Com S 417
Software Testing

Fall 2017 – Week 2, Lecture 4

Announcements

- Lab due Sept 5 midnight.
- Homework due Sept 7
- My office hours effective now:
  - Tues 2:10 – 3,
  - Wed 4-5, and
  - by appointment
- Reminder: Syllabus schedule is TENTATIVE – and not reliably updated.

Questions

Reading, Last Lecture, Lab?

Evaluating the Quality of Tests

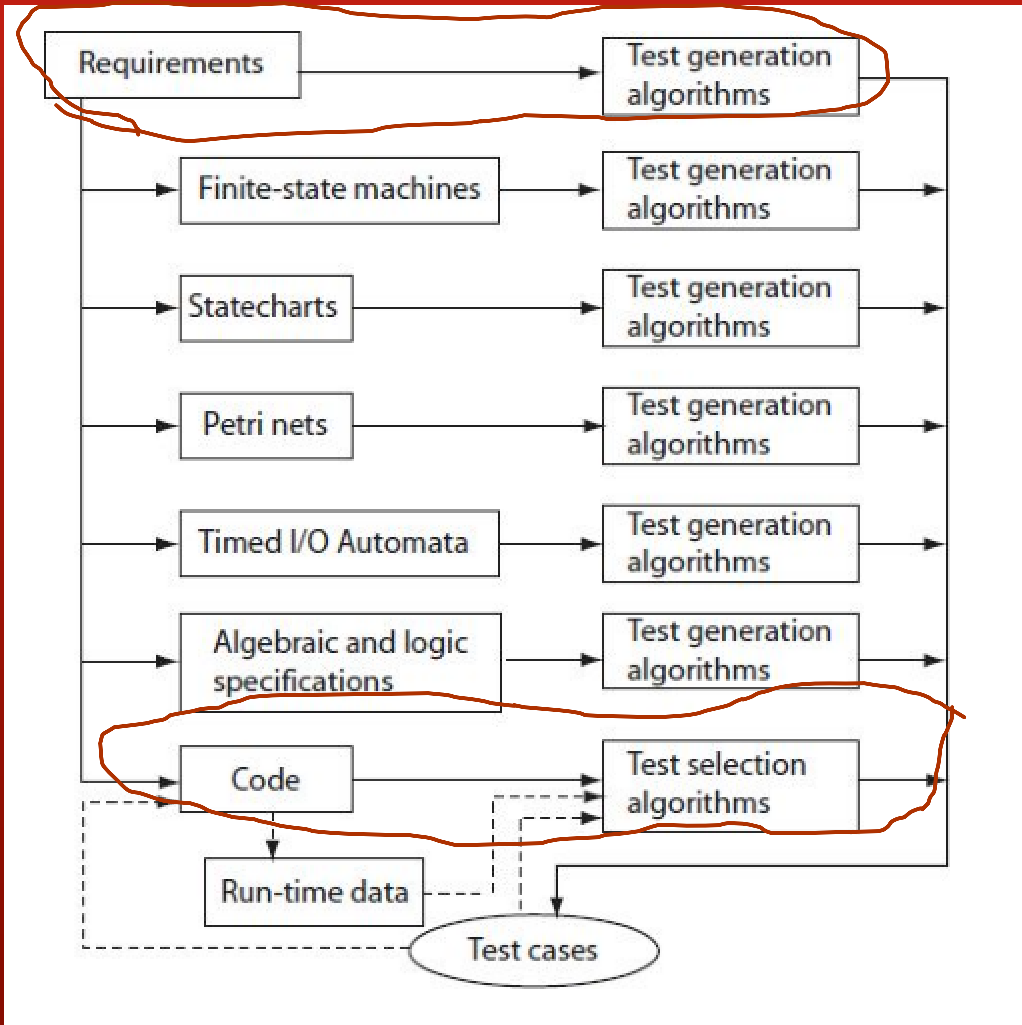
Requirements; Black Box vs. White Box; Input Partitioning

Talking About Tests

- Source of test generation
- Life cycle phase in which testing takes place
- Goal of a specific testing
- Characteristics of the artifact under test
- Test process

White Box Techniques

- Dynamic Test strategies
  - Control Flow Testing
  - Predicate Testing
  - Data Flow Testing
  - Tests from FSM
  - Tests from Decision Tables
  - Mutation Testing



Tests by Dev. Phase

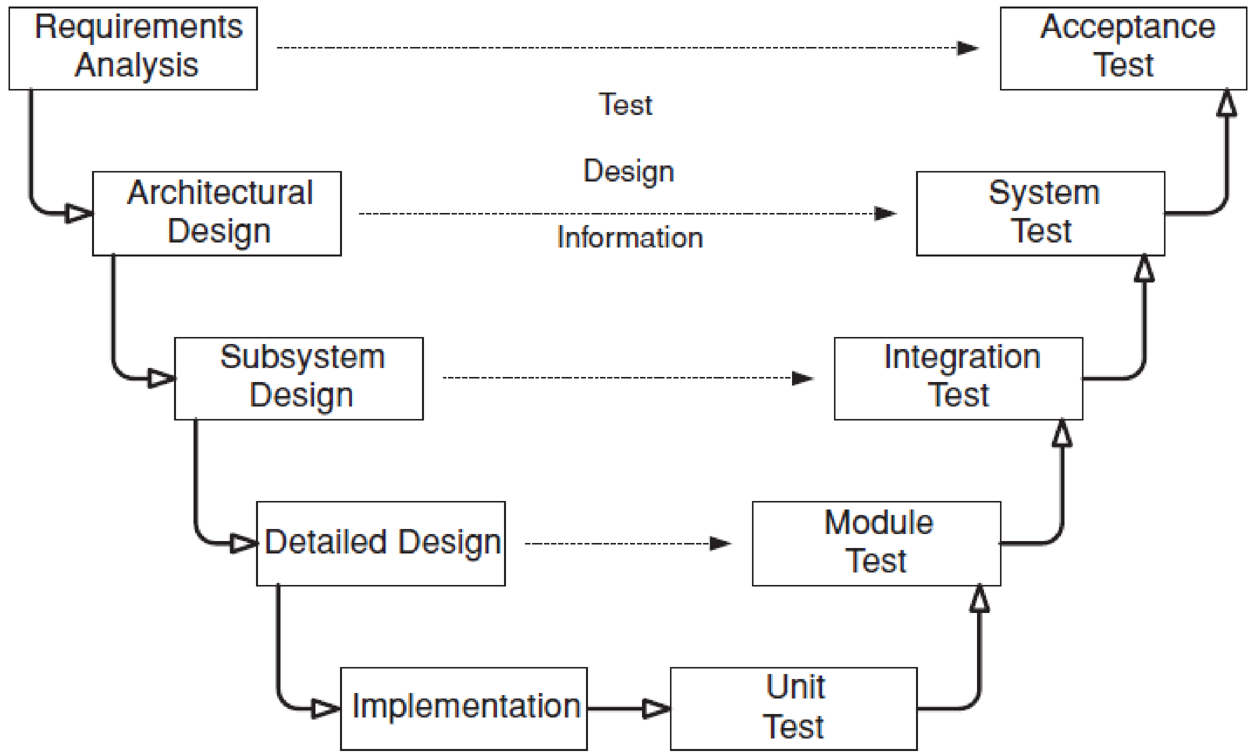


Figure 1.2. Software development activities and testing levels – the "V Model".

From A & O, p. 6

Performance Tests

- Stress tests - load system using many users, devices etc – see how it fails
- Recovery tests - response to faults and loss of data
- Load tests - load system using many users, devices etc
- Volume tests - test ability to handle large amounts of data
- configuration tests - test s/w and h/w configs
- compatability tests - test interfacing with other systems
- security tests
- reliability tests - up-time (Mean Time To Failure)
- Usability tests - test user interfaces

Regression Tests

"Regression testing refers to that portion of the test cycle in which a program P′ is tested to ensure that not only does the newly added or modified code behaves correctly, but also that code carried over unchanged from the previous version P continues to behave correctly."

- Note chapter 5 in the text is dedicated to regression tests!
- Traditional vs. Agile realization.
- What about TDD?
- What about Test First Development

quote from the textbook

And More …

- Section 1.18 gives many more.
    - All are ‘fair game’. If the text isn’t clear to you, ask.

Code Coverage

Evaluating test sets

Imagine you know exactly how many bugs are in a particular piece of software and that you have two different test sets you can use to test the software.

(Note: this would imply knowledge of the specific implementation.)

How would you decide which was the better test set?

If you knew in advance how many bugs were in a piece of software how could you measure the quality of any particular test set?

‘Ideal” coverage

- An ideal (not practical) test set would guarantee to find all bugs.
- Thus if we knew how many bugs we were seeking we could measure the test set quality as
  - bugs found / total bugs in software.

But we can never know how many bugs are in a particular piece of software.

Can we still use the same general idea to compare test sets?

Coverage
as metric from surrogates

- Often when we can’t know one piece of information, we can find some other related information and use it in place of the ideal information.
- Studies show there is a reasonable degree of correlation between lines of code and the total number of bugs.
- We know that a prerequisite for finding a bug is reaching the fault.
- So … what about using these two surrogates to construct a measure of test quality?

Ideal Coverage vs *Code* Coverage

Ideal Measure
$(\textit{not realizable})$

$$\frac{\textit{Discovered Bugs}}{\textit{Existing Bugs}}$$

Surrogate Measure
$(\textit{practical})$

$$\frac{\textit{Lines of code Reached}}{\textit{Total lines of code}}$$

Question:

Does 100% code coverage guarantee the test set finds all bugs?

Assumptions in Code Coverage

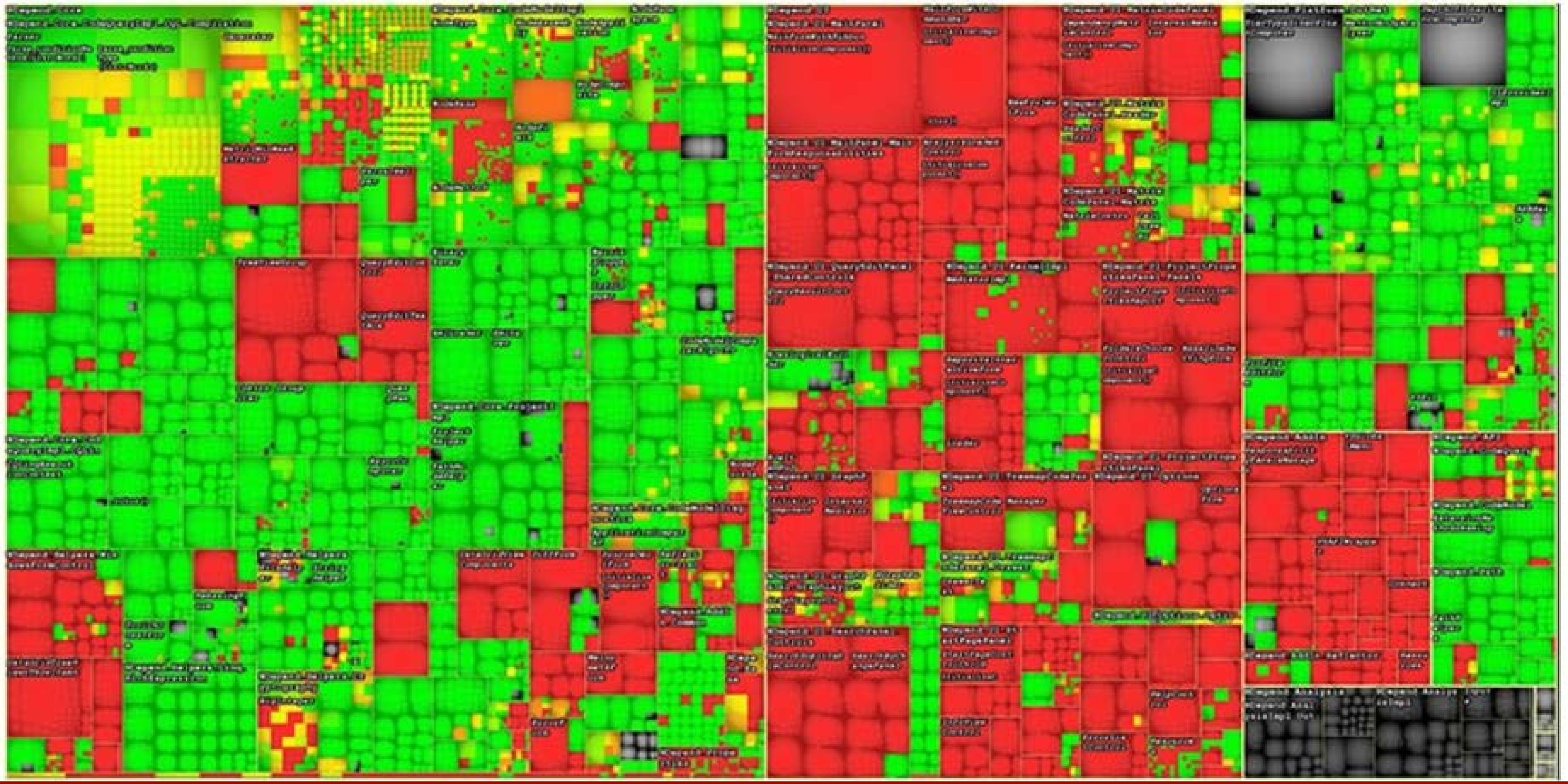
- A high percentage of bugs propagate if reached
  - Generally true.
- Bugs are uniformly distributed in the code
  - Not true, they tend to cluster
    - logic expressions
    - other types of complexity
    - developer capabilities
  - To compensate, we combine code coverage with conditional coverage and demand higher levels of coverage and inspection on code with higher complexity.

Code Coverage Tools

- Code coverage tools track what lines are executed as tests are executed. They then generate a report computing the percent coverage (usually for different types of coverage) and show you where you might need more tests.
  - These typically require the code be specially compiled (or run with certain virtual machine capabilities enabled).
  - They may impact run time.
  - To get ‘big picture’ results, you need to run all tests in the same “logging session.”

Visualization

Code Coverage Heat Map

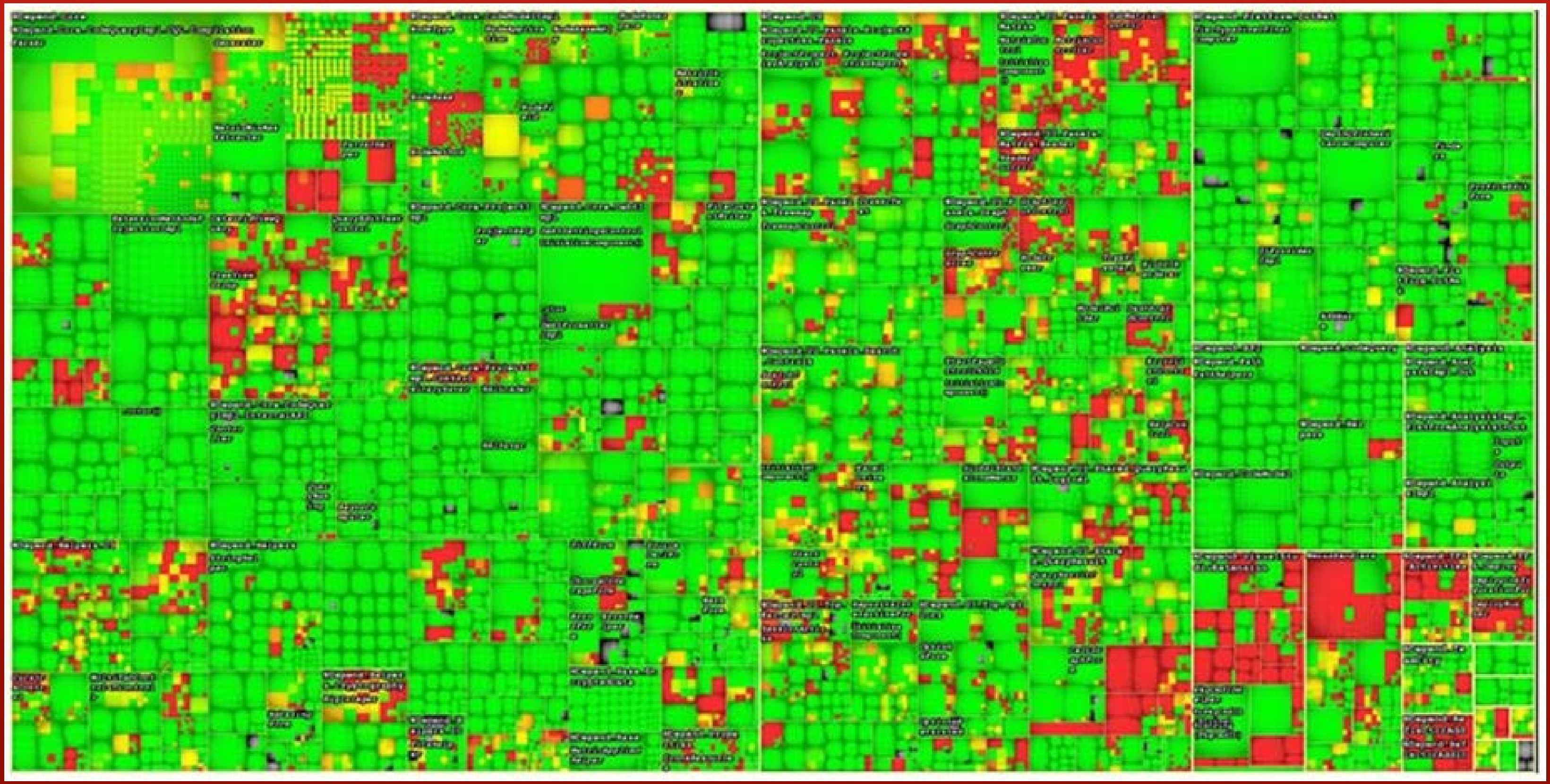


49.8% Code Coverage

image from: https://blog.ndepend.com/code-visualization-ndepend-treemap/

Visualization

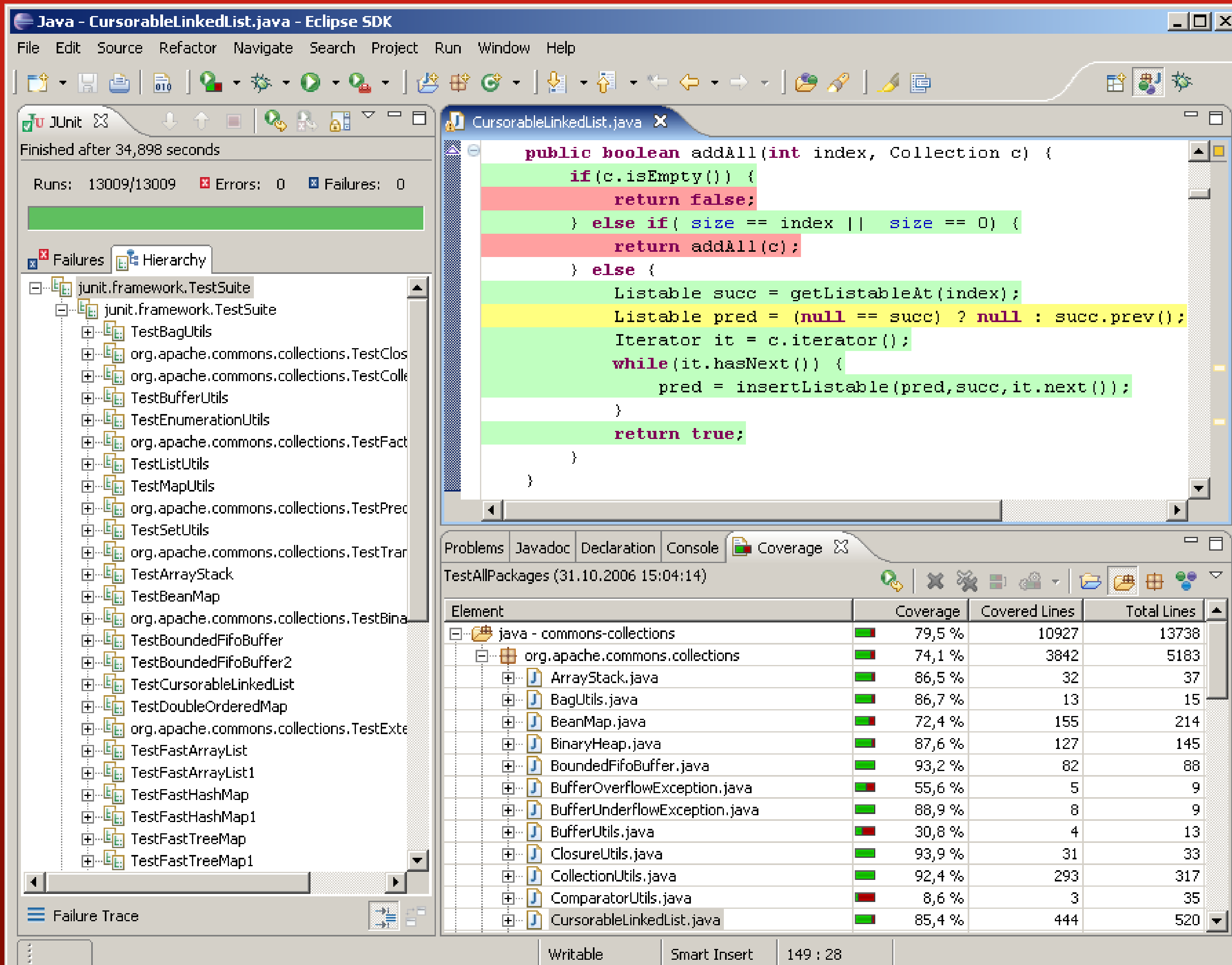
Code Coverage Heat Map



84.52% Code Coverage

image from: https://blog.ndepend.com/code-visualization-ndepend-treemap/

EclEMMA Eclipse Coverage Tool



Code Coverage Tool Terms

- LOC (lines of code)
- NCLOC (Non Comment lines of code)
- Branch Coverage
  - Each path from an if (or switch) was executed.
- Conditional Coverage (sometimes predicate coverage)
  - Each boolean subexpression in a conditional was placed in both true and false state at some time.

Predicate Coverage

Example

```
if ((A || B) && C)
{
  << Few Statements >>
}
else
{
   << Few Statements >>
}
```

Result

In order to ensure complete Condition coverage criteria for the above example, A, B and C should be evaluated at least once against "true" and "false".

```
So, in our example, the 3 following tests would be sufficient for 100% Cond
A = true  | B = not eval | C = false
A = false | B = true     | C = true
A = false | B = false    | C = not eval
```

From: https://www.tutorialspoint.com/software_testing_dictionary/condition_coverage_testing.htm

White Box Techniques

- Static Tests
  - Flow graph based
    - Cyclomatic complexity
    - Synchronization issues (e.g. findbugs)
  - Lexical Analysis
    - compilation
    - style rules
    - copy/paste detection
    - common mistakes (= when == was probably needed).

Static Tests

Cyclomatic Complexity

Cyclomatic complexity is a software metric

- the value computed for cyclomatic complexity defines the number of independent paths in a program.
- Cyclomatic complexity, $V(G)$, for a flow graph G is defined as

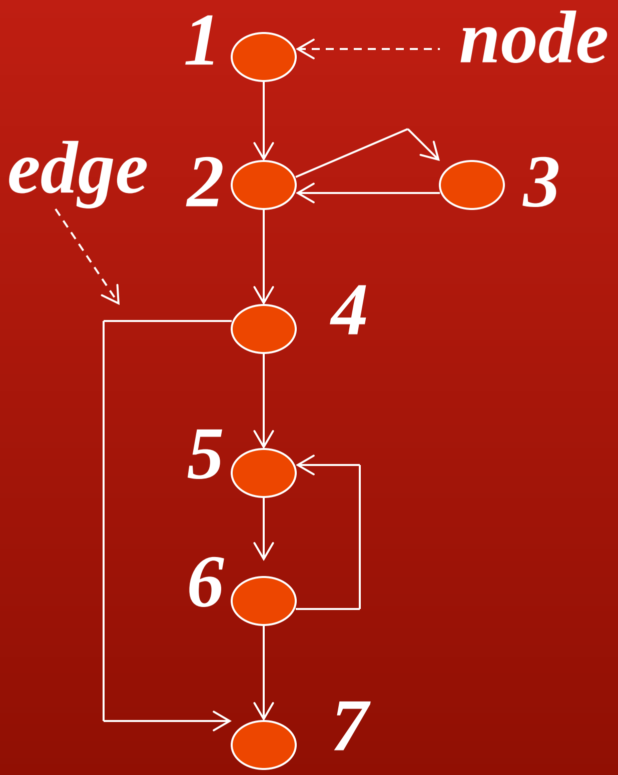
  $V(G) = E - N + 2$

  where E is the number of flow graph edges and N is the number of flow graph nodes.
- Alternatively, Cyclomatic complexity can be computed from the count of predicate nodes:

  $V(G) = P + 1$

  where P is the number of predicate nodes contained in the flow graph G.

An Example of cc



No. of edges = 9

No. of nodes = 7

No. of predicate nodes = 3

P-Nodes = {2, 4, 6}

V(G) = 3 + 1 = 4

V(G) = 9 - 7 + 2 = 4

From: Materials prepared by Simatra Mitra