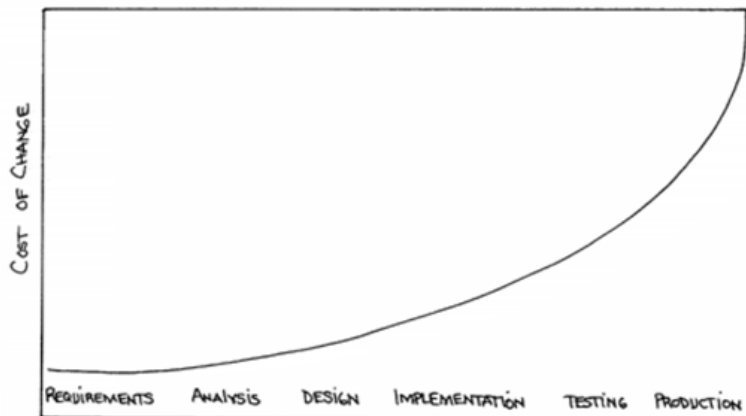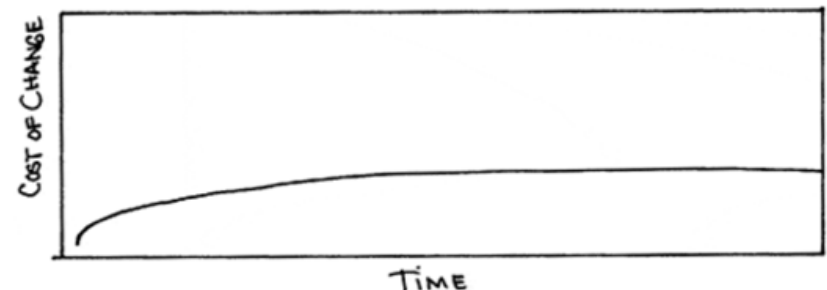# Economy and Test

- Cost/benefit
  - We want to find as **many faults** as possible with as **little effort** as possible, i.e. cheapest ☺

- Static testing is often done before dynamic testing
  - keeps rework costs low as faults are detected at early stage

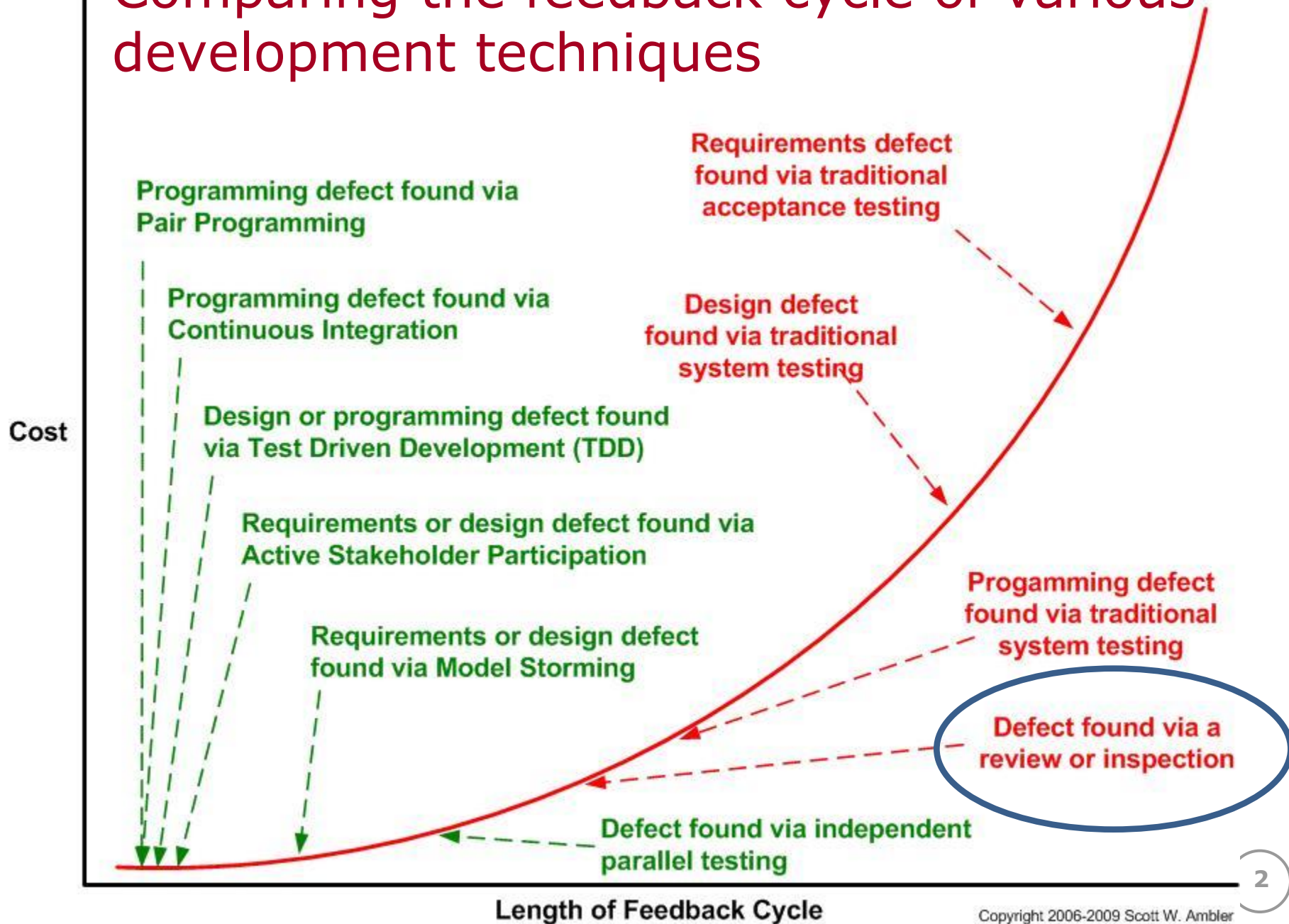- General assumptions of cost of change in development:

Traditional view   **Agile view**

Comparing the feedback cycle of various development techniques

Programming defect found via Pair Programming

Programming defect found via Continuous Integration

Design or programming defect found via Test Driven Development (TDD)

Requirements or design defect found via Active Stakeholder Participation

Requirements or design defect found via Model Storming

Requirements defect found via traditional acceptance testing

Design defect found via traditional system testing

Progamming defect found via traditional system testing

Defect found via a review or inspection

Defect found via independent parallel testing

Cost

Length of Feedback Cycle

2

Early feedback → confidence

Reviews

Formal reviews	Informal reviews
Formal reviews Well structured Regulated	**Informal reviews** No documented procedure More casual

- The formality is related to factors such as
  - Maturity of the development process
  - Legal or regulatory requirements

- Reviews often present milestones

Extract from job advertisement
- Write test case descriptions
- Conduct functional and non-functional tests
- Write and execute test automation
- Review documentation
- Troubleshoot customer installations together with Support
- Review, identify and refine functional, content and data requirements

Review is often used activity "out there"

Extract from job advertisement
- Analyzing product descriptions and specifications for new products and features
- Creating test plans and design of test cases
- Executing test cases, including stability, performance and scalability tests
- Handling of and follow-up on identified bugs
- Documentation review
- Test status reporting
- Configuration and maintenance of test lab equipment

Types of Reviews

❑ Walkthrough

informal

❑ Technical review

↓

❑ Inspection

formal

Walkthrough 1

- A **presentation** led by the <u>author</u> who presents his artifacts to:
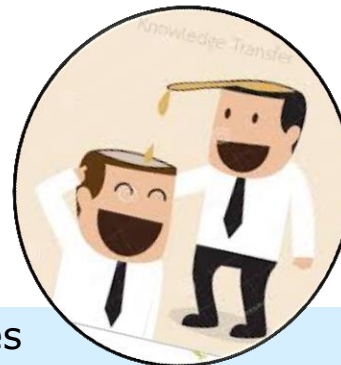
  1. Gather *information*

  2. Get *feedback*

  3. Establish a *common understanding*
     - knowledge transfer
     - education

Walkthrough 2

- **Good for higher level documents,** i.e.
  - Requirement specifications
  - Architectural issues

- **Participants**
  - IT and non-IT people

Technical Review

- A **discussion** **meeting** to
  - achieve <u>consensus</u> about the content of artifact
  - To find and fix <u>defects</u>
  - often as <u>informal</u> <u>peer reviews</u>

- **Focus** on
  1. <u>suitability</u> of technical concepts and alternatives

  2. <u>defects</u> in requirements, design doc, test plan, code etc.

- **Participants**
  - <u>Experts</u>: Architects, chief designers, lead developer, key users

Inspection – formal review

- **Reviewers** review artifact
  - Preparation time before meeting where artifact is checked thoroughly by reviewers

- A **review procedure** is followed
  - ✓ Faults are logged
  - ✓ Discussions can be postponed to discussion phase
  - ✓ Objective is to find faults, not solutions

- **Different goals**:
  1. Remove defects quickly (cheaper to fix early in process)

  2. Improve quality

  3. Establish a common understanding of topic (knowledge transfer/education)

A formal review process

- Roles
- Activities
- Responsibilities
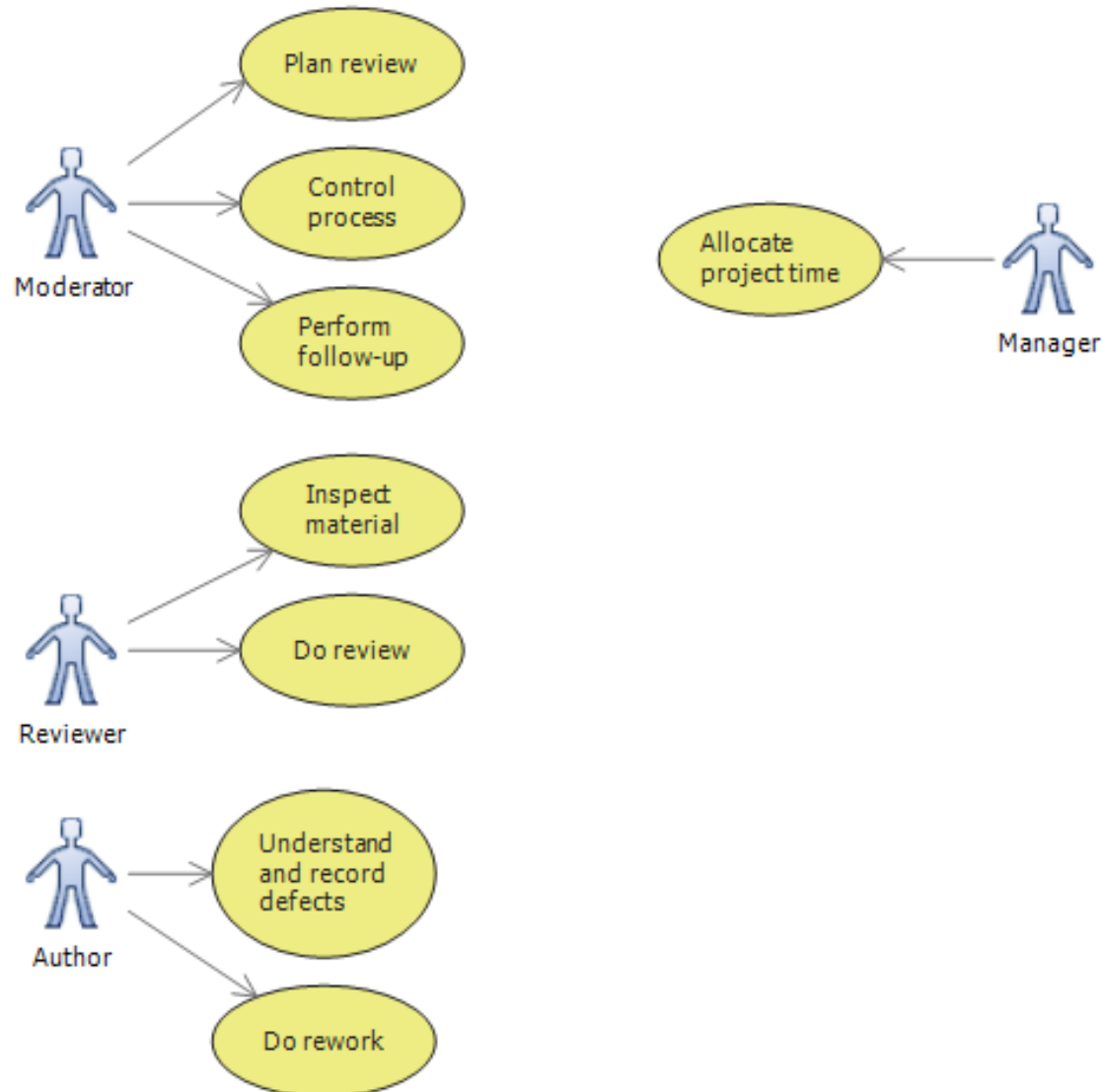
Requirement Review Exercise

Train Scenario

Pricing must be identified based on this scenario:

- If you take your train before 9.30 am or in the afternoon after 4.00 pm until 7.30 pm ('the rush hour') you must pay full fare. A saver ticket is available for trains between 9.30 am and 4.00 pm, and after 7.30 pm.

Problem

- How prepared are you from the above text to make test cases (you don't have to make all of them, but just consider possibilities)? Is anything unclear in the text?

Exercise

- **Work in pairs for 10 minutes as test reviewers for the train scenario.**

Best Practices for Peer Code Review
by SmartBear

1. Review fewer than 200-400 lines of code (LOC) at a time.
2. Aim for an inspection rate of less than 300-500 LOC/hour.
3. Take enough time for a proper, slow review, but not more than 60-90 minutes.
4. Authors should annotate source code before the review begins.
5. Establish quantifiable goals for code review and capture metrics so you can improve processes (e.g. metrics like inspection rate)
6. Checklists substantially improve results for both authors and reviewers.
7. Verify that defects are actually fixed!
8. Managers must foster a good code review culture in which finding defects is viewed positively.
9. Beware the "Big Brother" effect (don't single out developers)
10. The Ego Effect: Do at least some code review, even if you don't have time to review it all.
11. Lightweight-style code reviews are efficient, practical, and effective at finding bugs.

Source: http://smartbear.com/SmartBear/media/pdfs/WP-CC-11-Best-Practices-of-Peer-Code-Review.pdf

Static Code Analysis

Static Code Analysis

- Coding standards
- Code metrics

> Static analysis can be performed on requirements and design artifacts, but most tools focus on software code, so **our focus** will be on **static code analysis**

- Static code analysis is preferably done by the developers *before* reviews and test activities

Developer

Do static code analysis on code → Review code

Do static code analysis on code → Test code
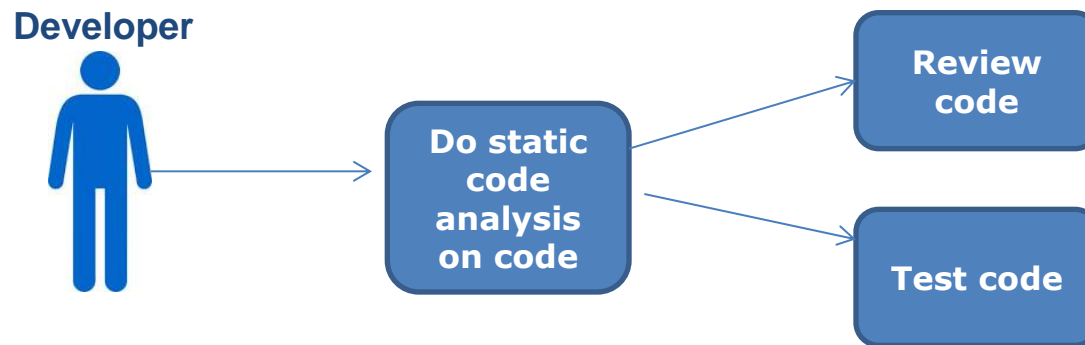
Anything wrong with this code?

```
package democodingstandards;
import java.util.ArrayList;


public class DemoCodingStandards {

    private void findAll() {
        DB db = new DB();
        ArrayList list = db.retrieve();
        int size = list.size();
        System.out.println("Number of records retrieved " + size);
    }


    public static void main(String[] args) {
        DemoCodingStandards demo = new DemoCodingStandards();
        demo.findAll();
    }
}
```

Coding Standards

- Tool can check for adherence to coding standards

Examples of coding standards document
1. https://google.github.io/styleguide/javaguide.html

2.

> 1. Use the Sun code conventions by default:
>   http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
> 2. Never catch exceptions without logging the stack trace or rethrowing.
> 3. Use dependency injection to decouple classes from each other
> 4. Avoid abbreviations unless well-known e.g. DTO
> 5. Methods that return Collections or arrays should not return null. Return empty collections and arrays instead of null
> 6. …

Demo of Static Code Analysis

- Demo in NetBeans

  Choose project/file → Source→ Inspect → Configuration → All Analyzers

Code Structure

- Often, it is not the size of the program which makes it complex to implement and test, but its <u>structure</u>.

- Several kinds of structural measures:

  Control flow structure
 – Sequence of instructions and nested levels

  Data flow structure
  – The trail of data item and the transactions to it

  Data structure
  – Complex data organizations

Code Metrics

- Code metrics
  - a set of software measures
  - provide developers better insight into their code and which parts of the code should be reworked or more thoroughly tested.

- Typical metrics types:

  - Depth of nesting
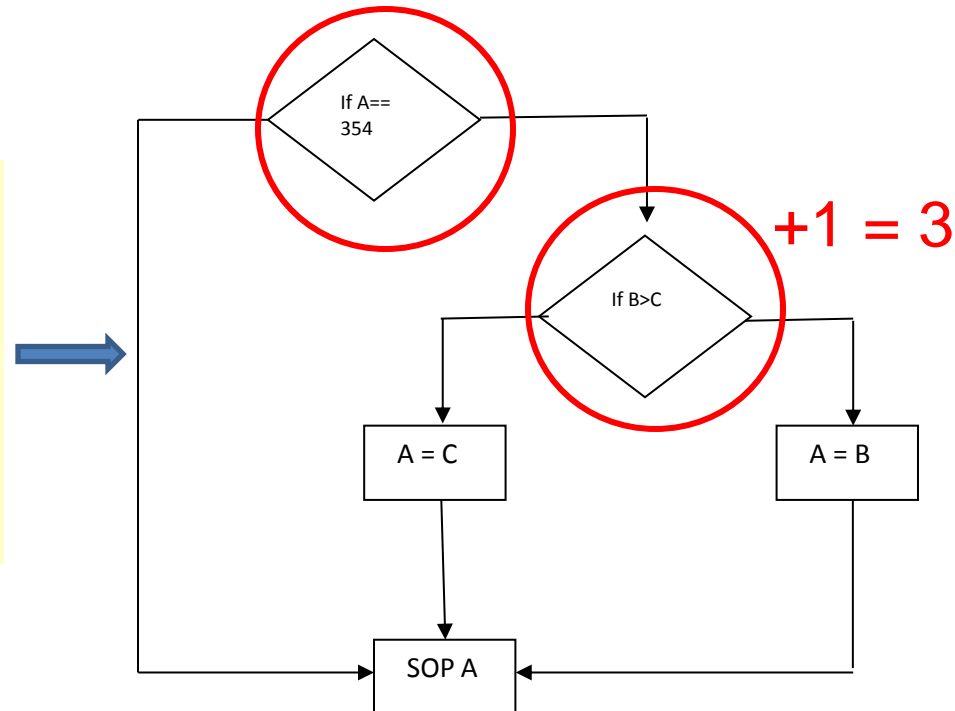

  - Cyclomatic Complexity

  - Lines of Code
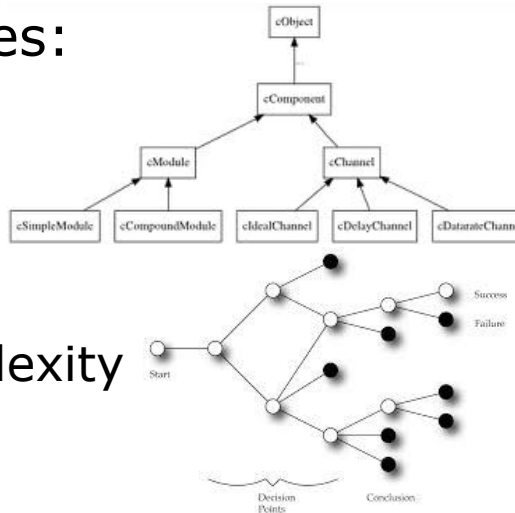
Cyclomatic Complexity (CC) Metrics

- The number of decisions in a program
- A simple way to calculate CC:
  - sum the number of decision points and add 1 to it.

- Example (Black Fig. 3.2)

```
if (A == 354)
{
    if (B > C)
        A = B;
    else A = C;
}
System.out.println(A);
```


+1 = 3

Cyclomatisk Complexity again

- CC comes in different variations (CC, C2, C3):

Metric	Name	Boolean operators	Select Case	Alternative name
CC	Cyclomatic complexity	Not counted	+1 for each Case branch	Regular cyclomatic complexity
CC2	Cyclomatic complexity with Booleans	+1 for each Boolean	+1 for each Case branch	Extended or strict cyclomatic complexity
CC3	Cyclomatic complexity without Cases	Not counted	+1 for an entire Select Case	Modified cyclomatic complexity

Code Standards Checking Tools

Eclipse:
- CheckStyle http://checkstyle.sourceforge.net/index.html
    Install plugin via Eclipse marketplace

NetBeans
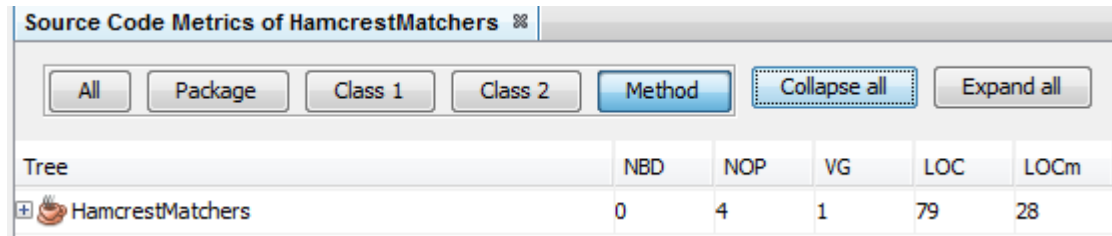- Integrated into tool. See: Choose Tools – Options – Editor – Hints
- Firebugs https://netbeans.org/kb/docs/java/code-inspect.html

Alternative http://kenai.com/projects/sqe/pages/Home
    See web page for how to install tool - it is an integrated package of
    Software Quality tools

Metrics Tools

- Eclipse:
  - Metrics tool
  - Google tool: https://developers.google.com/java-dev-tools/download-codepro (works with many versions of Eclipse, I think ☺)

- Netbeans
  - Version 7.0-7.4: http://plugins.netbeans.org/plugin/42970/sourcecodemetrics
  - Version 7.4: https://blogs.oracle.com/geertjan/entry/just_how_messed_up_is

Download 📄 1337018664_SourceCodeMetrics.nbm Tools → Plugins → downloaded → Add plugins…

Right click project → Source Code Metrics (Choose Window – view, if it doesn't show automatically)

Source Code Metrics of HamcrestMatchers ✖					
All	Package	Class 1	Class 2	Method	Collapse all	Expand all

Tree	NBD	NOP	VG	LOC	LOCm
⊞ ☕ HamcrestMatchers	0	4	1	79	28

Software measurements (Ex: Visual Studio) I

- **Maintainability Index** – Calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability. The calculation is based on the Halstead Volume, Cyclomatic Complexity and Lines of Code.

- **Cyclomatic Complexity** – Measures the structural complexity of the code. It is created by calculating the number of different code paths in the flow of the program such as if blocks, switch cases, and do, while, foreach and for loops then adding 1 to the total. A program that has complex control flow will require more unit tests to achieve good code coverage and will be less maintainable.

http://msdn.microsoft.com/en-us/library/bb385914.aspx

Software measurements (Ex: Visual Studio) II

- **Depth of Inheritance** – Indicates the number of class definitions that extend to the root of the class hierarchy. The deeper the hierarchy the more difficult it might be to understand where particular methods and fields are defined or/and redefined.

- **Class Coupling** – Measures the coupling to unique classes through parameters, local variables, return types, method calls,...

  Good software design dictates that types and methods should have high cohesion and low coupling. High coupling indicates a design that is difficult to reuse and maintain because of its many interdependencies on other types.

Software measurements (Ex: Visual Studio) III

- **Lines of Code** – Indicates the approximate number of lines in the code. The count is based on the IL code and is therefore not the exact number of lines in the source code file.

  A very high count might indicate that a type or method is trying to do too much work and should be split up. It might also indicate that the type or method might be difficult to maintain.

Screen Dump from VS2010 Code Metrics