium on empirical software engineering*, Rio de Janeiro, Brazil, 2006, pp. 356 – 363
2. Sanchez, Williams, and Maximilien, "*On the Sustained Use of a Test-Driven Development Practice at IBM*", Agile 2007 Conference, Washington, D.C.
3. R. Jeffries and G. Melnik, "*TDD: The Art of Fearless Programming*", IEEE Software, May/June 2007

Contact



Noopur Davis

- nd@sei.cmu.edu
- ndavis@davissys.com

Larry Maccherone

- LMaccherone@cmu.edu
- Larry@Maccherone.com