Static Analysis
(with a little Dynamic Analysis thrown in)

Some of these notes adopted from Sommerville 9th ed.

Inquiry

A recent study[1] recommended teams code review 300-500 LOC per hour to achieve 70-90% defect discovery rates

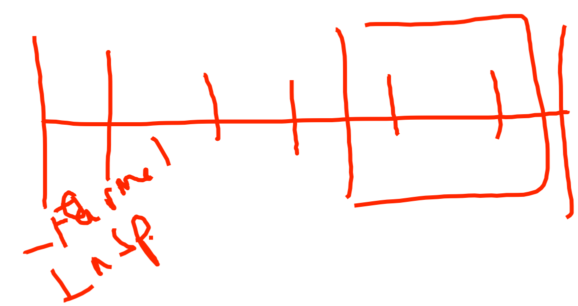
C.R. + SA.

Question: What is the upper and lower bound of an industry code review for fagan-style inspections?

Question: What would it take for your team to review *all of your code*?

WHERE WHAT

HOW LONG QP

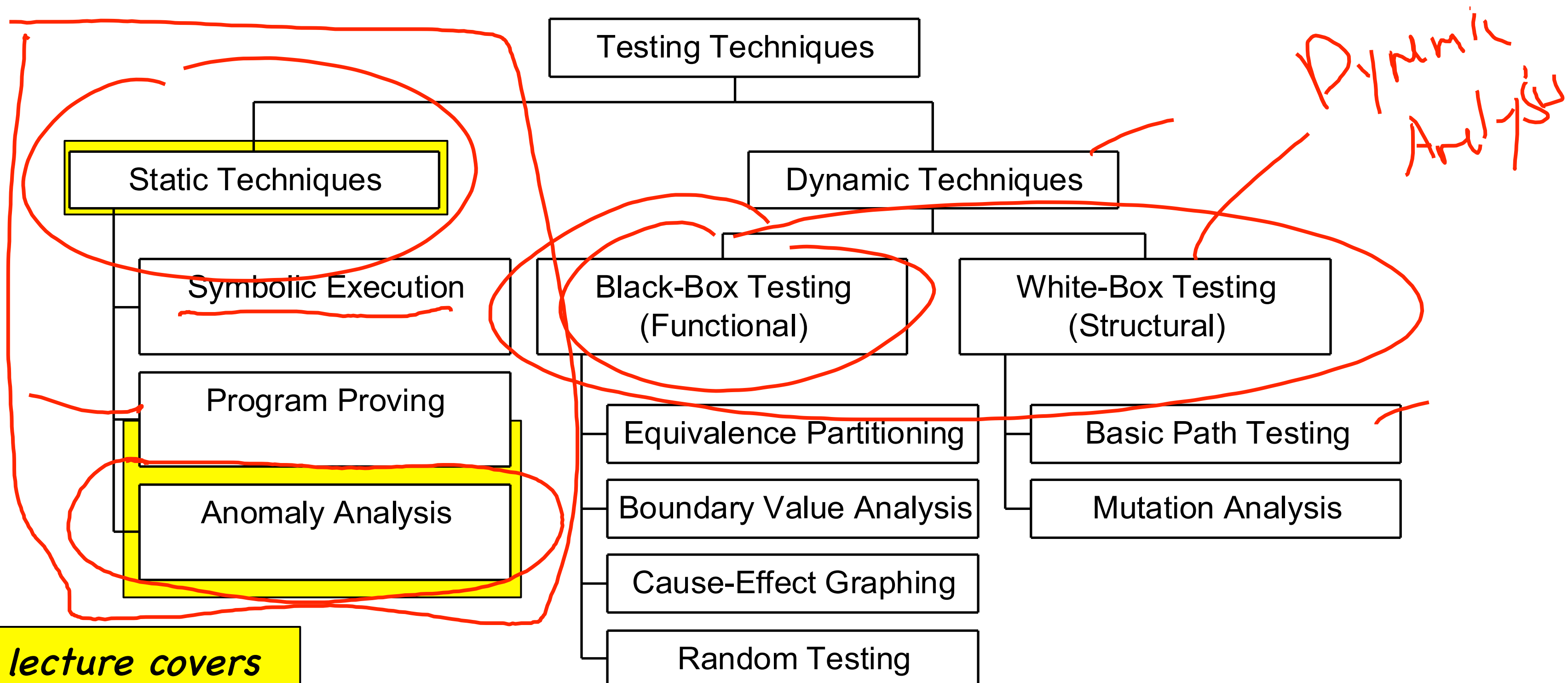


1. http://www.ibm.com/developerworks/rational/library/11-proven-practices-for-peer-review/

Unit Testing Techniques

Unit Testing checks that an individual program unit (subprogram, object class, package, module) behaves correctly.

- Static Testing - testing a unit without executing the unit code
- Dynamic Testing - testing a unit by executing a program unit using test data



This lecture covers

Program testing can be used to show the presence of bugs, but never to show their absence [Dijkstra]

Static Analysis Motivation (part 1)

C.R.

Obviously a costly process! Solution: Static Analysis

- Static analysers are tools for source processing
- They parse the program text and try to discover potentially erroneous conditions

— Automating Code reviews

- Very effective as an aid to inspections. A supplement to but not a replacement for inspections
- Static analysis tools can discover program anomalies which may be an indication of faults in the code automatically
  - Can be integrated into a continuous integration process
- Static analysis tools are often rule-based and extensible
  - The rules constructed are language-specific, as are the parsing routines

. – Automated

Static Analysis

security

Historical View Static Analysis Stages

- *Control flow analysis:* Checks for loops with multiple exit or entry points, finds unreachable code, etc.
- *Data use analysis:* Detects uninitialized variables, variables written twice without an intervening assignment, variables declared but never used, etc.
- *Interface analysis:* Checks the consistency of routine and procedure declarations and their use (i1, i2, obj)
- *Information flow analysis:* Identifies the dependencies of output variables. Does not detect anomalies itself but highlights information for code inspection or review
- *Path analysis:* Identifies paths through the program and sets out the statements executed in that path. Again, potentially useful in the review process

Levels of static analysis

Characteristic error checking

- The static analyzer can check for patterns in the code that are characteristic of errors made by programmers using a particular language.

User-defined error checking

- Users of a programming language define error patterns, thus extending the types of error that can be detected. This allows specific rules that apply to a program to be checked.

Assertion checking

- Developers include formal assertions in their program and relationships that must hold. The static analyzer symbolically executes code and highlights potential problems

Static Analysis

Modern static analyzers - what do they look at?

Example: PMD (http://pmd.sf.net)

- *Possible bugs* - Empty try/catch/finally/switch blocks.
- *Dead code* - Unused local variables, parameters and private methods
- *Empty* if/while statements
- *Overcomplicated expressions* - Unnecessary if statements, for loops that could be while loops
- *Suboptimal code* - wasteful String/StringBuffer usage
- *Duplicate code* - Copied/pasted code can mean copied/pasted bugs, and decreases maintainability.
- *Security* – particularly in dynamic languages
- *Style Checking* – considered static analysis but really its own thing.

PMD Sample output

25.3%

.000x KLOC

Element	# Violations	# Violations/LOC	# Violations/M	Project
banking.primitive.core	96	292.7 / 1000	1.92	CST316 lab1Lab2
Checking.java	12	279.1 / 1000	1.71	CST316 lab1Lab2
IfStmtsMustUseBraces	1	23.3 / 1000	0.14	CST316 lab1Lab2
CollapsibleIfStatements	1	23.3 / 1000	0.14	CST316 lab1Lab2
AvoidDeeplyNestedIfStmts	2	46.5 / 1000	0.29	CST316 lab1Lab2
MethodArgumentCouldBeFinal	(max) 5	116.3 / 1000	0.71	CST316 lab1Lab2
BeanMembersShouldSerialize	1	23.3 / 1000	0.14	CST316 lab1Lab2
OnlyOneReturn	2	46.5 / 1000	0.29	CST316 lab1Lab2
Savings.java	8	222.2 / 1000	1.33	CST316 lab1Lab2
IfStmtsMustUseBraces	1	27.8 / 1000	0.17	CST316 lab1Lab2
MethodArgumentCouldBeFinal	(max) 5	138.9 / 1000	0.83	CST316 lab1Lab2
BeanMembersShouldSerialize	1	27.8 / 1000	0.17	CST316 lab1Lab2
OnlyOneReturn	1	27.8 / 1000	0.17	CST316 lab1Lab2
AccountServer.java	5	714.3 / 1000	0.83	CST316 lab1Lab2
UnusedModifier	(max) 5	714.3 / 1000	0.83	CST316 lab1Lab2
Account.java	11	314.3 / 1000	1.10	CST316 lab1Lab2
ShortVariable	4	114.3 / 1000	0.40	CST316 lab1Lab2
MethodArgumentCouldBeFinal	4	114.3 / 1000	0.40	CST316 lab1Lab2
AbstractNaming	1	28.6 / 1000	0.10	CST316 lab1Lab2
BeanMembersShouldSerialize	2	57.1 / 1000	0.20	CST316 lab1Lab2
ServerSolution.java	30	283.0 / 1000	3.75	CST316 lab1Lab2
IfStmtsMustUseBraces	3	28.3 / 1000	0.38	CST316 lab1Lab2
SystemPrintln	2	18.9 / 1000	0.25	CST316 lab1Lab2
DefaultPackage	2	18.9 / 1000	0.25	CST316 lab1Lab2
ShortVariable	1	9.4 / 1000	0.12	CST316 lab1Lab2
MethodArgumentCouldBeFinal	(max) 5	47.2 / 1000	0.62	CST316 lab1Lab2
AvoidPrintStackTrace	4	37.7 / 1000	0.50	CST316 lab1Lab2
PreserveStackTrace	1	9.4 / 1000	0.12	CST316 lab1Lab2
BeanMembersShouldSerialize	1	9.4 / 1000	0.12	CST316 lab1Lab2
DataflowAnomalyAnalysis	1	9.4 / 1000	0.12	CST316 lab1Lab2
LocalVariableCouldBeFinal	(max) 5	47.2 / 1000	0.62	CST316 lab1Lab2
OnlyOneReturn	3	28.3 / 1000	0.38	CST316 lab1Lab2
AvoidCatchingThrowable	2	18.9 / 1000	0.25	CST316 lab1Lab2
AccountServer2Test.java	10	384.6 / 1000	3.33	CST316 lab1Lab2
AccountServerFactory.java	2	153.8 / 1000	0.67	CST316 lab1Lab2
UncommentedEmptyConstructor	1	76.9 / 1000	0.33	CST316 lab1Lab2
NonThreadSafeSingleton	1	76.9 / 1000	0.33	CST316 lab1Lab2
AccountServerTest.java	18	290.3 / 1000	2.57	CST316 lab1Lab2

Static Analysis Tools

1. *PMD* (see previous slide)
2. *Checkstyle* (checkclipse) - principally enforces code style guidelines
3. *FindBugs* - finds potential defects by inspecting compiled Java bytecode (hybrid static/dynamic)
4. *Jdepend/Classcyle* - Package dependencies
5. *Ruby/Javascript* – codeclimate.com, check ruby toolbox
6. *Javascript* – JSLint, JSHint
7. *Python* - Pylint
8. *PHP*: see http://mark-story.com/posts/view/static-analysis-tools-for-php
9. *Grails*: CodeNarc (excellent name!) http://grails.org/plugin/codenarc
10. *C*: Coverity the market leader, others: http://spinroot.com/static/

The output of sourcecode analyzers is dependent on what language you are using

Static Analysis Motivation (part 2)

Security

One perspective on static analysis is that it fills the gap in weak compiler messages

C

What does this mean for dynamic languages?

- Static analyzers look for security holes, issues with missing resources at run-time, and unbounded objects, functions, and/or variables.

  LISP ( )

- They typically do a type of *flow analysis*, which is kind of like our coverage graphs from unit testing

  W.B

  - Try to figure out what gets executed in what order
  - In a sense heuristically executed the program, attempting to discern what the call stack will look like at each point
  - This is computationally expensive
  - Often exposes lack of semantic completeness of the language (Javascript)

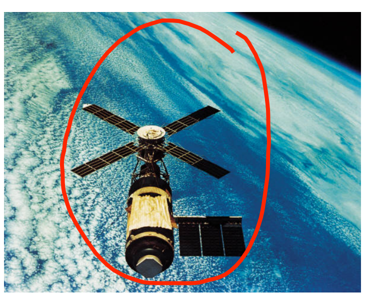
Static Analysis Motivation (part 3)

Verification & validation for critical systems involves additional validation processes and analysis than non-critical systems:

- *Because of the additional activities involved, validation costs for critical systems are usually significantly higher than for non-critical systems*
  - Normally, V & V costs take up > 50% of the total system development costs.
  - The costs and consequences of failure are high so it is cheaper to find and remove faults than to pay for system failure;

a auditor

- You may have to make a formal case to customers or to a regulator that the system meets its dependability requirements. A dependability case may require specific V & V activities to be carried out.
  - The outcome of the validation process is a tangible body of evidence that demonstrates the level of dependability of a software system.



NOTE: We don't do much formal methods in your degree program or on your projects, but this is an aside that is a very important perspective for engineers

From Sommerville 9th ed: Chapter 15 Dependability and Security Assurance

Verification and formal methods

Formal methods are the ultimate static verification technique

meth.

- A formal specification may be developed and mathematically analyzed for consistency, helping discover specification errors.

  Prover + Solver

- Formal arguments that a program conforms to its mathematical specification may be developed. This is effective in discovering programming and design errors.

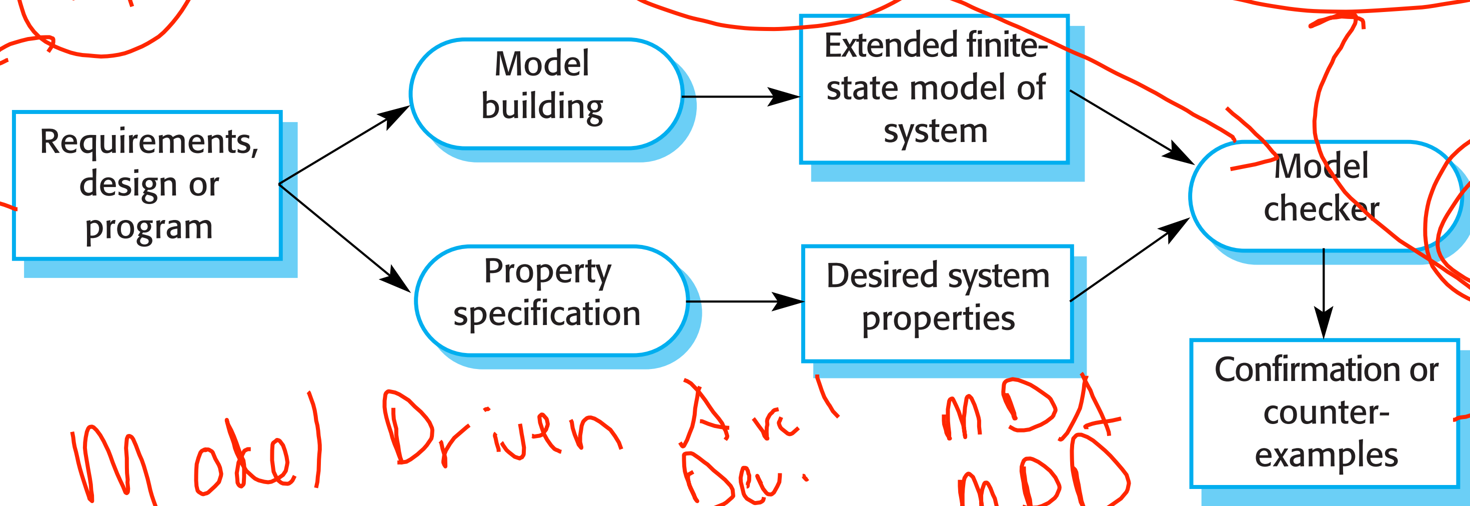
Producing a mathematical spec requires a detailed analysis of the requirements and this is likely to uncover errors.

- Concurrent systems can be analyzed for race conditions that might lead to deadlock. Testing for such problems is very difficult.
- Require notations that cannot be understood by domain experts.
- Very expensive to develop a specification and even more expensive to show that a program meets that specification.
- Proofs may contain errors.
- It may be possible to reach the same level of confidence in a program more cheaply using other V & V techniques.

Model checking

- Involves creating a finite state model of a system & using a special system (a model checker), checking that model for errors.
- The model checker explores all possible paths through the model and checks that a user-specified property is valid for each path.
- Model checking is particularly valuable for verifying concurrent systems, which are hard to test.
- Model checking is computationally very expensive, but is now practical to use in the verification of small to medium sized critical systems.

M → Verify → Transformer



Code

Model Driven Arch / Dev.

MDA

MDD

Examples

Z (ISO/IEC 13568/2002) *http://spivey.oriel.ox.ac.uk/~mike/zrm/zrm.pdf*

[Name, Date]

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \rightarrow\!\!\!\!\!+\; DATE$$
$$known = \mathrm{dom}\; birthday$$

AddBirthday

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$
$$name? \notin known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

1st item says domain has names & dates

2nd says *known* is the set of names we have birthdays for, and *birthday* is a function returning a name given a date

3rd is a state change behavior where a new birthday is added

HW RT

SPIN

2 doors (UPPAAL, Larsen, Larsson, Petterson & Skou 2003)

```
A room has 2 doors which can't be open at the same time. A door starts to open if its button is
pushed. The door opens for 6 seconds, then it stays open for at least 4 seconds but no more
than 8 seconds. The door takes 6 seconds to close and stays closed for at least 5 seconds.
/*Mutex: The two doors are never open at the same time.*/
A[] not (Door1.open and Door2.open)
/*Bounded Liveness: A door will open within 31 seconds.*/
A[] (Door1.opening imply User1.w<=31) and \
    (Door2.opening imply User2.w<=31)
/* Doors 1 and 2 can open. */
E<> Door1.open
E<> Door2.open
/* Liveness: Whenever a button is pushed, the corresponding door will eventually open.*/
Door1.wait --> Door1.open
Door2.wait --> Door2.open
/* The system is deadlock-free. */
A[] not deadlock
```

Specify rules constraints declarative

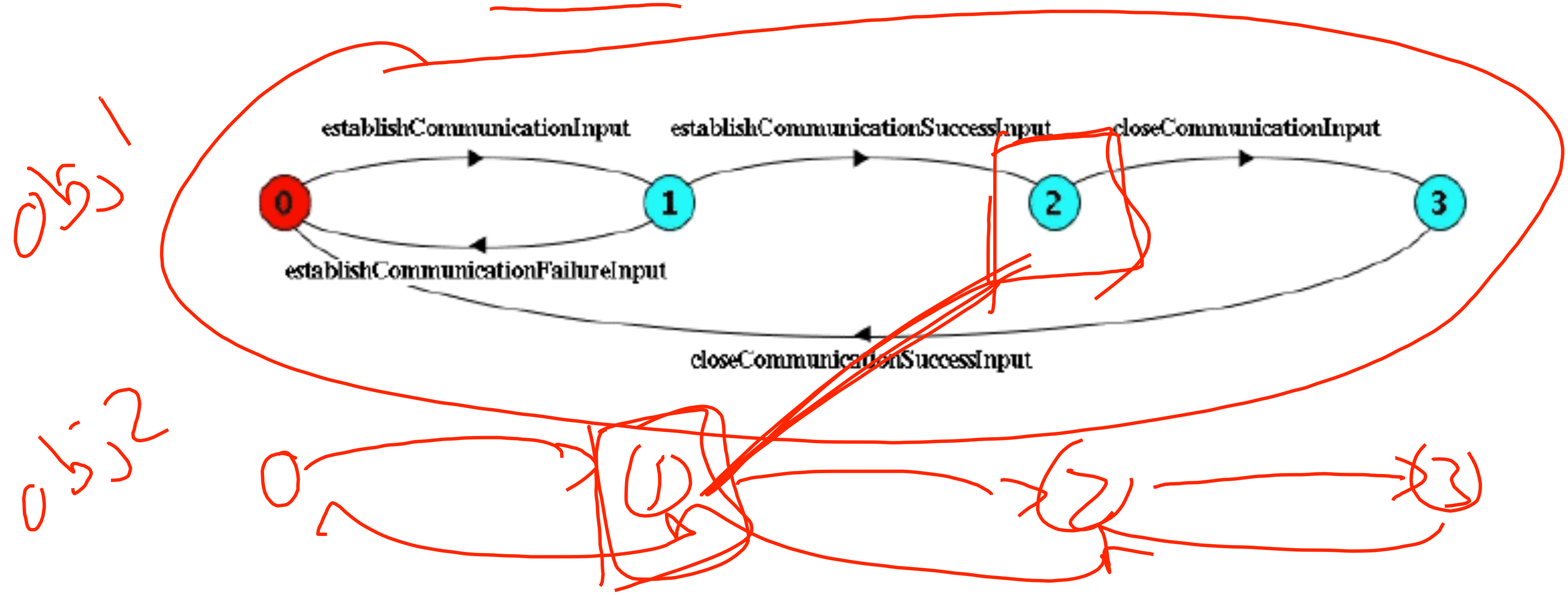
LTSA Example

(obj2 && obj2@) → valid?

Labeled Transition System Analyser (LTSA)

-Magee and Kramer (2006)

-Typical tool exploring visualization and rule checking of concurrent programs



Dynamic Analysis Goals

1. Does run-time behavior match intended behavior?
   - Reliability*: "The ability of a system or component to perform its required functions under stated conditions for a specified period of time." [IEEE 610.12-1990]
     - Do you observe the behavior that you expect to observe?
   - Robustness: The ability to degrade gracefully in the face of adverse scenarios/conditions
     - Often, the converse is not asked: "*Does the system not behave in a manner that is unintended?*"
     - This is where negative testing comes in
   - Therefore this is cast as a *testing* activity

Question: *Who* does this testing and *when?*

- QA/Test group is responsible for functional test
  - But how early do they get involved in the process?
  - How much functional testing is the development group responsible for itself before involving QA/Test?

*See Linda Rosenberg, Ted Hammer, Jack Shaw "Software Metrics and Reliability", 9th International Symposium - Germany, Nov 1998

Dynamic Analysis Goals

2. Are non-functional requirements met?

- Examples of non-functional requirements:
  - Reliability/Robustness (see previous slide): Actually, there are metrics for reliability, but not so much for robustness
  - Performance/Scalability: These are often seen as the same, but in fact are not.
  - Security/Safety: More and more important nowadays, particularly in light of recent security and privacy failures

Performance and Scalability

- *Performance* is typically externally measurable, and therefore included in the testing process
  - Ex: “The web site must respond to the user within 7 seconds”
- *Scalability* can be “somewhat” expressed in measurable terms
  - Ex: “The web site must handle a peak load of 3,000 sessions”
  - The issue again is *who* does *what* *when*?
    - If you (developer) wait until the system goes to test to find out, *it is too late*!
    - It can be very difficult to recover when a system does not exhibit good run-time properties here - need more analytical activities during development
      - » Profiling, Benchmarking, Tuning, Design scenario review

(more…)

Dynamic Analysis Goals

3. Do the internal elements in the run-time environment interact and behave in a manner consistent with the design goals of the system?
   - "White-box" in the sense that your design intends to create objects and interaction patterns to satisfy requirements - and does it in fact do so?
   - This is solely a developer activity
     - Debugging is important here
       - Run-time trace through logging or debugger
       - Use of assertions
     - Profiling is also important
       - Can tell you your object structure
       - Can drill-down into possible bottlenecks when you observe aggregate negative properties of the run-time environment (e.g. slowness or other "ungraceful" degradation)
       - Can also pinpoint resource leak and contention issues

Dev. expectation

profiler

2 dmB

20k

Dynamic Analysis Activities

Activities and Tools for the developer

Debug

Compile

Release

1. A good debugger (see Eclipse debugger)
2. Good instrumentation

   Log4j

   - Read: “logging”. But you need to introduce logging in a way that is unobtrusive, maintainable, and doesn’t degrade performance.
3. Profiler - to observe object graphs at run-time
   - Useful for finding memory access violations or resource leaks
     - Examples: Jbuilder, Valgrind, Purify

   profile
4. Run-time environment monitor
   - Observe run-time process aggregate statistics
   - Both per language platform and per system tools

     Unix sys
     - Examples: jconsole, hprof and thread dump, but also vmstat, top, etc.

     Activity monitor
5. Rapid deployment process
   - You need to do this over and over in a dev env

   Dev Test / Staging

Static and Dynamic Analysis

Using software tools to “critically and carefully examine” the properties (and potential issues) in a software system

Static analysis – *analysis of the source code*

- Became popular in weakly-typed languages (C) as many errors were undetected by the compiler
- Strongly-typed languages detect more errors during compilation
- *Modern Perspective*:

  consistent

  - Agile processes use static analysis as a *better* and more *productive replacement* for costly code reviews.

Dynamic analysis – *Analyzing the run-time behaviors of an executable element (object, component, [sub]system …)*

1. Does run-time behavior match intended behavior?
2. Are non-functional requirements met?
3. Do the internal elements in the run-time environment interact and behave in a manner consistent with the design goals of the system?
   – In a sense, “white-box” analysis of the executing code

Analysis Guidelines

1. Do it throughout the lifecycle!
   - "Analysis" implies you are observing, measuring, and *judging*; so you should be doing this all the time!
2. Know what you are looking for
   - Have a goal in mind; *(a Sprint goal?)* — Quality Pol
     - Static analysis tools can find a lot of things, need to configure against a goal
3. Benchmark often, and track your results!
   - Do not wait until the app is done to create a result
   - Create iteration-level benchmarks and track over time

4. Automate Automate Automate
   - *Static analysis* can be viewed as a substitute (or augmentation of) time-consuming code reviews
5. Use in safety-critical or secure situations
   - Mission critical domains require a level of <u>assurance</u>