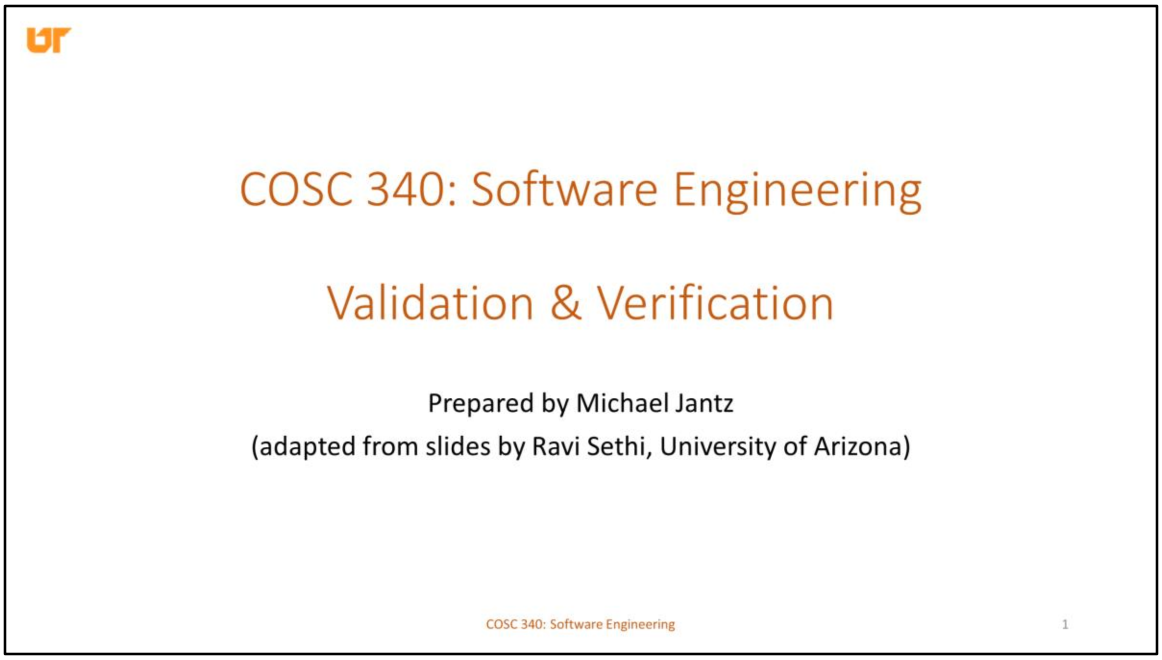
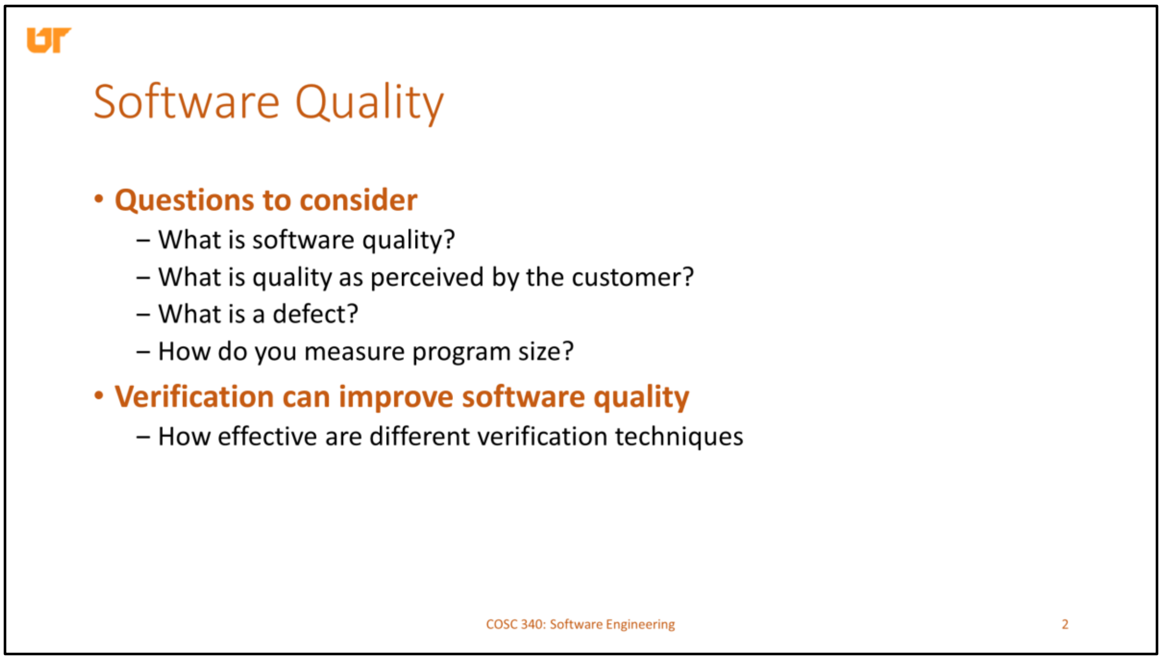


COSC 340: Software Engineering

Validation & Verification

Prepared by Michael Jantz
(adapted from slides by Ravi Sethi, University of Arizona)



Software Quality

- **Questions to consider**
  - What is software quality?
  - What is quality as perceived by the customer?
  - What is a defect?
  - How do you measure program size?
- **Verification can improve software quality**
  - How effective are different verification techniques

COSC 340: Software Engineering 2

The main purpose of validation and verification is to improve software quality.

In this lecture, we'll consider several questions related to software quality including:

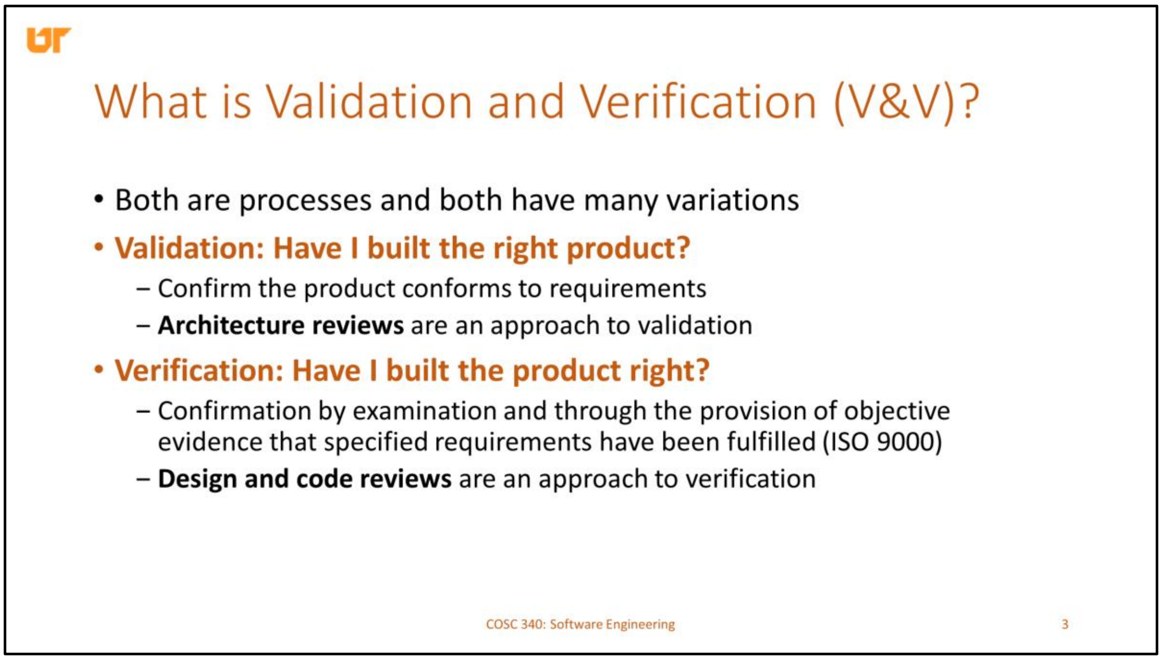
What is software quality? What sorts of features are important for high-quality software?

Software quality might be different depending on your point of view. What makes software high quality as perceived by the customer? Or as perceived by the developer?

What is a software defect? Programs with more defects are often perceived to have lower quality.

Related to program quality is program size. How do you measure program size? Larger programs require more effort and might be harder to improve.

We'll also discuss different verification techniques to improve program quality and examine their effectiveness.


What is Validation and Verification (V&V)?

- Both are processes and both have many variations
- **Validation: Have I built the right product?**
  - Confirm the product conforms to requirements
  - **Architecture reviews** are an approach to validation
- **Verification: Have I built the product right?**
  - Confirmation by examination and through the provision of objective evidence that specified requirements have been fulfilled (ISO 9000)
  - **Design and code reviews** are an approach to verification

COSC 340: Software Engineering 3

Throughout the next few lectures, we'll also be discussing validation and verification.

V&V are often thought to be the same thing. They are both processes that developers can use to improve program quality.

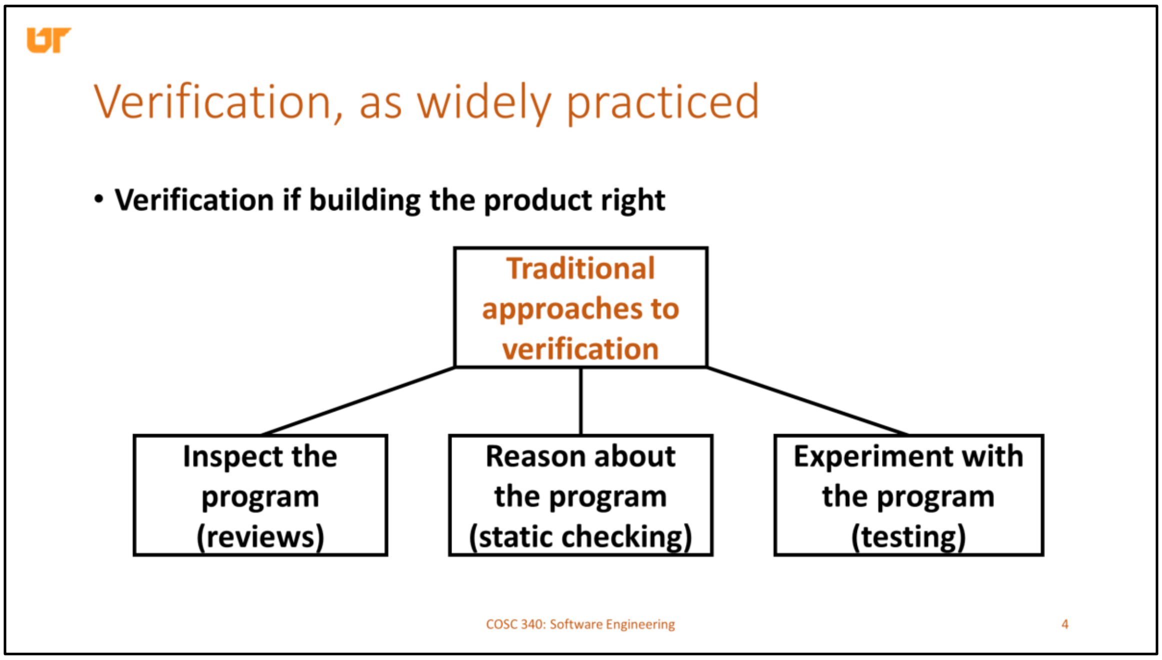
However, these processes address different questions and concerns.

(credit to Boehm who also did the risk reduction framework for this construction)
Validation aims to answer the question: Have I built the right product? So, validation processes try to confirm that the product you've built meets the customer's needs. Validation is typically performed at a higher level than verification, and includes activities such as requirements testing and architectural reviews.

In contrast, verification asks the question: Have built the product right? According to the ISO 9000 family of quality management standards, verification is to use examination and objective evidence to determine that the specified requirements have been fulfilled.

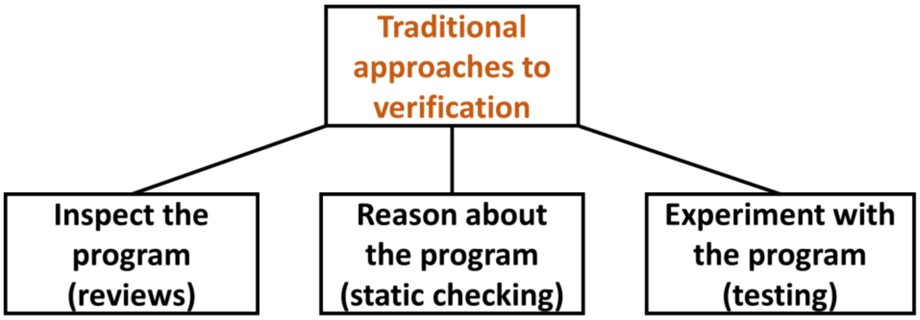
For verification, the requirements might be intermediate requirements, such as the requirements of a particular phase or iteration.

As an example, verification is typically associated with internal reviews, such as design and code reviews, with in-depth analysis of some aspect of the software.



Verification, as widely practiced

- Verification if building the product right


Traditional approaches to verification
Inspect the program (reviews)
Reason about the program (static checking)
Experiment with the program (testing)

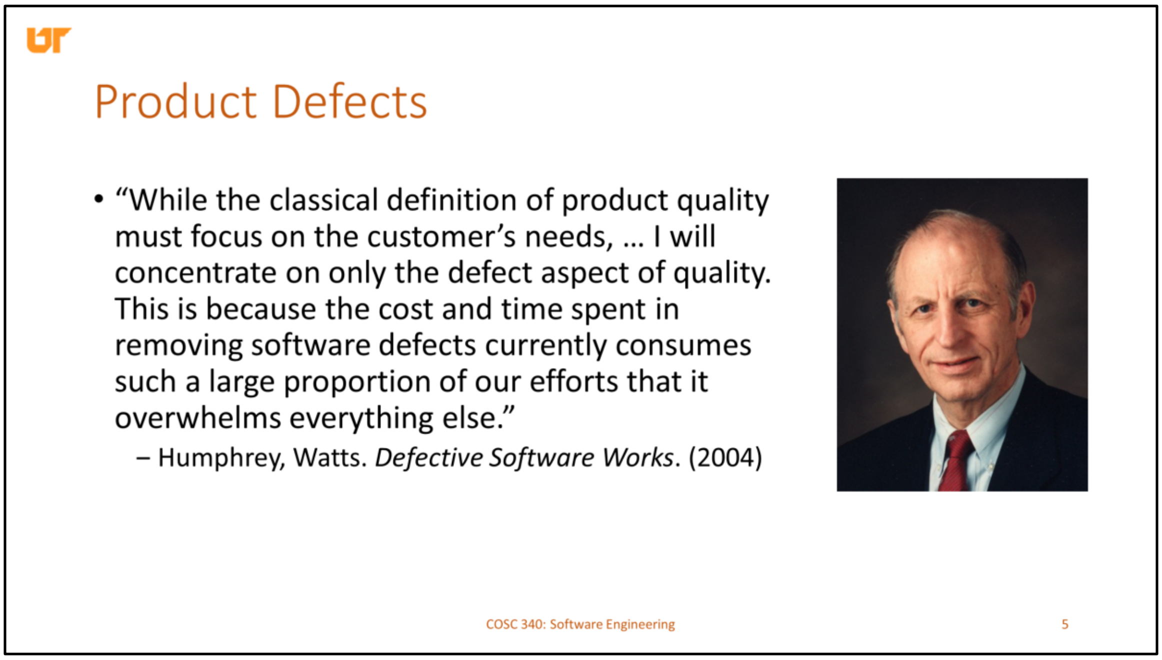
A number of different strategies are used, both formally and informally, to verify that a software product is well constructed.

Some techniques that are widely used in practice are shown on this slide.

Formal and informal reviews of program design and source code are used to inspect the software.

Static analysis tools, such as tools to examine the call graph, or tools for identifying common bugs not reported by the compiler, are another useful form of verification and can help you reason about the program and its structure.

And dynamic analysis tools such as Pin or valgrind, debuggers, and other types of runtime experimentation are another form of verification that help you test different properties of your program.



Product Defects

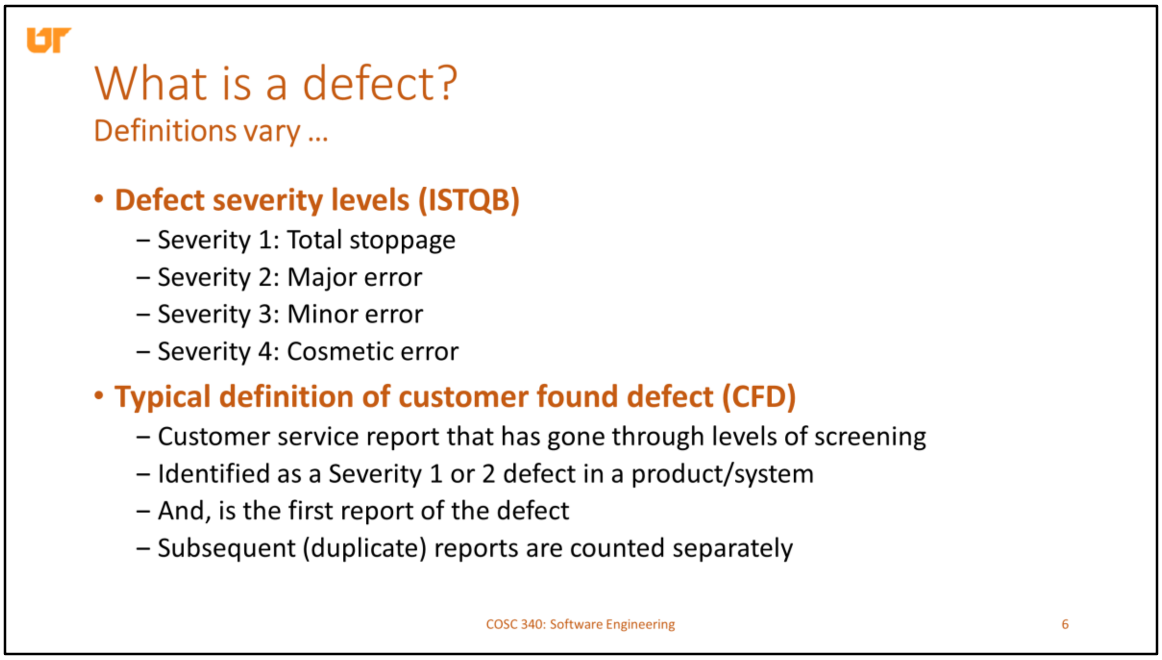
- "While the classical definition of product quality must focus on the customer's needs, … I will concentrate on only the defect aspect of quality. This is because the cost and time spent in removing software defects currently consumes such a large proportion of our efforts that it overwhelms everything else."
  - Humphrey, Watts. *Defective Software Works*. (2004)

COSC 340: Software Engineering 5

One of the most important aspects of software quality is the amount of errors or defects in the software.

In fact, Watts Humphrey, who was the leader of software quality at IBM, and who wrote many articles on software quality wrote in one of his articles:

"While the classical definition of product quality must focus on the customer's needs, … I will concentrate on only the defect aspect of quality. This is because the cost and time spent in removing software defects currently consumes such a large proportion of our efforts that it overwhelms everything else."



What is a defect?

Definitions vary …

- **Defect severity levels (ISTQB)**
  - Severity 1: Total stoppage
  - Severity 2: Major error
  - Severity 3: Minor error
  - Severity 4: Cosmetic error
- **Typical definition of customer found defect (CFD)**
  - Customer service report that has gone through levels of screening
  - Identified as a Severity 1 or 2 defect in a product/system
  - And, is the first report of the defect
  - Subsequent (duplicate) reports are counted separately

So, let's begin by asking, 'what is a defect'?

There can be many definitions – broadly can be thought of as some sort of error in your product.

According to the International Software Testing Qualifications Board (ISTQB), there are four levels of defect severity.

The first is a critical defect that results in total stoppage of usage. There is no ready workaround for a level 1 defect. Example is unsuccessful installation, complete failure of some feature.

The next level is a major defect – which affects some major functionality or major data. There is a workaround – but the workaround is not obvious and difficult. For instance, it may require using a different module or installing new software.

Next are minor defects which affect some minor functionality or non-critical data.

These typically have an easier workaround. For example, using the keyboard shortcut to save is broken – and so you have to use the menu.

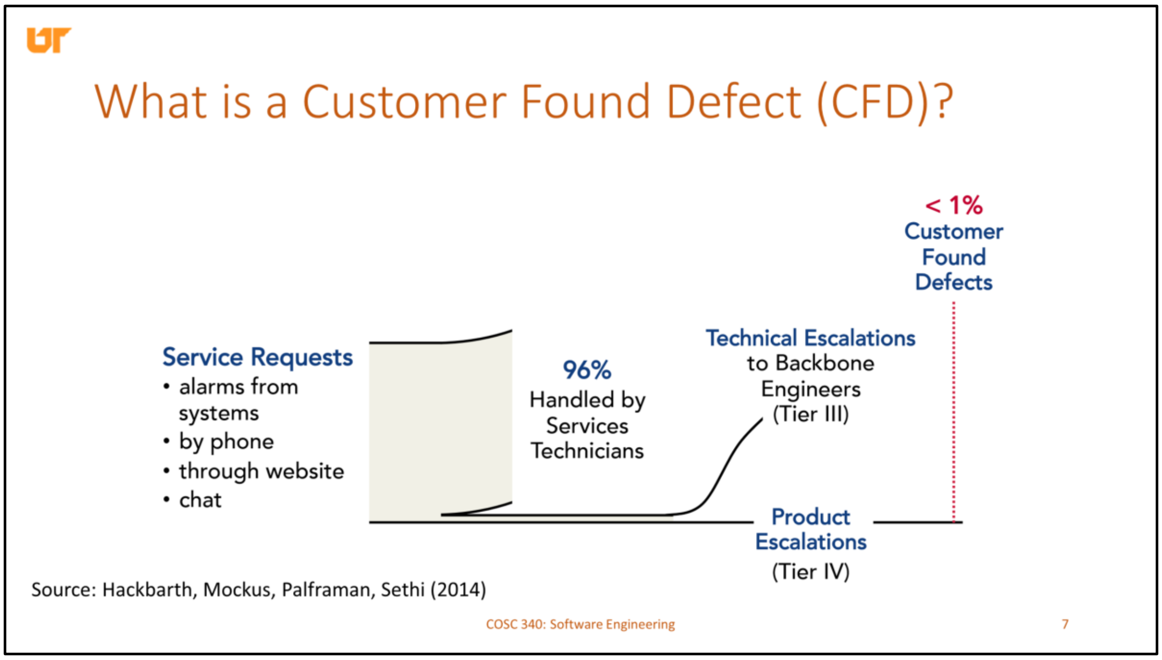
Lastly are cosmetic or trivial defects. These don't really affect functionality or data. They don't really need a workaround – they're just inconvenient. An example would be like inconsistent layouts or spelling mistakes.

Not all defects are equal. Most defects are found and fixed during development or testing, before a product is delivered.

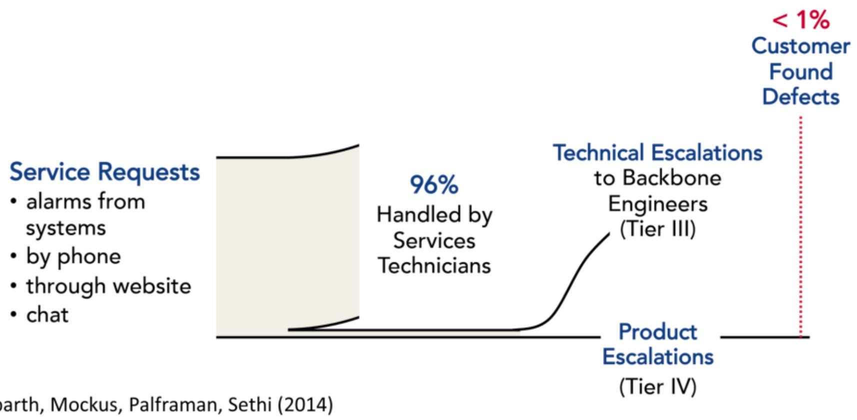
In evaluating software quality, it's useful to focus on those defects that are found by the customer. These are called customer found defects (CFD's for short).

A 2014 study at Avaya (part of AT&T) tried to quantify customer found defects in their system. They used this definition, which is typical for defining CFDs.

Look at only those defects that were found by customers in the field, and have been through several levels of screening, before being elevated to the development group and were severe enough to be classified by the development group as a software defect.



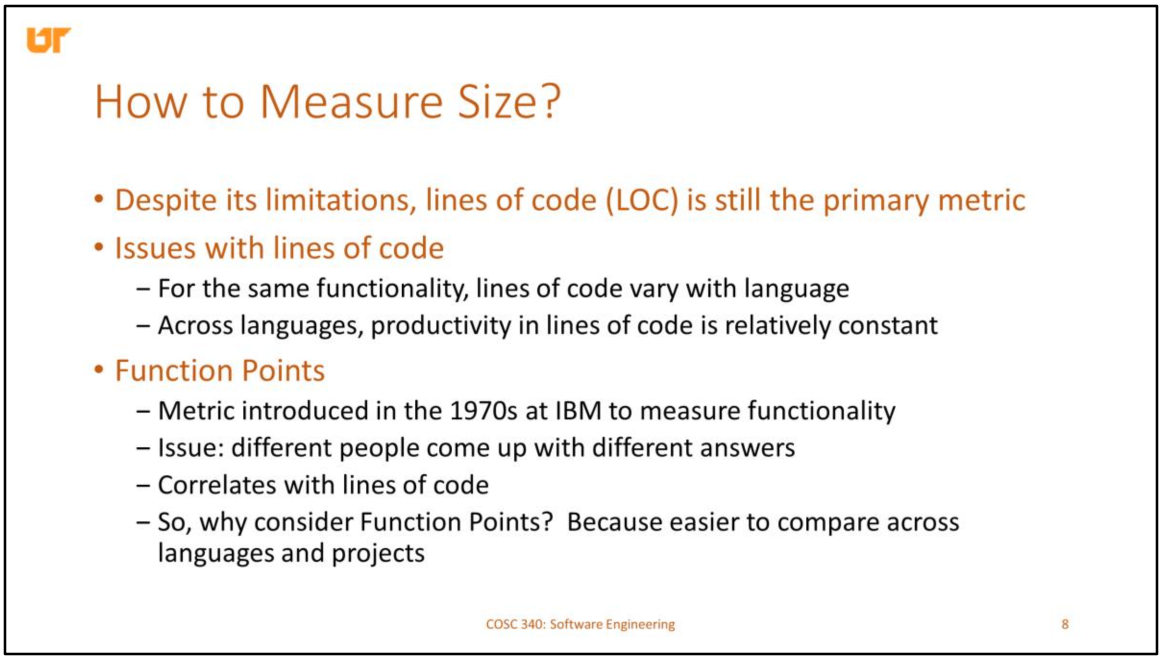
What is a Customer Found Defect (CFD)?



Source: Hackbarth, Mockus, Palframan, Sethi (2014)

This figure is taken from the Avaya study. They found that less than 1% of customer service requests materialized as CFD’s.

They looked at every service request raised by phone, through their website, or through chat. Almost everything is handled by service technicians. About 4% of requests are escalated to the product engineers , and only about 1% are classified as CFD’s by the software engineers.



How to Measure Size?

- Despite its limitations, lines of code (LOC) is still the primary metric
- Issues with lines of code
  - For the same functionality, lines of code vary with language
  - Across languages, productivity in lines of code is relatively constant
- Function Points
  - Metric introduced in the 1970s at IBM to measure functionality
  - Issue: different people come up with different answers
  - Correlates with lines of code
  - So, why consider Function Points? Because easier to compare across languages and projects

COSC 340: Software Engineering 8

Another important aspect to consider when evaluating quality is the size of the project you're working on. For instance, to measure the quality of your output, you might want to consider defects per unit size, rather than just looking at defects alone.

So, how can we measure program size?

Although it has some significant limitations, lines of code is still the primary metric for program size.

One major issue is that its very hard to compare projects using different languages in terms of lines of code. In most cases, a project written in C or COBOL is just going to use more lines of code than a project with similar functionality written in python or lisp or other HLL.

However, within a particular language, productivity in lines of code is relatively constant – but it can vary from person to person.

Other issues:

Using LOC as a proxy for productivity is a problem (only about 30% to 35% of effort is spent writing code)
Skilled developers can produce same functionality in fewer LOC
Lack of counting standards: count braces? Whitespace?

To address some of these issues, the function point metric was introduced at IBM to measure functionality.

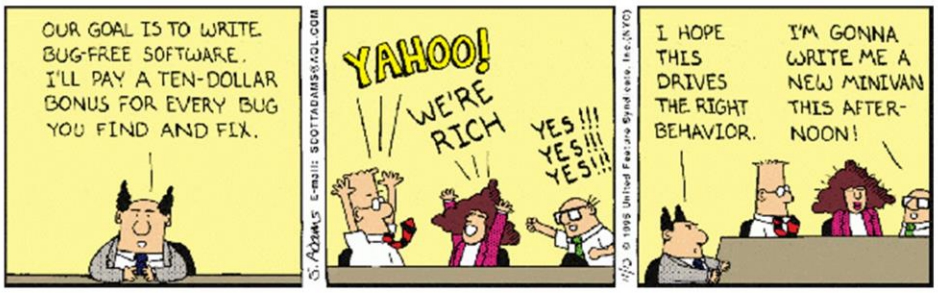
Basically, the development team classifies the functional user requirements for the project and assigns them a number of points based on their complexity.

The approach is standardized, but how many points a particular functionality is worth is still subjective.

People have found that function points correlates with lines of code – but people still use it because they make it much easier to compare productivity across languages and projects.

Issues with Metrics

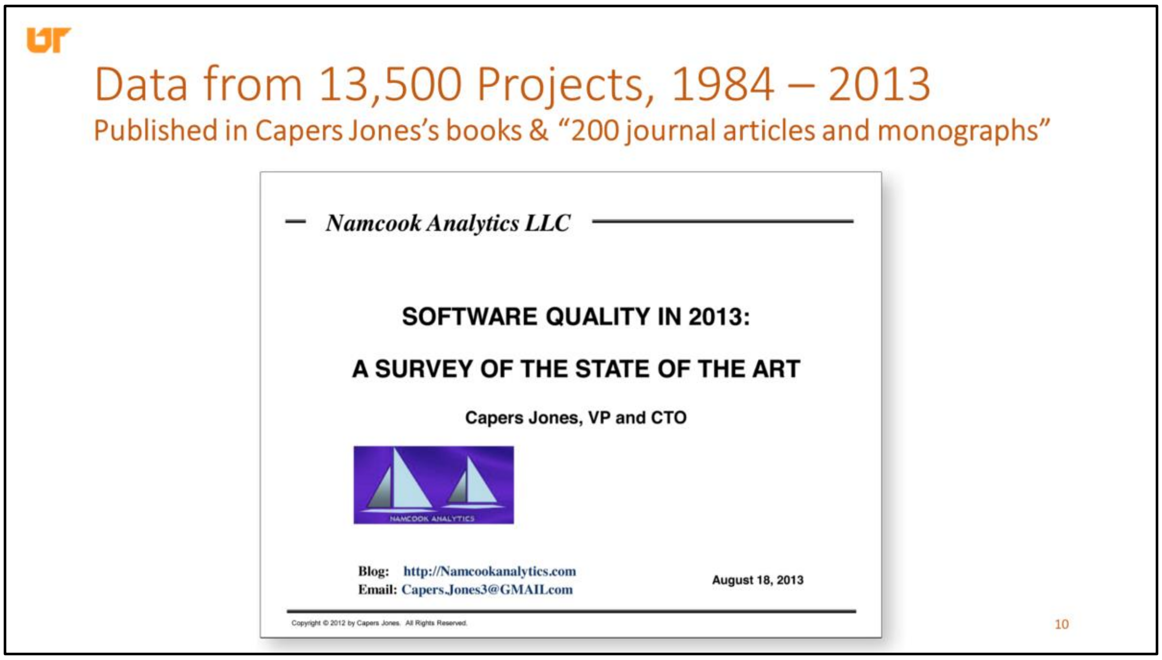
- Metrics can be subverted!


OUR GOAL IS TO WRITE BUG-FREE SOFTWARE. I'LL PAY A TEN-DOLLAR BONUS FOR EVERY BUG YOU FIND AND FIX.
S. Adams E-mail: SCOTTADAMS@AOL.COM
YAHOO!
WE'RE RICH
YES!!! YES!!! YES!!!
© 1995 United Feature Syndicate, Inc.(NYC)
I HOPE THIS DRIVES THE RIGHT BEHAVIOR.
I'M GONNA WRITE ME A NEW MINIVAN THIS AFTERNOON!

Now – another issue with metrics – as pointed out by this comic – is that certain metrics can be subverted.

For instance, if you're using lines of code to measure productivity, your developers might decide to write their code in a verbose style to make themselves appear more productive.

Or – for instance – as in this comic – developers might focus on the quickest bug fixes rather than the most important if you specifically measure their output in terms of the number of bugs they've fixed.



Data from 13,500 Projects, 1984 – 2013

Published in Capers Jones's books & "200 journal articles and monographs"

Namcook Analytics LLC

SOFTWARE QUALITY IN 2013:

A SURVEY OF THE STATE OF THE ART

Capers Jones, VP and CTO

NAMCOOK ANALYTICS

Blog: http://Namcookanalytics.com
Email: Capers.Jones3@GMAIL.com

August 18, 2013

Copyright © 2012 by Capers Jones. All Rights Reserved.

10

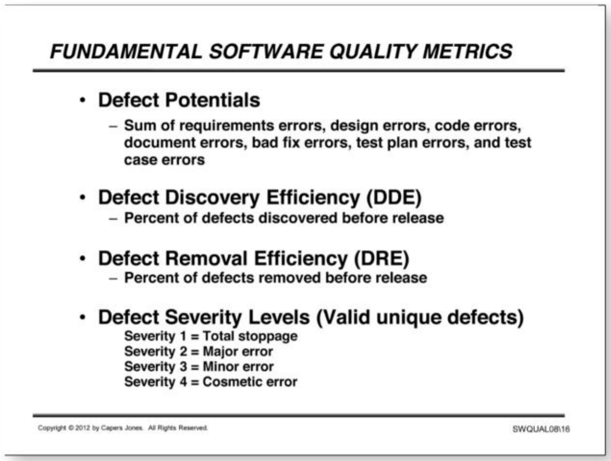
Next, I want to discuss large study on software quality by Caper Jones.

Capers Jones is the VP and CTO of Namcook Analytics, which is a company focused primarily on the study of software quality.

Jones collected data from more than 13,500 projects over almost 20 years.

Software Quality: State of the Art

Measures focus primarily on development



FUNDAMENTAL SOFTWARE QUALITY METRICS

- **Defect Potentials**
  - Sum of requirements errors, design errors, code errors, document errors, bad fix errors, test plan errors, and test case errors
- **Defect Discovery Efficiency (DDE)**
  - Percent of defects discovered before release
- **Defect Removal Efficiency (DRE)**
  - Percent of defects removed before release
- **Defect Severity Levels (Valid unique defects)**

  Severity 1 = Total stoppage
  Severity 2 = Major error
  Severity 3 = Minor error
  Severity 4 = Cosmetic error

SWQUAL08\16

The study considered a number of different metrics to assess software quality. These are listed on the next couple slides.

Defect potentials: total number of different types of errors in the software
Defect Discovery Efficiency: % of defects discovered before release
Defect Removal Efficiency: % of defects removed before release.

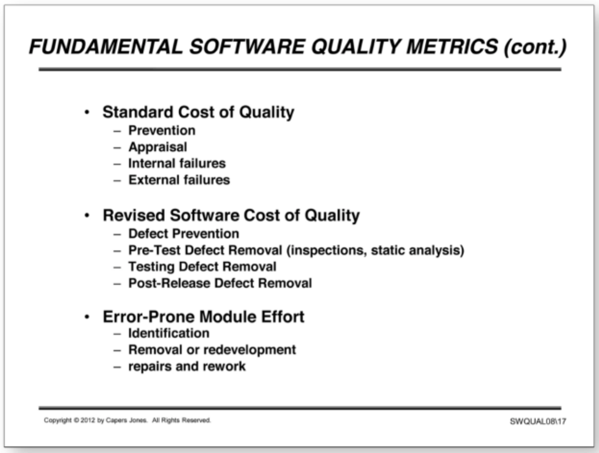
Also recorded defect severity levels of each defect (similar to the severity levels we discussed earlier).

Jones says that DRE is the ‘most powerful quality metric in the industry’.
Very simple to calculate: defects found / defects present (estimate based off published data).

Ideally, you should have over 90% of your bugs – but this is very hard to do without special training and static analysis tools.

Software Quality: State of the Art

Measures focus primarily on development



FUNDAMENTAL SOFTWARE QUALITY METRICS (cont.)

- **Standard Cost of Quality**
  - Prevention
  - Appraisal
  - Internal failures
  - External failures
- **Revised Software Cost of Quality**
  - Defect Prevention
  - Pre-Test Defect Removal (inspections, static analysis)
  - Testing Defect Removal
  - Post-Release Defect Removal
- **Error-Prone Module Effort**
  - Identification
  - Removal or redevelopment
  - repairs and rework

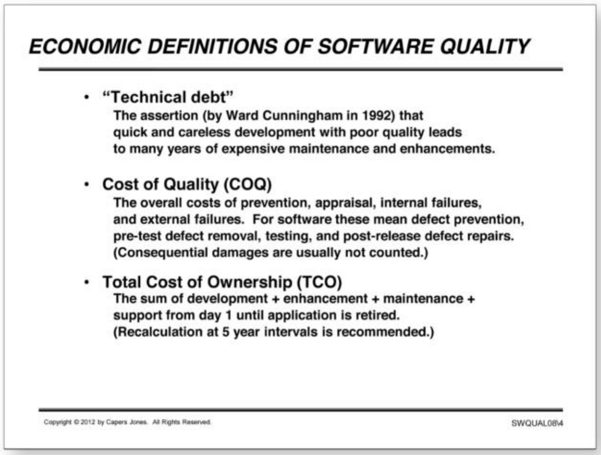
They also considered cost metrics for dealing with certain types of defects.

The standard cost of quality assessed the cost of preventing, finding, and dealing with failures due to defects.

The revised software cost of quality considers the cost of prevention along with the cost of removing the defect.

And they also looked at the error prone module effort. Studies have shown that most of the bugs in the system come from a very small number of modules or files in your source code. So, the cost associated with identifying and repairing these error prone modules is also recorded.

Terms you might encounter



ECONOMIC DEFINITIONS OF SOFTWARE QUALITY

- "Technical debt"
  The assertion (by Ward Cunningham in 1992) that quick and careless development with poor quality leads to many years of expensive maintenance and enhancements.
- Cost of Quality (COQ)
  The overall costs of prevention, appraisal, internal failures, and external failures. For software these mean defect prevention, pre-test defect removal, testing, and post-release defect repairs. (Consequential damages are usually not counted.)
- Total Cost of Ownership (TCO)
  The sum of development + enhancement + maintenance + support from day 1 until application is retired. (Recalculation at 5 year intervals is recommended.)

SWQUAL08\4

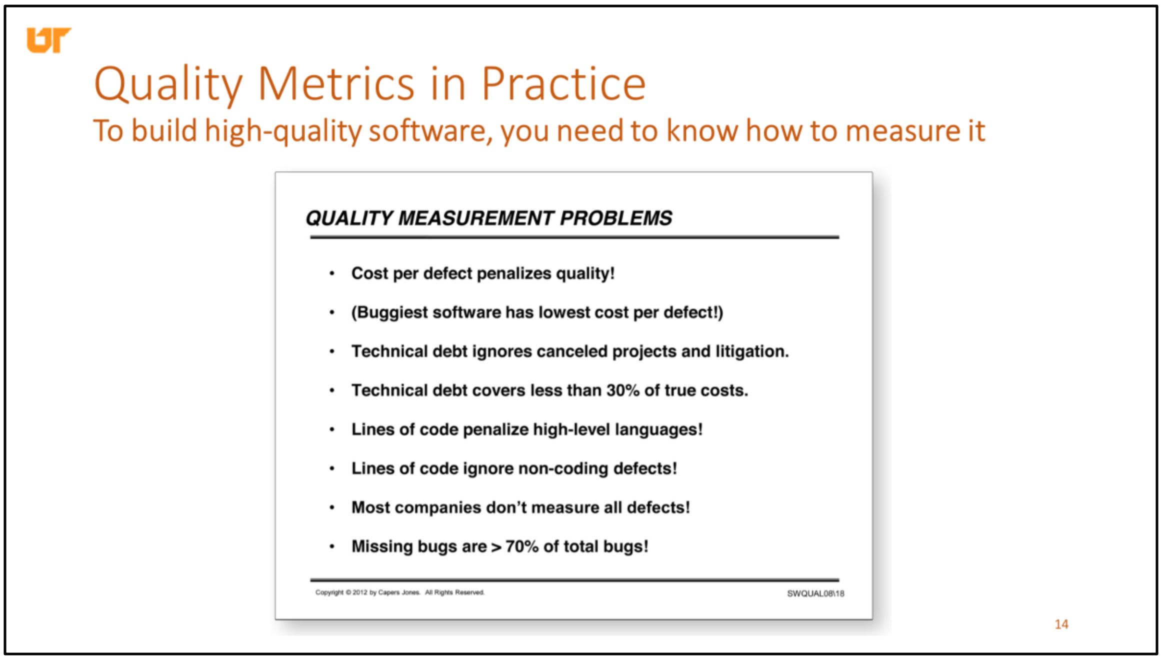
Jones also discusses some economic definitions of software quality.

I include this slide to point out some terms you will see in the next few slides.

The first is technical debt. This is the assertion that quick and careless development of poor quality software will often lead to many years of expensive maintenance and enhancements – which you could have otherwise avoided had you invested a little bit of money and effort into software quality.

Related to this technical debt idea are the "cost of quality" – which measures the overall cost of defect prevention and (pre- and post-release) defect repair.

Also related is the "total cost of ownership" a product – that is the sum cost of development, enhancement, and maintenance of the software since day 1.



Quality Metrics in Practice

To build high-quality software, you need to know how to measure it

QUALITY MEASUREMENT PROBLEMS

- Cost per defect penalizes quality!
- (Buggiest software has lowest cost per defect!)
- Technical debt ignores canceled projects and litigation.
- Technical debt covers less than 30% of true costs.
- Lines of code penalize high-level languages!
- Lines of code ignore non-coding defects!
- Most companies don't measure all defects!
- Missing bugs are > 70% of total bugs!

Copyright © 2012 by Capers Jones. All Rights Reserved. SWQUAL08\18

14

OK – so if you want to build high quality software – you need to know how to measure it.

Unfortunately, there are a number of issues with methods that have been used to measure software quality in practice.

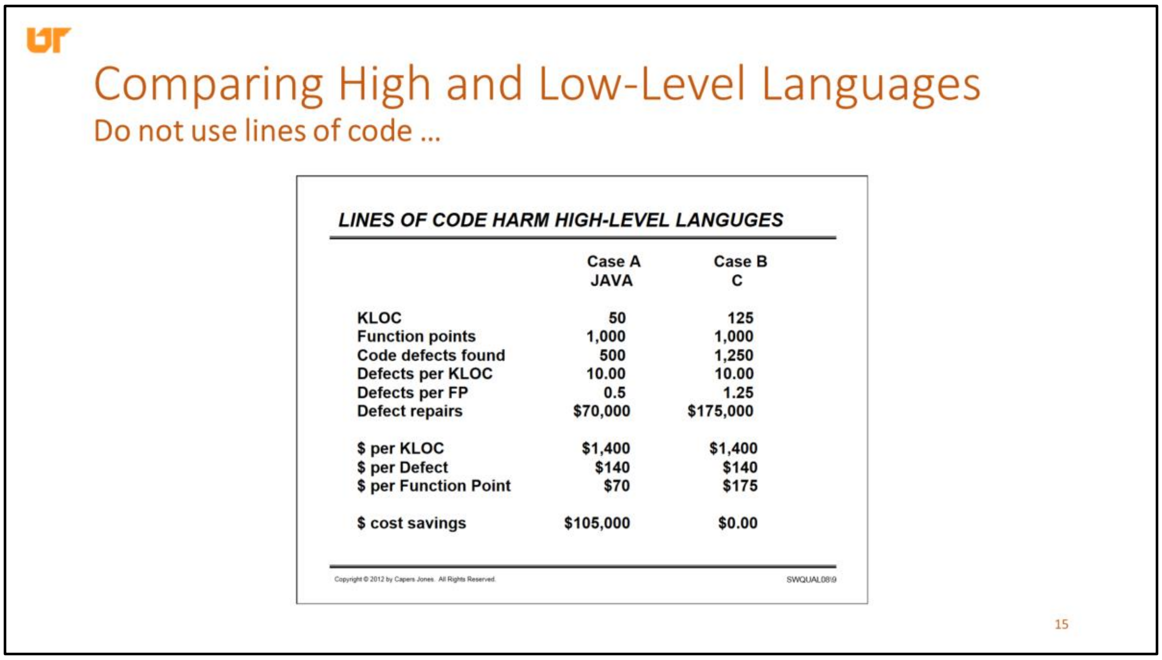
Cost per defect is the amount of time, money, resources you spend fixing each bug. Some would say having a low cost per defect is a good thing. But why might cost per defect not be the best measurement of code quality?

(Buggiest software has the lowest cost per defect because code with a lot of bugs will have many that are easy to fix). As you fix them, cost per defect will increase because the last bugs you fix will always be more subtle.

There are issues with technical debt because it ignores costs that are not part of a completed project – it covers less than 30% of total cost.

Lines of code can really vary from language to language – so it’s not good to use lines of code in quality comparisons across languages.

Also, Jones points out that using lines of code ignores non-coding defects. According to Jones, requirements and design defects outnumber coding defects – and most companies do not measure those defects.



Comparing High and Low-Level Languages

Do not use lines of code …

LINES OF CODE HARM HIGH-LEVEL LANGUGES

	Case A JAVA	Case B C
KLOC	50	125
Function points	1,000	1,000
Code defects found	500	1,250
Defects per KLOC	10.00	10.00
Defects per FP	0.5	1.25
Defect repairs	$70,000	$175,000
$ per KLOC	$1,400	$1,400
$ per Defect	$140	$140
$ per Function Point	$70	$175
$ cost savings	$105,000	$0.00

SWQUAL08\9

15

This example shows how using lines of code in a quality comparison can harm high-level languages.

Say, we have two applications, one written in Java and the other in C. The application in Java only has 50K LOC, while the C application has 125K LOC.

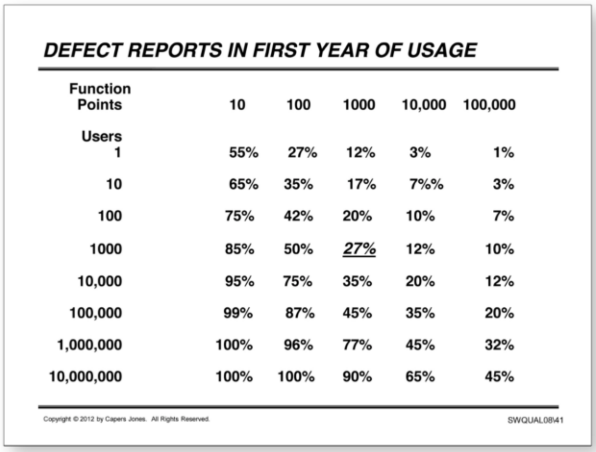
Both applications have the same number of functions points – so they do roughly the same amount of functional work. In the Java app, we found 500 defects that cost $70K to fix, and in the C app, we have 1250 defects that cost $175K to fix.

If we look at the cost to fix these defects per line of code – then the cost appears the same.

But if you look at the cost per function point – which is what you really care about – the cost with the Java app is much lower. We save about $100K using Java.

Benchmark Data on Post-Install Defects

More users will find your bugs more quickly



DEFECT REPORTS IN FIRST YEAR OF USAGE

Function Points	10	100	1000	10,000	100,000
Users 1	55%	27%	12%	3%	1%
10	65%	35%	17%	7%%	3%
100	75%	42%	20%	10%	7%
1000	85%	50%	*27%*	12%	10%
10,000	95%	75%	35%	20%	12%
100,000	99%	87%	45%	35%	20%
1,000,000	100%	96%	77%	45%	32%
10,000,000	100%	100%	90%	65%	45%

SWQUAL08\41

16

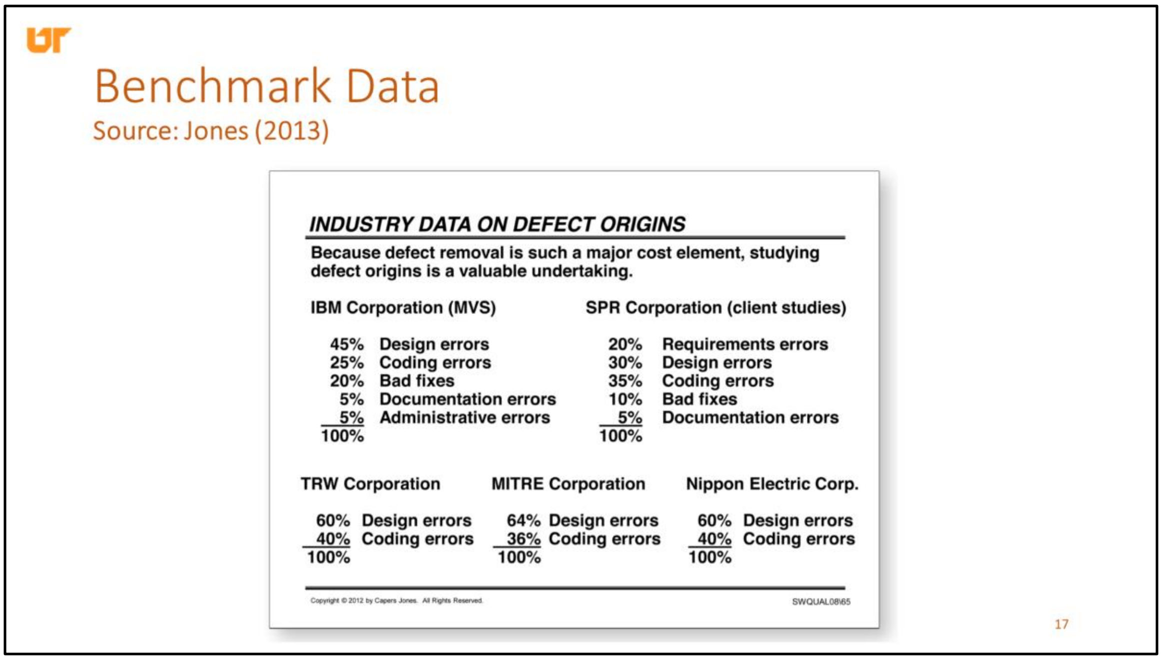
Now, let's get to some results from the study.

This table shows benchmark data for various projects with a different number of users and a different number of function points. Remember function points are just a metric related to the functional requirements that measures the total size of the project.

On the left y-axis, we have the number of users, and on the x-axis we have the number of function points.

So, this means, that with 1000 users and 1000 function points, the users will find 27% of all post-release bugs in your software.

Notice with only 1 user with a project with 1000 FPs, the user will only report 12% of all bugs. If you have 10M users with 1000 FPs, the users will report more than 90% of all bugs in your software.



Benchmark Data

Source: Jones (2013)

INDUSTRY DATA ON DEFECT ORIGINS

Because defect removal is such a major cost element, studying defect origins is a valuable undertaking.

IBM Corporation (MVS)		SPR Corporation (client studies)	
45%	Design errors	20%	Requirements errors
25%	Coding errors	30%	Design errors
20%	Bad fixes	35%	Coding errors
5%	Documentation errors	10%	Bad fixes
5%	Administrative errors	5%	Documentation errors
100%		100%	

TRW Corporation		MITRE Corporation		Nippon Electric Corp.	
60%	Design errors	64%	Design errors	60%	Design errors
40%	Coding errors	36%	Coding errors	40%	Coding errors
100%		100%		100%	

SWQUAL08\65

17

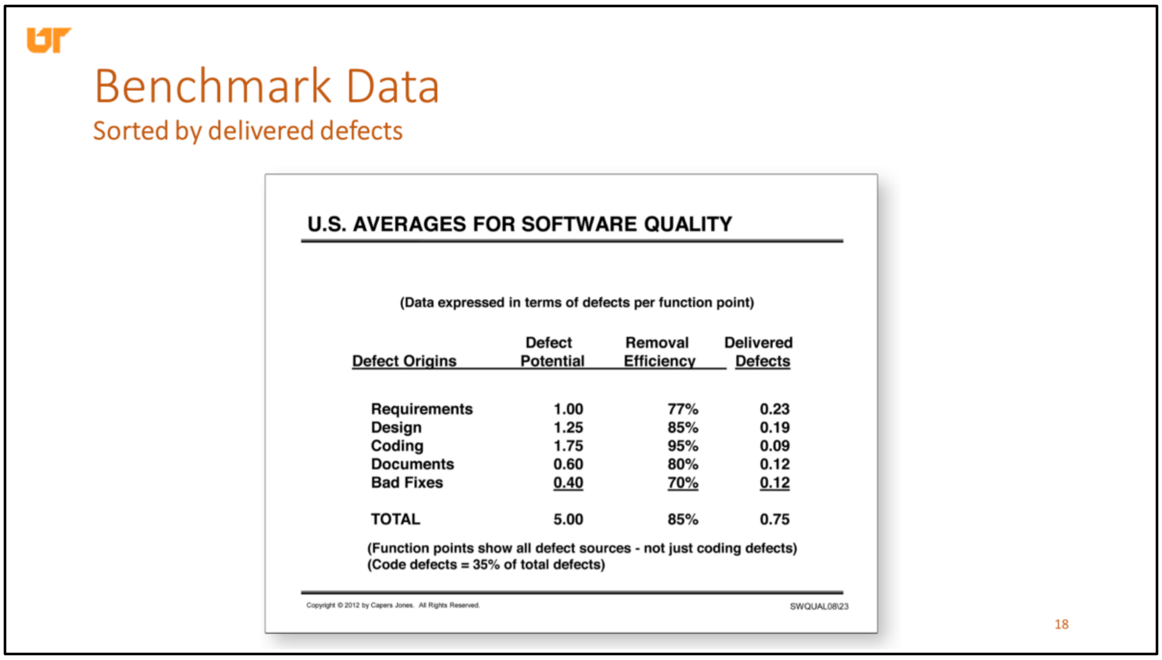
This slide puts some hard numbers on the assertion that most bugs are non-coding bugs from early in the software development process.

The study looked at industry data on the origin of various software bugs.

At IBM, they found that 45% of all defects came from the design phase. At SPR, 20% came from requirements, while another 30% came from the design phase.

The bottom line is that more defects are caused by issues in the earlier stages of software development.

Earlier bugs also happen to be the bugs that are more expensive to fix.



Benchmark Data

Sorted by delivered defects

U.S. AVERAGES FOR SOFTWARE QUALITY

(Data expressed in terms of defects per function point)

Defect Origins	Defect Potential	Removal Efficiency	Delivered Defects
Requirements	1.00	77%	0.23
Design	1.25	85%	0.19
Coding	1.75	95%	0.09
Documents	0.60	80%	0.12
Bad Fixes	0.40	70%	0.12
TOTAL	5.00	85%	0.75

(Function points show all defect sources - not just coding defects)
(Code defects = 35% of total defects)

Copyright © 2012 by Capers Jones. All Rights Reserved. SWQUAL08\23

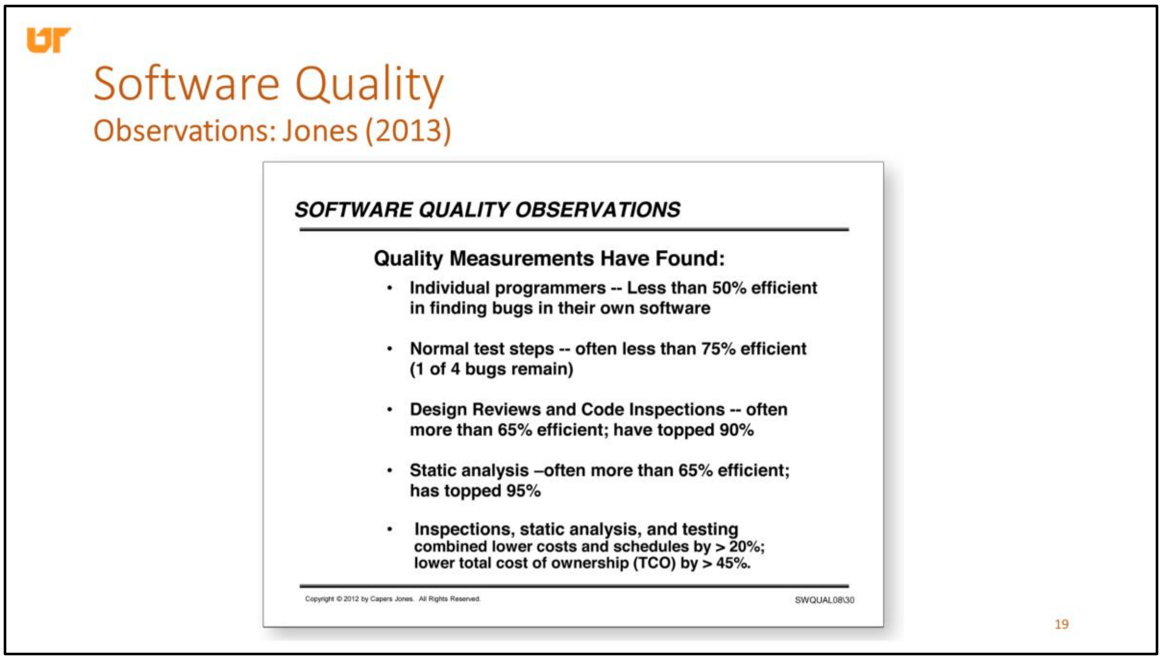
18

This slide looks at the total number of defects per function point at different stages of the development process.

So, there are substantial defects introduced in the coding phase, but it's easier to remove those errors prior to release (the average development team removes about 95% of defects introduced in the coding stage prior to release).

However, there are more defects introduced in these non-coding phases of requirements and design – and these are harder to detect and remove prior to release.

In sum, coding defects are only about 35% of total defects, and only about 12% of delivered defects.



Software Quality

Observations: Jones (2013)

SOFTWARE QUALITY OBSERVATIONS

Quality Measurements Have Found:

- Individual programmers -- Less than 50% efficient in finding bugs in their own software
- Normal test steps -- often less than 75% efficient (1 of 4 bugs remain)
- Design Reviews and Code Inspections -- often more than 65% efficient; have topped 90%
- Static analysis –often more than 65% efficient; has topped 95%
- Inspections, static analysis, and testing combined lower costs and schedules by > 20%; lower total cost of ownership (TCO) by > 45%.

Copyright © 2012 by Capers Jones. All Rights Reserved. SWQUAL08\30

19

The study is able to make a number of interesting observations.

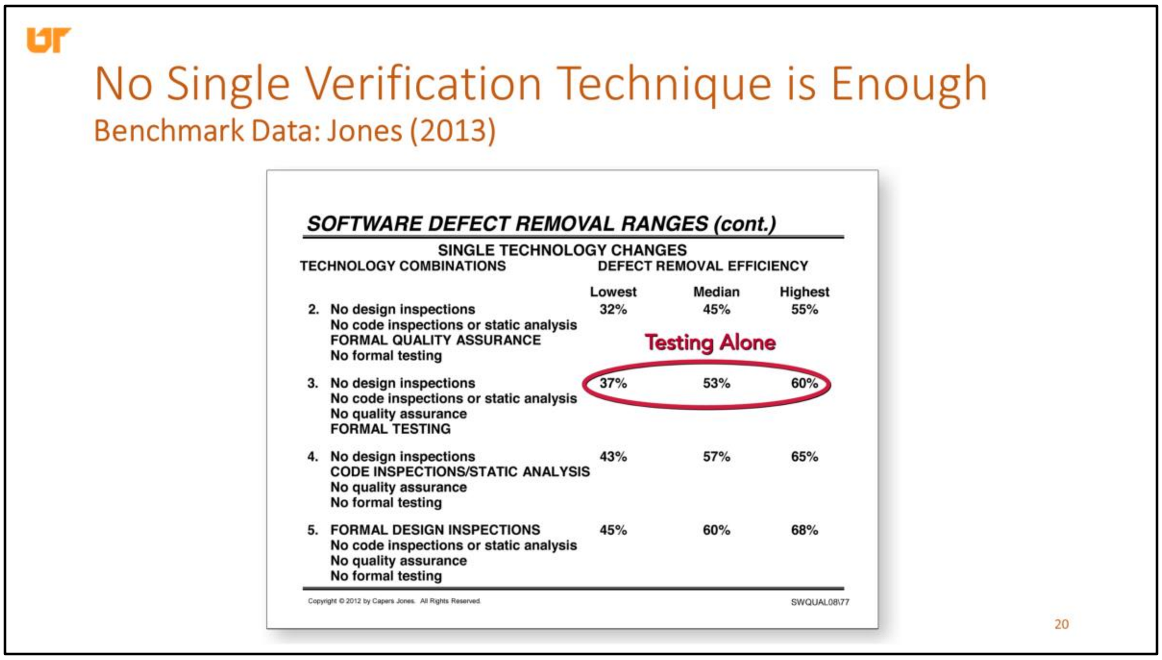
Individual programmers are typically very bad at finding defects in their own software. An individual programmer finds less than 50% of the bugs they create in their own code.

Normal testing (including unit tests, function tests, system tests, etc) is < 75% efficient at finding bugs. So, you will release code with about 25% of the bugs still in the code.

Design reviews and code inspections alone can find 65% of all bugs – in the best case – these practices can find 95% of bugs.

Static analysis is similar.

Combining the practices of design and code inspections, static analysis, and testing can lower costs and reduce development time by more than 20%, and it can reduce the total cost of ownership (including defect repair and maintenance) by more than 45%.



No Single Verification Technique is Enough

Benchmark Data: Jones (2013)

SOFTWARE DEFECT REMOVAL RANGES (cont.)

SINGLE TECHNOLOGY CHANGES

TECHNOLOGY COMBINATIONS	DEFECT REMOVAL EFFICIENCY		
	Lowest	Median	Highest
2. No design inspections No code inspections or static analysis FORMAL QUALITY ASSURANCE No formal testing	32%	45%	55%
3. No design inspections No code inspections or static analysis No quality assurance FORMAL TESTING	37%	53%	60%
4. No design inspections CODE INSPECTIONS/STATIC ANALYSIS No quality assurance No formal testing	43%	57%	65%
5. FORMAL DESIGN INSPECTIONS No code inspections or static analysis No quality assurance No formal testing	45%	60%	68%

Copyright © 2012 by Capers Jones. All Rights Reserved. SWQUAL08\77

20

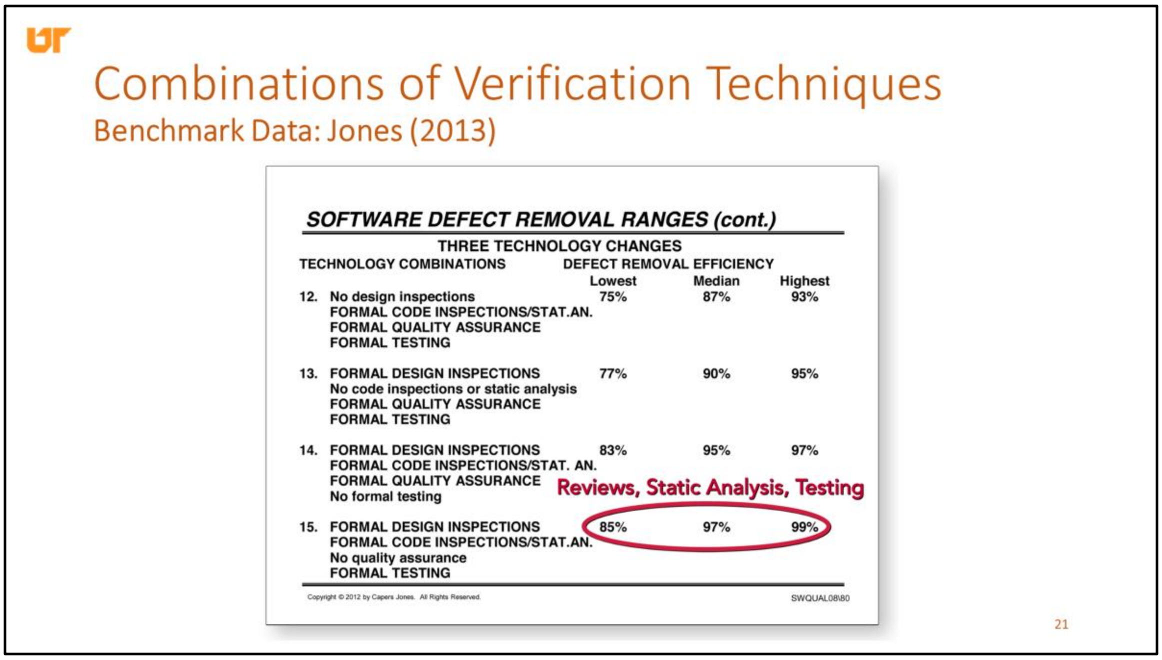
So these ftables show specifically how many bugs you should expect to find with different combinations of using four different defect removal techniques:

Design inspections
Code inspections / static analysis
Quality assurance
Formal testing

These items show the defect removal efficiency if you do each of these practices in isolation.

So, if all you do is formal testing, you should not expect to find much more than half the bugs in your software.

The best practice to do alone is formal design inspections, but even then, you'll only find about 60% of your program's bugs.



Combinations of Verification Techniques

Benchmark Data: Jones (2013)

SOFTWARE DEFECT REMOVAL RANGES (cont.)

THREE TECHNOLOGY CHANGES

TECHNOLOGY COMBINATIONS	DEFECT REMOVAL EFFICIENCY		
	Lowest	Median	Highest
12. No design inspections FORMAL CODE INSPECTIONS/STAT.AN. FORMAL QUALITY ASSURANCE FORMAL TESTING	75%	87%	93%
13. FORMAL DESIGN INSPECTIONS No code inspections or static analysis FORMAL QUALITY ASSURANCE FORMAL TESTING	77%	90%	95%
14. FORMAL DESIGN INSPECTIONS FORMAL CODE INSPECTIONS/STAT. AN. FORMAL QUALITY ASSURANCE No formal testing	83%	95%	97%
15. FORMAL DESIGN INSPECTIONS FORMAL CODE INSPECTIONS/STAT.AN. No quality assurance FORMAL TESTING	85%	97%	99%

Reviews, Static Analysis, Testing

Copyright © 2012 by Capers Jones. All Rights Reserved.

SWQUAL08\80

21

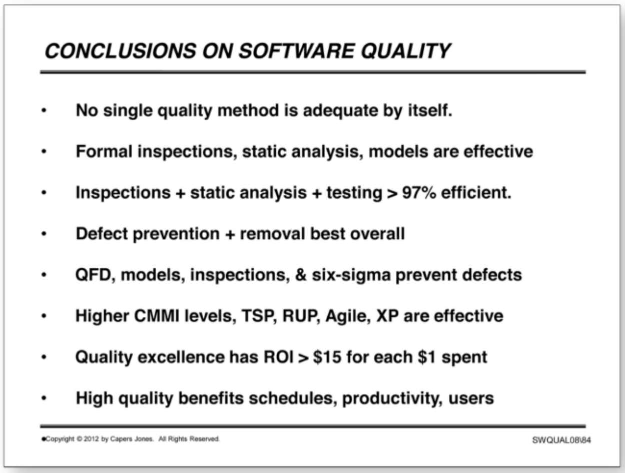
Doing these techniques together in a synergistic package yields much better results.

This slide shows DRE with different combinations of these activities.

If you do design and code inspections, as well as static analysis, as well as formal testing, you will find about 97% of all defects in your software prior to release.

Software Quality Conclusions

Jones (2013)



CONCLUSIONS ON SOFTWARE QUALITY

- **No single quality method is adequate by itself.**
- **Formal inspections, static analysis, models are effective**
- **Inspections + static analysis + testing > 97% efficient.**
- **Defect prevention + removal best overall**
- **QFD, models, inspections, & six-sigma prevent defects**
- **Higher CMMI levels, TSP, RUP, Agile, XP are effective**
- **Quality excellence has ROI > $15 for each $1 spent**
- **High quality benefits schedules, productivity, users**

©Copyright © 2012 by Capers Jones. All Rights Reserved. SWQUAL08\84

22

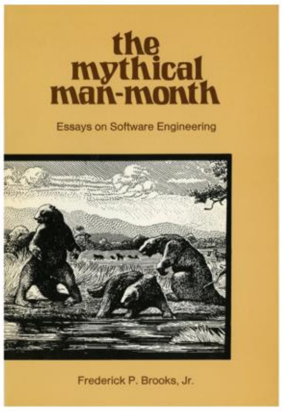
So, the main conclusion to take from this study is that no single quality method is adequate by itself.

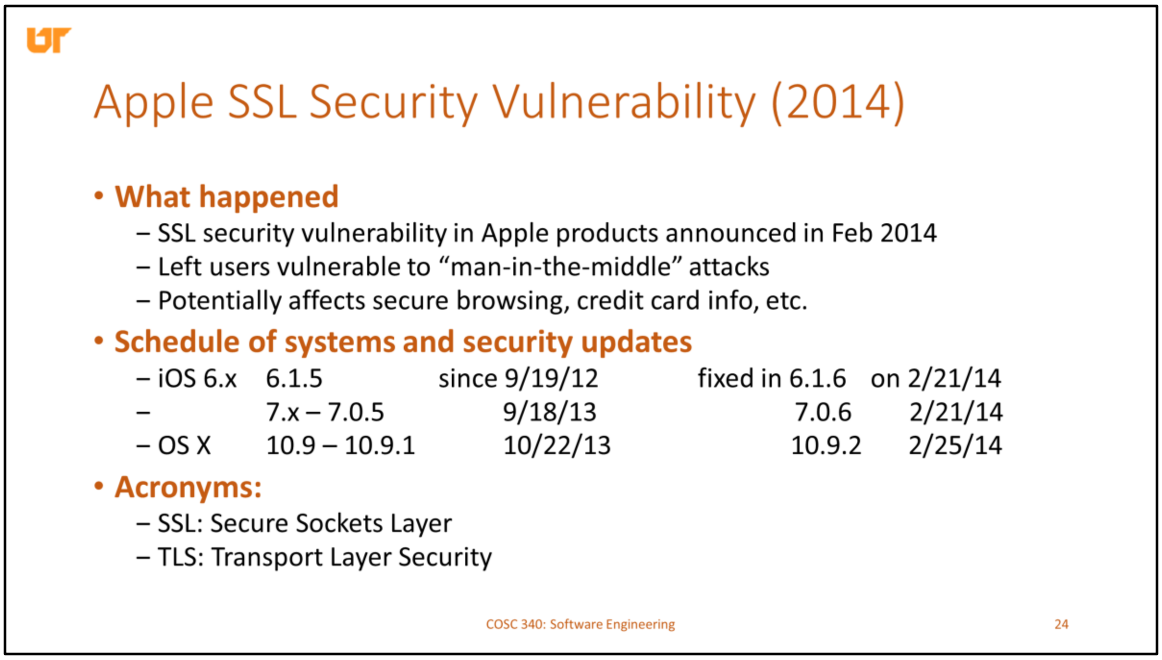
Combining formal inspections, static analysis, and testing can be quite effective at improving defect removal efficiency.

And ensuring quality in your software pays off. You'll get a $15 return on investment for each $1 spent. And high quality benefits everyone. It puts less strain on your product schedule, it can increase productivity, and is more beneficial to users.

Motivation for Static Analysis

- "Software products are among the most complex of man-made systems, and software by its very nature has intrinsic, essential properties (e.g., complexity, invisibility, and changeability) that are not easily addressed"
  - Brooks, F.P. The Mythical Man Month (1995)





Apple SSL Security Vulnerability (2014)

- **What happened**
  - SSL security vulnerability in Apple products announced in Feb 2014
  - Left users vulnerable to “man-in-the-middle” attacks
  - Potentially affects secure browsing, credit card info, etc.
- **Schedule of systems and security updates**

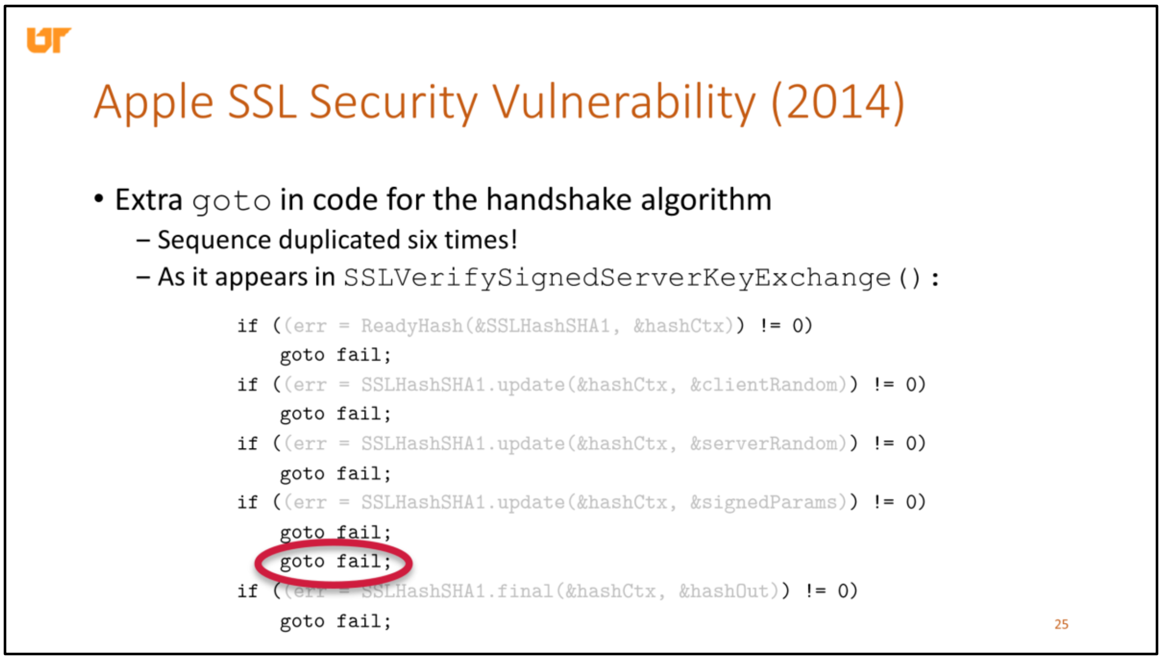
– iOS 6.x	6.1.5	since 9/19/12	fixed in 6.1.6	on 2/21/14
–	7.x – 7.0.5	9/18/13	7.0.6	2/21/14
– OS X	10.9 – 10.9.1	10/22/13	10.9.2	2/25/14

- **Acronyms:**
  - SSL: Secure Sockets Layer
  - TLS: Transport Layer Security

COSC 340: Software Engineering 24

In February 2014, Apple revealed and fixed an SSL (Secure Sockets Layer) vulnerability that had gone undiscovered since the release of iOS 6.0 in September 2012.

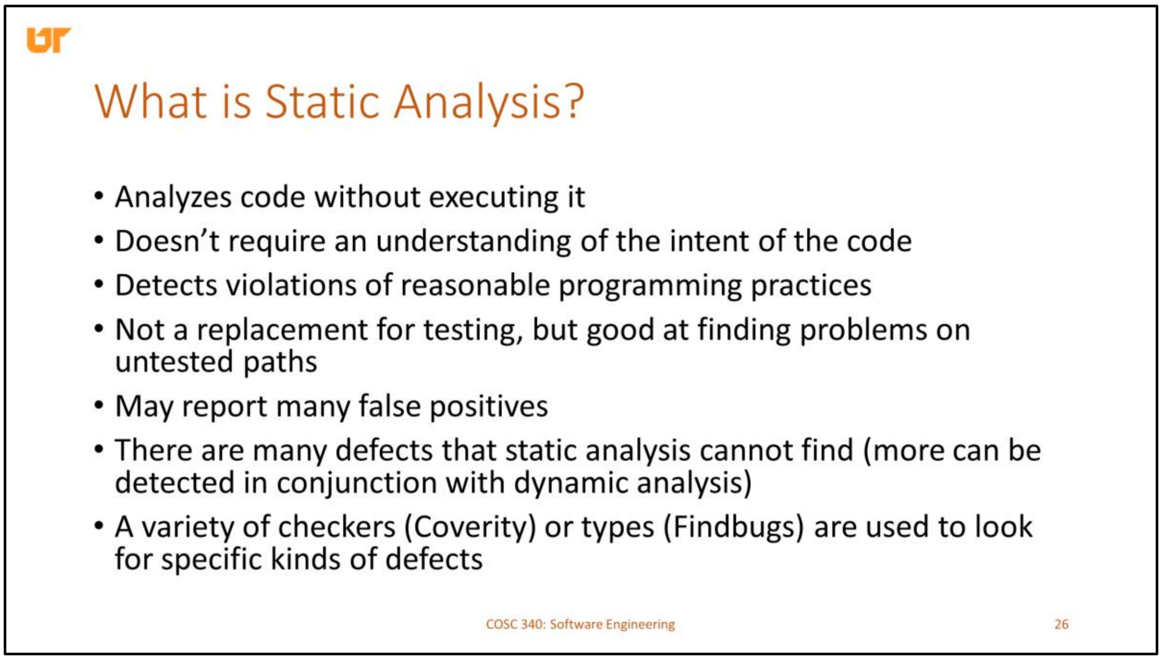
It left users vulnerable to man-in-the-middle attacks thanks to a short circuit in the SSL/TLS (Transport Layer Security) handshake algorithm introduced by the duplication of a goto statement.



Apple SSL Security Vulnerability (2014)

- Extra `goto` in code for the handshake algorithm
  - Sequence duplicated six times!
  - As it appears in `SSLVerifySignedServerKeyExchange()`:

```
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```



What is Static Analysis?

- Analyzes code without executing it
- Doesn’t require an understanding of the intent of the code
- Detects violations of reasonable programming practices
- Not a replacement for testing, but good at finding problems on untested paths
- May report many false positives
- There are many defects that static analysis cannot find (more can be detected in conjunction with dynamic analysis)
- A variety of checkers (Coverity) or types (Findbugs) are used to look for specific kinds of defects

COSC 340: Software Engineering 26

Today we’re going to discuss static analysis.

Static analysis are great tools for improving your code without too much effort. You can run static analysis tools similar to how you run your compiler on your source program.

You don’t need much understanding of the source code to run static analysis on it – you can just run the tool and it will start giving recommendations.

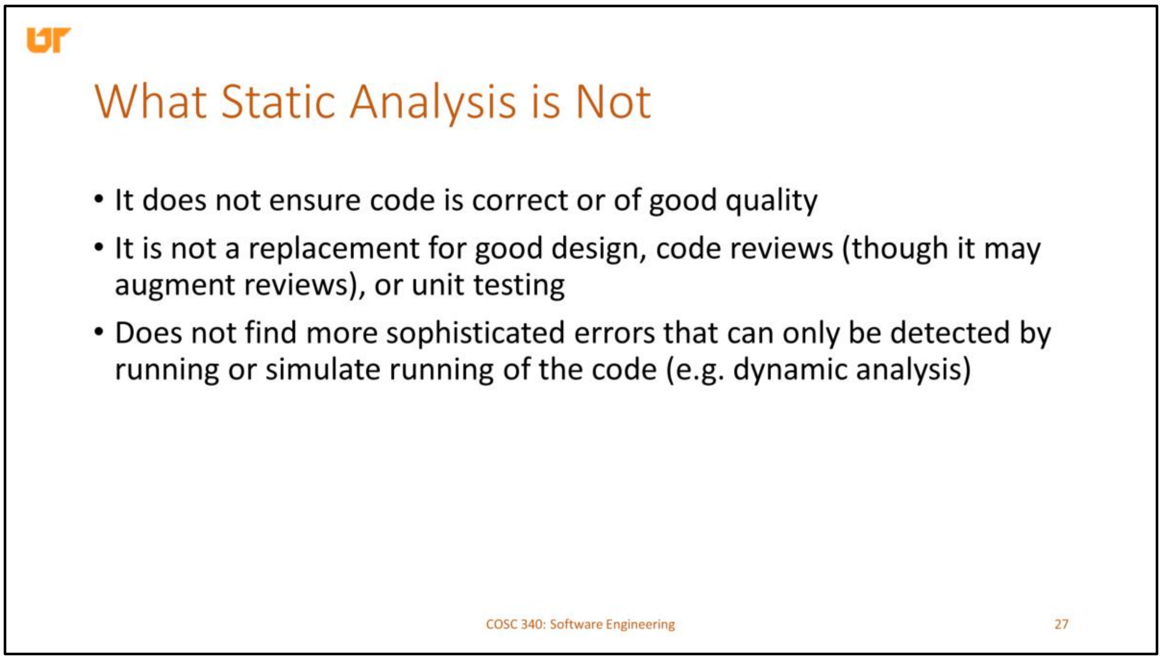
In addition to finding potential errors, they can also tell you when you’re program does not conform to a specific style or violates some reasonable programming practice.

While SA is not a replacement for testing, it is certainly a useful supplement. It can help you find problems you might not detect in testing – or show you paths of your code that you haven’t covered with your tests.

A major issue with SA is the problem of false positives. Since static analysis makes its recommendations without program input, there are many cases where it might report a problem that is not really a problem at all because of the nature of your input. Finding a balance between reporting the most important issues and overwhelming the user with false positives or trivial issues is a challenge.

Additionally, there are many issues that static analysis is not able to solve. For instance, run-time errors such as null pointer dereference are difficult for SA's to detect. Thus, you should always use SA in conjunction with testing and perhaps dynamic analysis.

As we'll see, there are a variety of SA tools – each of which focuses on specific kinds of defects. Depending on your project, you might want to combine multiple tools to get the best result.



What Static Analysis is Not

- It does not ensure code is correct or of good quality
- It is not a replacement for good design, code reviews (though it may augment reviews), or unit testing
- Does not find more sophisticated errors that can only be detected by running or simulate running of the code (e.g. dynamic analysis)

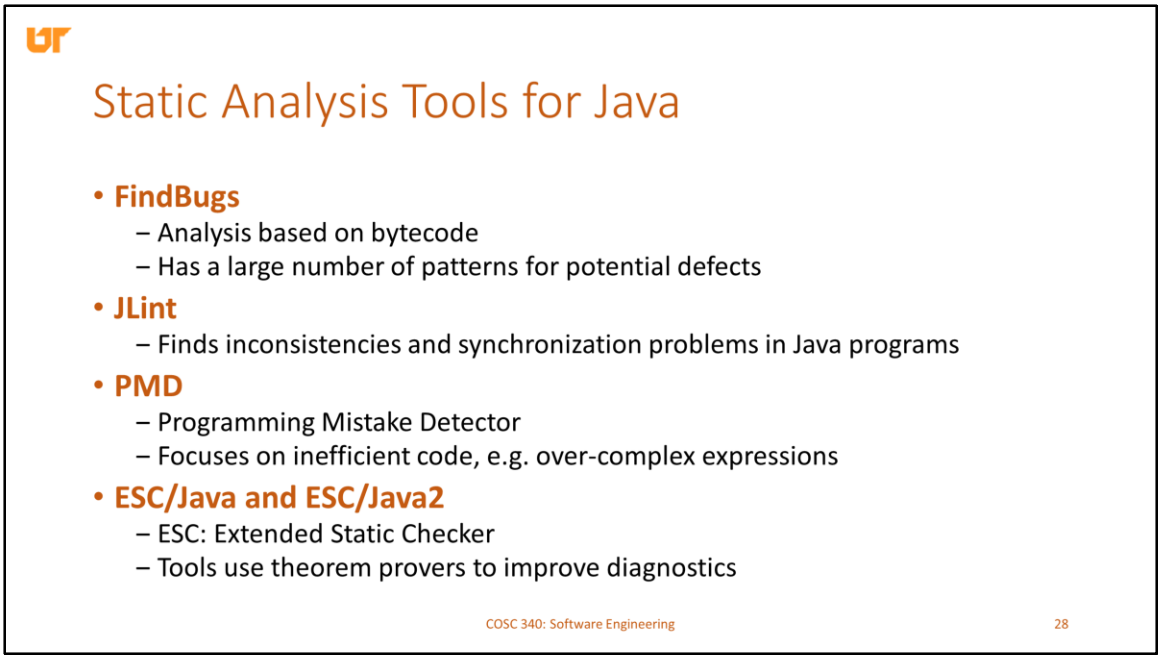
COSC 340: Software Engineering 27

Now, despite it’s advantages, SA is not a panacea.

SA can tell you whether your code violates some specific rules or practices and give recommendations, but it does not ensure your code is correct or good quality.

So, it’s not a replacement for good design, regular design and code reviews, and standard testing techniques. It can be a useful tool to augment these practices.

It also cannot find more sophisticated errors (such as performance or memory errors) that can only be detected by running or simulating the code with dynamic analysis.


Static Analysis Tools for Java

- **FindBugs**
  - Analysis based on bytecode
  - Has a large number of patterns for potential defects
- **JLint**
  - Finds inconsistencies and synchronization problems in Java programs
- **PMD**
  - Programming Mistake Detector
  - Focuses on inefficient code, e.g. over-complex expressions
- **ESC/Java and ESC/Java2**
  - ESC: Extended Static Checker
  - Tools use theorem provers to improve diagnostics

COSC 340: Software Engineering 28

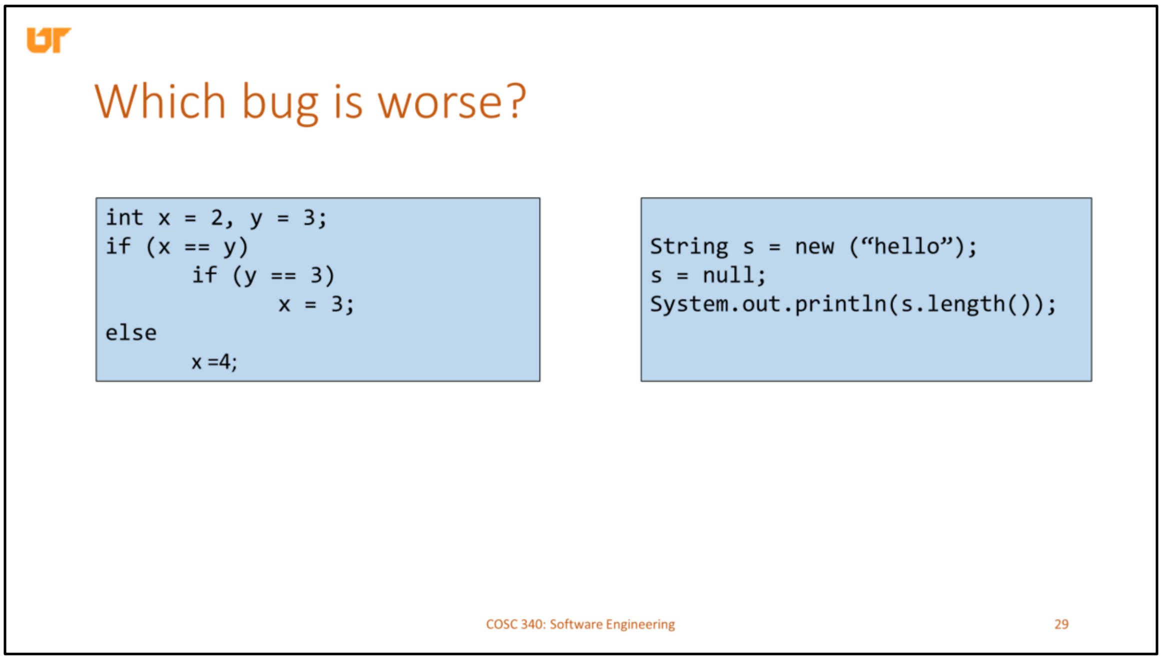
Let’s look at a few different static analysis tools.

FindBugs is a static analysis tool that examines Java bytecode (not the raw source code)
It searches for a large number of patterns in your bytecode to find ones that lead to common mistakes. FindBugs can detect issues such as “return value of method ignored”, “null pointer dereference”, and “redundant comparisons to null”

Jlint does data flow analysis and constructs the lock graph to find inconsistencies and synchronization problems in your Java programs. It might find methods that can be called from different threads, locks that are held but never released, or locking patterns that might lead to deadlock (e.g. lock A is requested while holding lock B, while another thread can hold A and also be requesting B)

PMD stands for programming mistake detector. PMD can find some of the bugs detected by FindBugs – but it operates at the source code level – not at the bytecode level. It also tries to simplify your code by searching for common mistakes or patterns that can complicate it (such as overcomplicated if statements, dead code, duplicate code, and empty try/catch/finally and switch blocks).

The ESC/Java and ESC/Java2 tools attempt to find common run-time errors in Java at compile time. These tools use an extended static checking approach, which you can think of as an extended form of type checking. It aims to identify errors such as divide by zero, array out of bounds, integer overflow, and null dereference.



Which bug is worse?

```
int x = 2, y = 3;
if (x == y)
        if (y == 3)
                x = 3;
else
        x =4;
```

```
String s = new (“hello”);
s = null;
System.out.println(s.length());
```

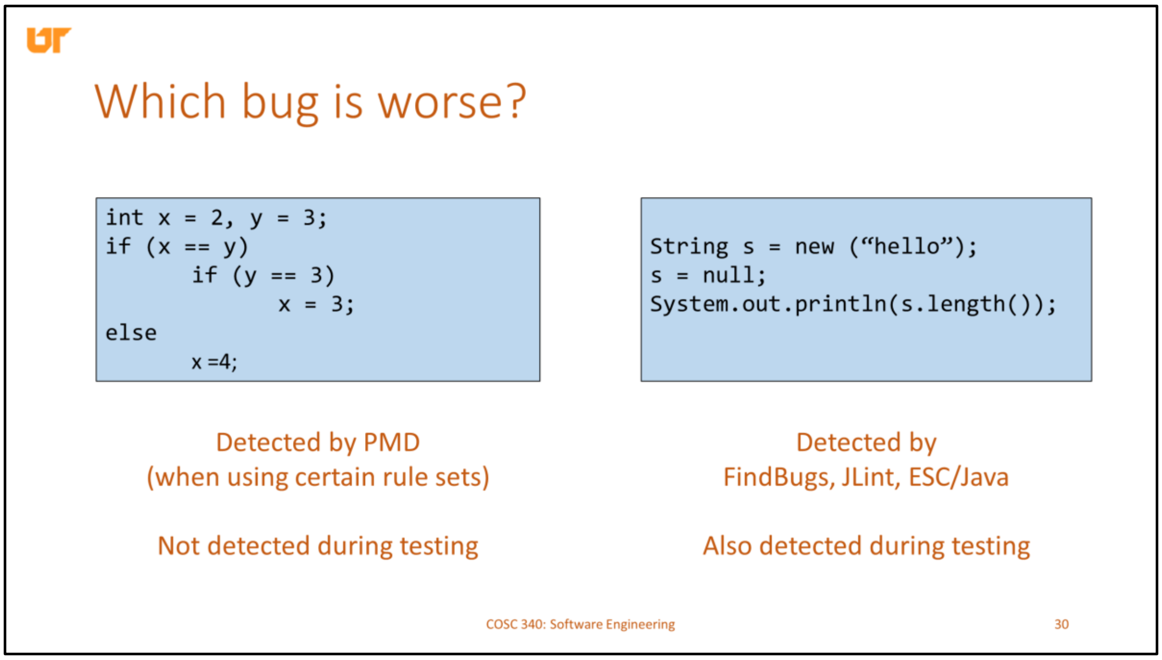
COSC 340: Software Engineering 29

Now, I want to show you a few examples of the types of things you can find with static analysis.

Here we have two bugs.

Which is worse?

(on the left) set x to 4 when x != y, but we set it to 4 when x==y and y!=3
(on the right) null pointer dereference



Which bug is worse?

```
int x = 2, y = 3;
if (x == y)
        if (y == 3)
                x = 3;
else
        x =4;
```

```
String s = new (“hello”);
s = null;
System.out.println(s.length());
```

Detected by PMD
(when using certain rule sets)

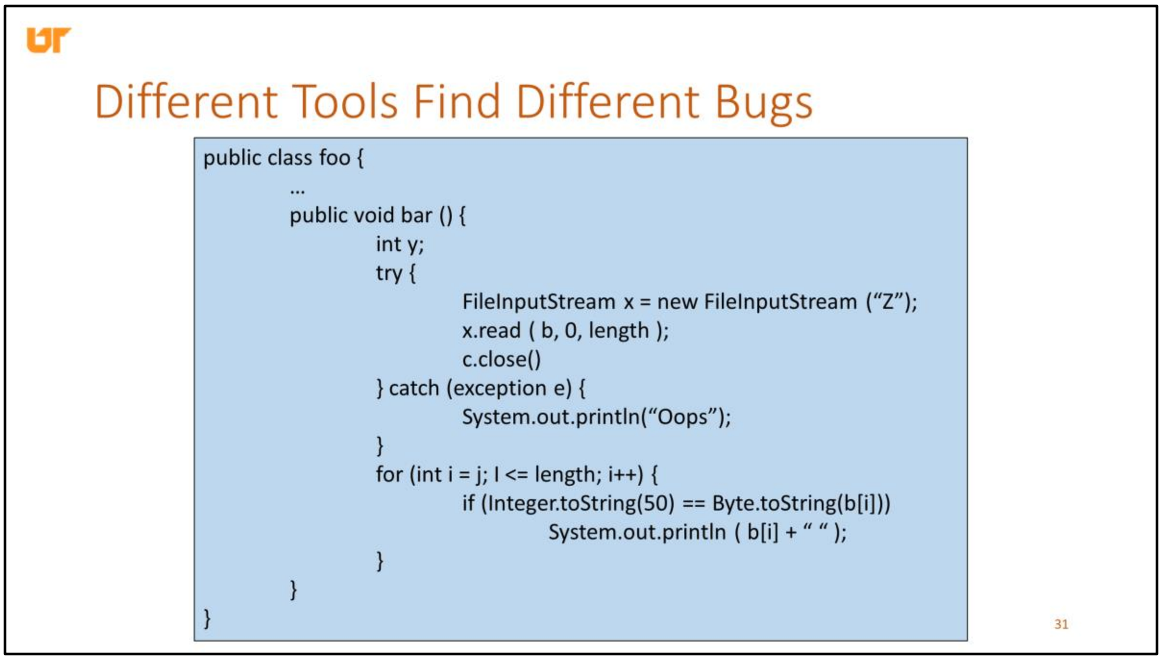
Not detected during testing

Detected by
FindBugs, JLint, ESC/Java

Also detected during testing

Left is harder to detect with testing. SA can alert you and avoid issues later.

Right may always cause a critical failure, but will likely be detected in testing.

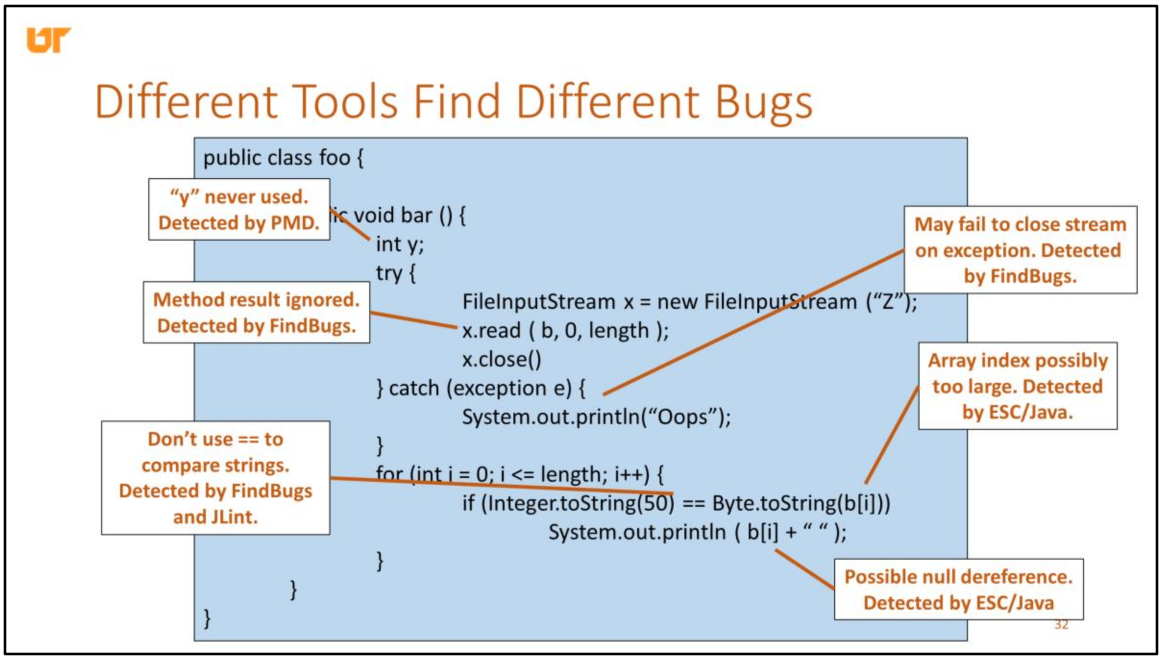


Different Tools Find Different Bugs

```
public class foo {
        …
        public void bar () {
                int y;
                try {
                        FileInputStream x = new FileInputStream ("Z");
                        x.read ( b, 0, length );
                        c.close()
                } catch (exception e) {
                        System.out.println("Oops");
                }
                for (int i = j; I <= length; i++) {
                        if (Integer.toString(50) == Byte.toString(b[i]))
                                System.out.println ( b[i] + " " );
                }
        }
}
```

31

Where are the bugs in this code?



Different Tools Find Different Bugs

```
public class foo {

        ...ic void bar () {
                int y;
                try {
                        FileInputStream x = new FileInputStream ("Z");
                        x.read ( b, 0, length );
                        x.close()
                } catch (exception e) {
                        System.out.println("Oops");
                }
                for (int i = 0; i <= length; i++) {
                        if (Integer.toString(50) == Byte.toString(b[i]))
                                System.out.println ( b[i] + " " );
                }
        }
}
```

"y" never used. Detected by PMD.
Method result ignored. Detected by FindBugs.
Don't use == to compare strings. Detected by FindBugs and JLint.
May fail to close stream on exception. Detected by FindBugs.
Array index possibly too large. Detected by ESC/Java.
Possible null dereference. Detected by ESC/Java
32

y never used.

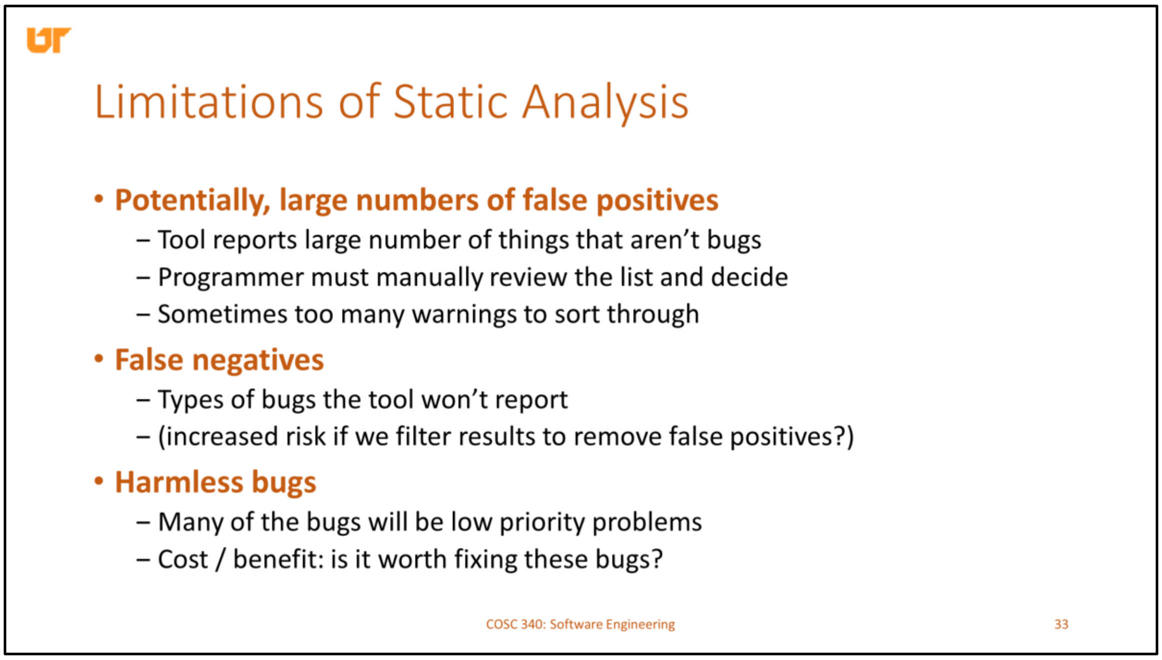
Method result ignored. In this case, read returns the number of bytes read or -1 if there is no more data to be read.

Don't use == to compare strings. (== checks if they are the same object, need to use .equals to check if two different objects have the same value).

May fail to close stream on exception (x is never closed).

Array index possibly too large. (i goes from 0 to length – code doesn't say the size of b).

Possible null dereference (don't know the size of b).



Limitations of Static Analysis

- **Potentially, large numbers of false positives**
  - Tool reports large number of things that aren't bugs
  - Programmer must manually review the list and decide
  - Sometimes too many warnings to sort through
- **False negatives**
  - Types of bugs the tool won't report
  - (increased risk if we filter results to remove false positives?)
- **Harmless bugs**
  - Many of the bugs will be low priority problems
  - Cost / benefit: is it worth fixing these bugs?

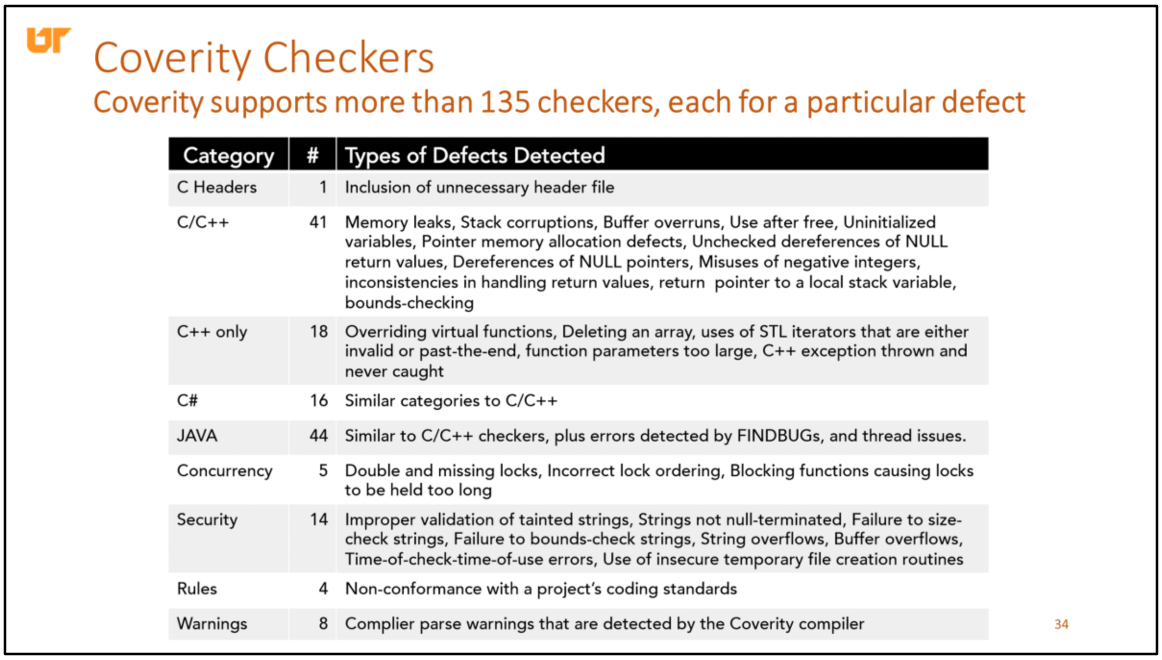
COSC 340: Software Engineering 33

While it can be useful, there are a number of limitations to static analysis.

False positives:
Tool will often report issues that aren't really bugs. Have to manually review. Sometimes too many warnings to sort through.

False negatives:
Many bugs the tool won't report. Some tools (like ESC/Java) intentionally limit what they report so they don't show you too many false positives. Striking the right balance is a challenge for SA tools.

Another issue related to false positives is that of harmless bugs. Many bugs will be low-priority problems. (for instance – unused variable or dead code – might be removed by compiler anyway). Might not be worth fixing. Clutter the output of the tool – makes it less useful.



Coverity Checkers

Coverity supports more than 135 checkers, each for a particular defect

Category	#	Types of Defects Detected
C Headers	1	Inclusion of unnecessary header file
C/C++	41	Memory leaks, Stack corruptions, Buffer overruns, Use after free, Uninitialized variables, Pointer memory allocation defects, Unchecked dereferences of NULL return values, Dereferences of NULL pointers, Misuses of negative integers, inconsistencies in handling return values, return pointer to a local stack variable, bounds-checking
C++ only	18	Overriding virtual functions, Deleting an array, uses of STL iterators that are either invalid or past-the-end, function parameters too large, C++ exception thrown and never caught
C#	16	Similar categories to C/C++
JAVA	44	Similar to C/C++ checkers, plus errors detected by FINDBUGs, and thread issues.
Concurrency	5	Double and missing locks, Incorrect lock ordering, Blocking functions causing locks to be held too long
Security	14	Improper validation of tainted strings, Strings not null-terminated, Failure to size-check strings, Failure to bounds-check strings, String overflows, Buffer overflows, Time-of-check-time-of-use errors, Use of insecure temporary file creation routines
Rules	4	Non-conformance with a project's coding standards
Warnings	8	Complier parse warnings that are detected by the Coverity compiler

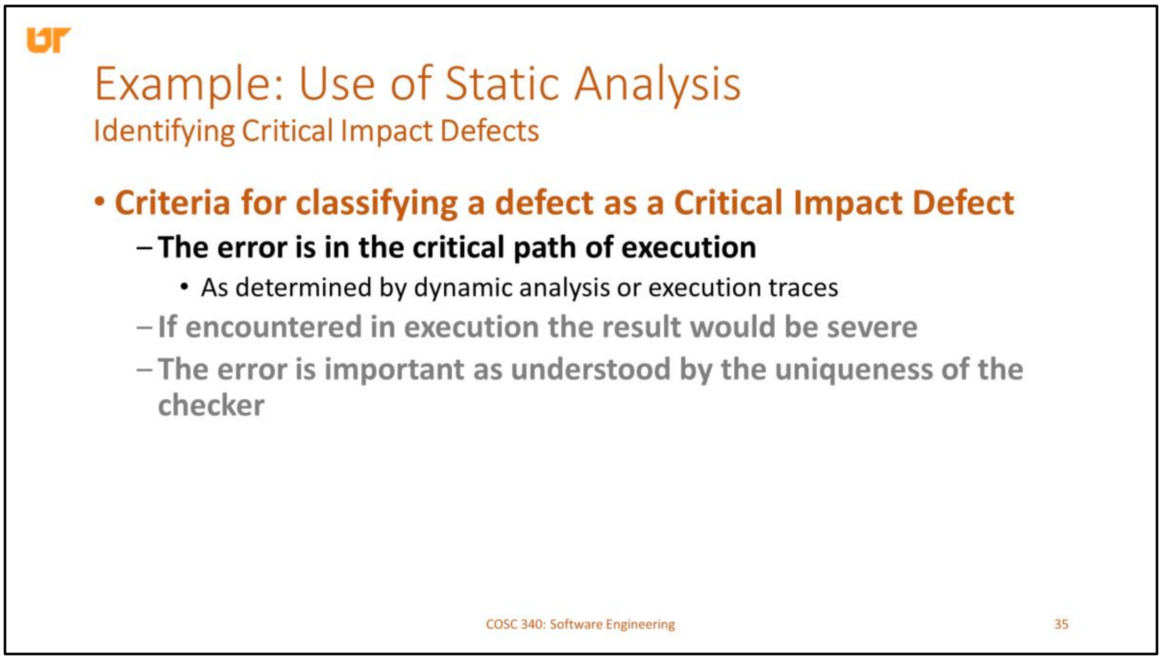
34

Next, let's talk about coverity.

Coverity is a brand of software development products from Synopsys. Coverity originated from a group out of Stanford who were building static analysis in their research lab. Today, it is one of the most comprehensive sets of static and dynamic analysis tools available. It has been quite successful on the market and was acquired by Synopses in 2014 for $350M.

Quote from creator: "The tool, like all static bug finders, leveraged the fact that programming rules often map clearly to source code; thus static inspection can find many of their violations."

The slide lists some of the different capabilities of coverity. Many of these are standard static analysis – but having them all in one tool that works with many different compilers across a variety of languages makes it a very powerful product.


Example: Use of Static Analysis
Identifying Critical Impact Defects

- **Criteria for classifying a defect as a Critical Impact Defect**
  - **The error is in the critical path of execution**
    - As determined by dynamic analysis or execution traces
  - **If encountered in execution the result would be severe**
  - **The error is important as understood by the uniqueness of the checker**

COSC 340: Software Engineering 35

Now, I'm going to show you some example use cases of the coverity checker.

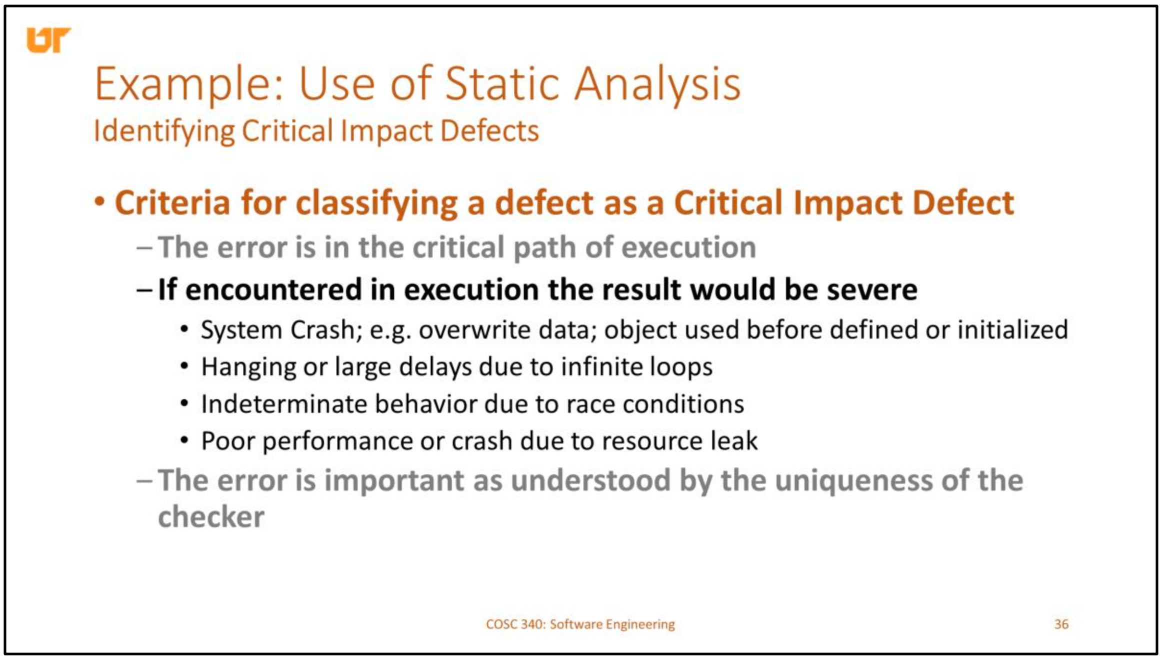
First, however, let's discuss a little more about the types of issues coverity and other SA tools can detect.

As I mentioned before, one issue with SA is that it sometimes often reports many trivial or harmless bugs and false positives.

So, we'd like to be able to quickly identify those defects that are going to be severe in nature so that we can focus our efforts. These severe defects are also called critical defects or critical impact defects.

There are a number of criteria that static and analysis and testing tools can look for when trying to determine whether a defect is critical impact or not. SA tools like coverity use these criteria to prioritize the reporting of certain types of issues.

The first criterion for critical defects is whether the error is on a critical execution path. Since SA does not have access to run-time information, we would need to use some sort of dynamic analysis or execution traces to determine which code paths are critical.



Example: Use of Static Analysis

Identifying Critical Impact Defects

- **Criteria for classifying a defect as a Critical Impact Defect**
  - **The error is in the critical path of execution**
  - **If encountered in execution the result would be severe**
    - System Crash; e.g. overwrite data; object used before defined or initialized
    - Hanging or large delays due to infinite loops
    - Indeterminate behavior due to race conditions
    - Poor performance or crash due to resource leak
  - **The error is important as understood by the uniqueness of the checker**

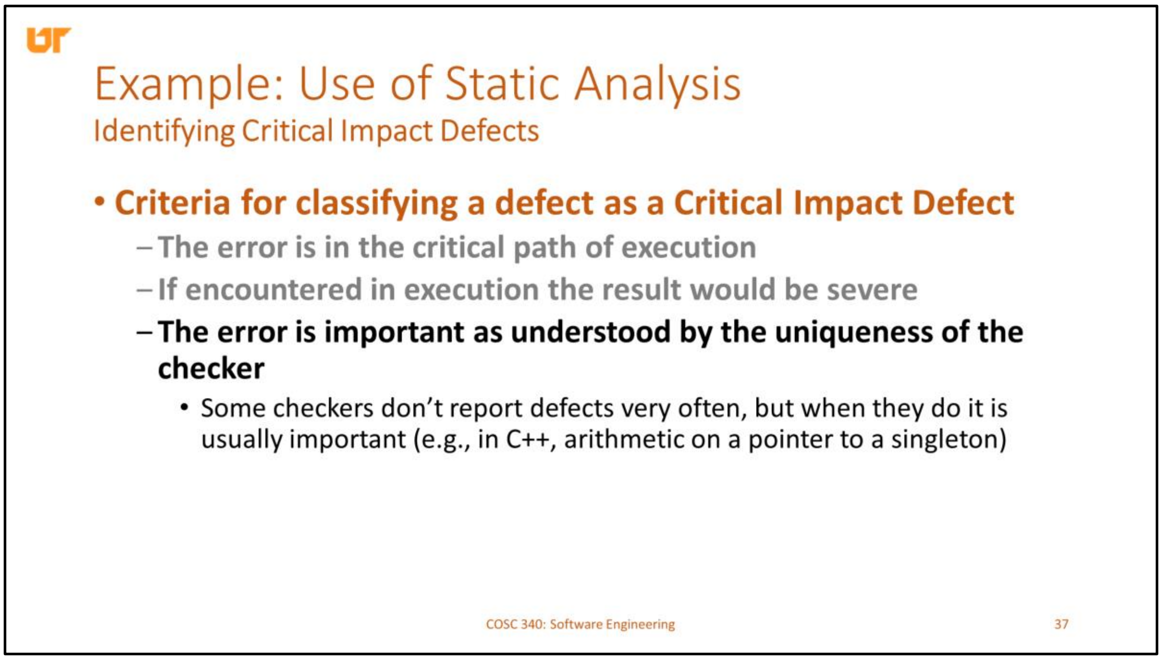
COSC 340: Software Engineering 36

Next, a defect is critical if, when it is encountered during execution, the result of the defect is severe.

For instance, the defect might cause a crash or result in hanging or large delays due to infinite loops.

Race conditions that cause inconsistent behavior as well as performance and memory issues are classified as critical impact.

Some of these things, such as infinite loops or memory leaks, are sometimes easy to detect statically.


Example: Use of Static Analysis
Identifying Critical Impact Defects

- **Criteria for classifying a defect as a Critical Impact Defect**
  - The error is in the critical path of execution
  - If encountered in execution the result would be severe
  - **The error is important as understood by the uniqueness of the checker**
    - Some checkers don't report defects very often, but when they do it is usually important (e.g., in C++, arithmetic on a pointer to a singleton)

COSC 340: Software Engineering 37

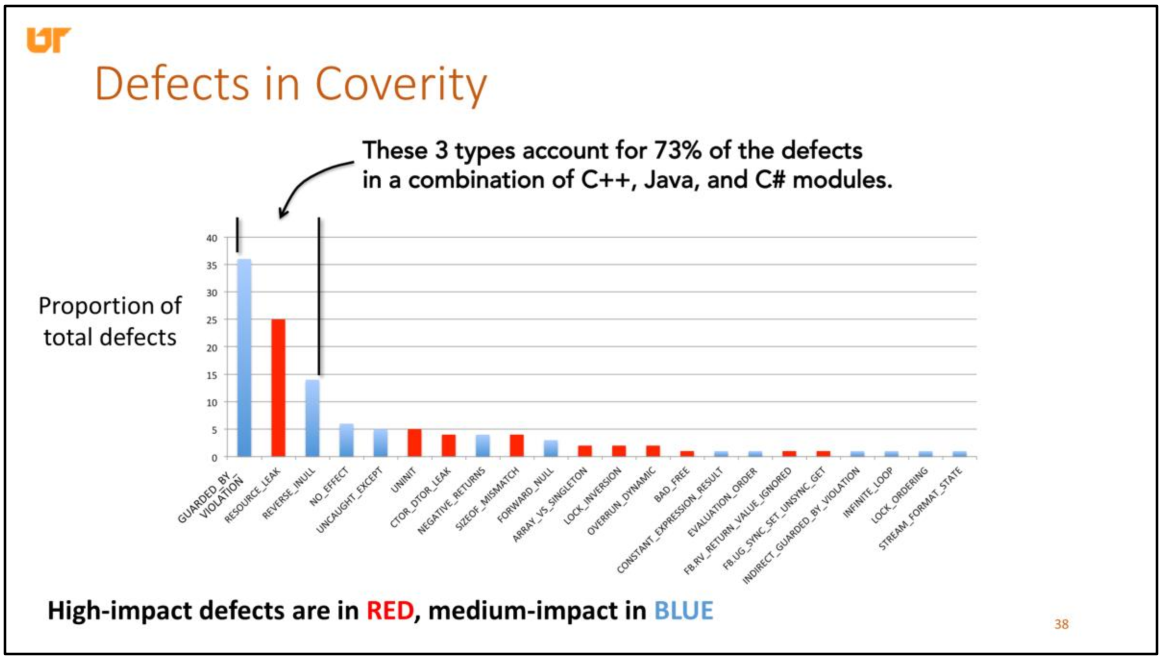
Another criterion that might be used is that some errors correspond to unique patterns that do not typically occur, but when they do, they are likely important.

An example of this might be when you try to use a singleton object as an array or do pointer arithmetic on it.

For instance,

```
void foo(char **result) {
  *result = (char*)malloc(80);
  if (...) {
    strcpy(*result, "some result string");
  } else {
    ...
    result[79] = 0; // Should be "(*result)[79] = 0“
   }
}

void bar() {
  char *s;
  foo(&s); // Defect reported here
}
```


Defects in Coverity

These 3 types account for 73% of the defects in a combination of C++, Java, and C# modules.

High-impact defects are in RED, medium-impact in BLUE

38

This figure shows the proportion of defects found by different checkers in the coverity and FindBugs systems across a range of software projects.

Using the critical-impact criteria, Coverity and FindBugs attempt to categorize defects as high-impact, medium-impact, and low-impact. Defects categorized as high-impact are more likely to be critical.

In this figure, checks that correspond to high-impact defects are shown in red, while checks that correspond to medium impact are shown in blue. Even if a defect is categorized as medium impact, it might still be critical, depending on how it affects the execution.

The most commonly reported defect is GUARDED_BY_VIOLATION. This checker infers guarded-by-relationships to track when fields are updated with known locks.

Example:

lock(myLock) {

```
  myData++;
  myData--;
}
…
myData++;
```

GUARDED_BY_VIOLATION, along with RESOURCE_LEAK, and REVERSE_INULL make up 73% of all reported defects. Of these most common defects, only RESOURCE_LEAK is classified as high-impact.

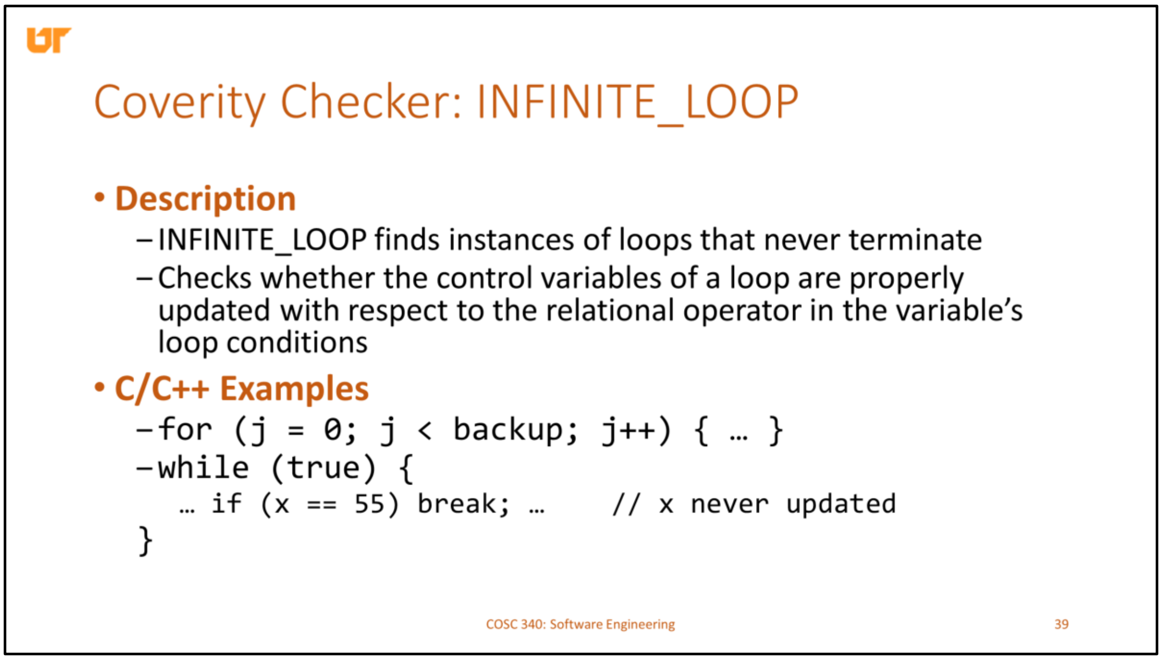
UNINIT looks for variables that are used without being initialized.

CTOR_DTOR_LEAK is where a constructor allocates memory and stores a pointer to it in an object field but the destructor does not free the memory

```
struct A {
  int *p;
  A() { p = new int; }
  ~A() { /*oops, leak*/ }
};
```

OVERRUN_DYNAMIC searches for array out of bounds errors.

Let's take a look at what some of these other checkers do.

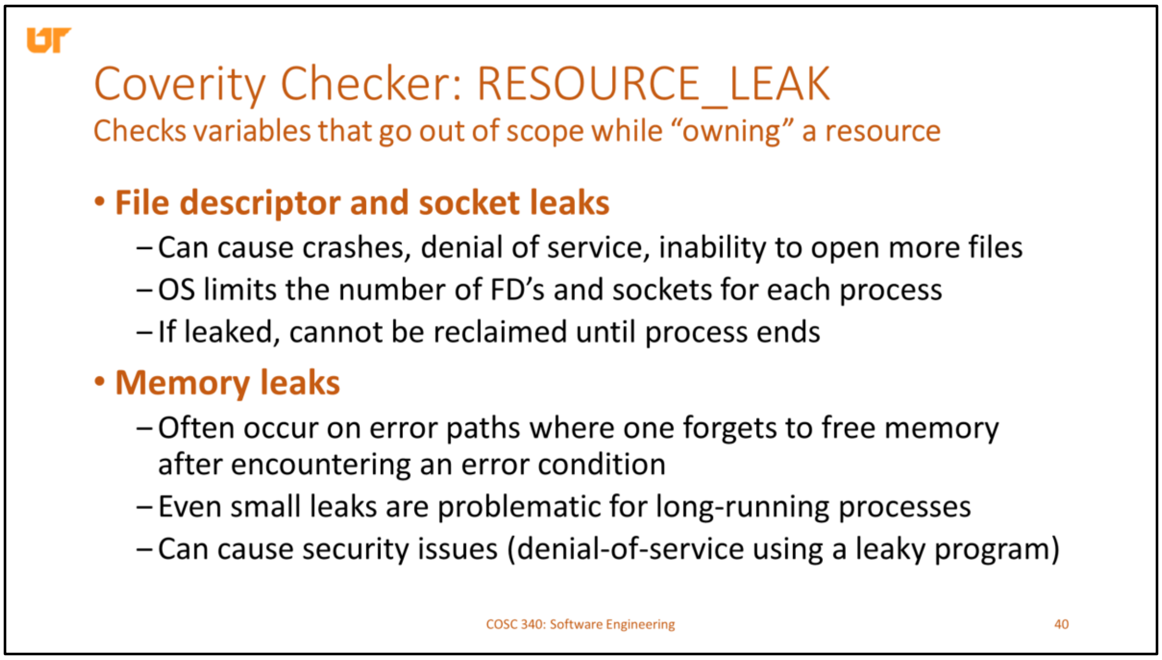


Coverity Checker: INFINITE_LOOP

- **Description**
  - INFINITE_LOOP finds instances of loops that never terminate
  - Checks whether the control variables of a loop are properly updated with respect to the relational operator in the variable's loop conditions
- **C/C++ Examples**
  - `for (j = 0; j < backup; j++) { … }`

  -
    ```
    while (true) {
      … if (x == 55) break; …    // x never updated
    }
    ```

COSC 340: Software Engineering 39



Coverity Checker: RESOURCE_LEAK

Checks variables that go out of scope while "owning" a resource

- **File descriptor and socket leaks**
  - Can cause crashes, denial of service, inability to open more files
  - OS limits the number of FD's and sockets for each process
  - If leaked, cannot be reclaimed until process ends
- **Memory leaks**
  - Often occur on error paths where one forgets to free memory after encountering an error condition
  - Even small leaks are problematic for long-running processes
  - Can cause security issues (denial-of-service using a leaky program)

COSC 340: Software Engineering 40

The resource_leak checker attempts to check if program variables go out of scope while you still "own" the resource.

It checks for two main types of leaks. The first is file descriptors and socket leaks. So, basically when you open a socket, pipe, or file, but forget to close it.

These are dangerous leaks because they can cause crashes, can be exploited for denial-of-service, and they may restrict your process from opening new files. Most OS's put a limit on the number of file descriptors you can have open for each process – so leaking file descriptor and socket descriptors will cause problems.

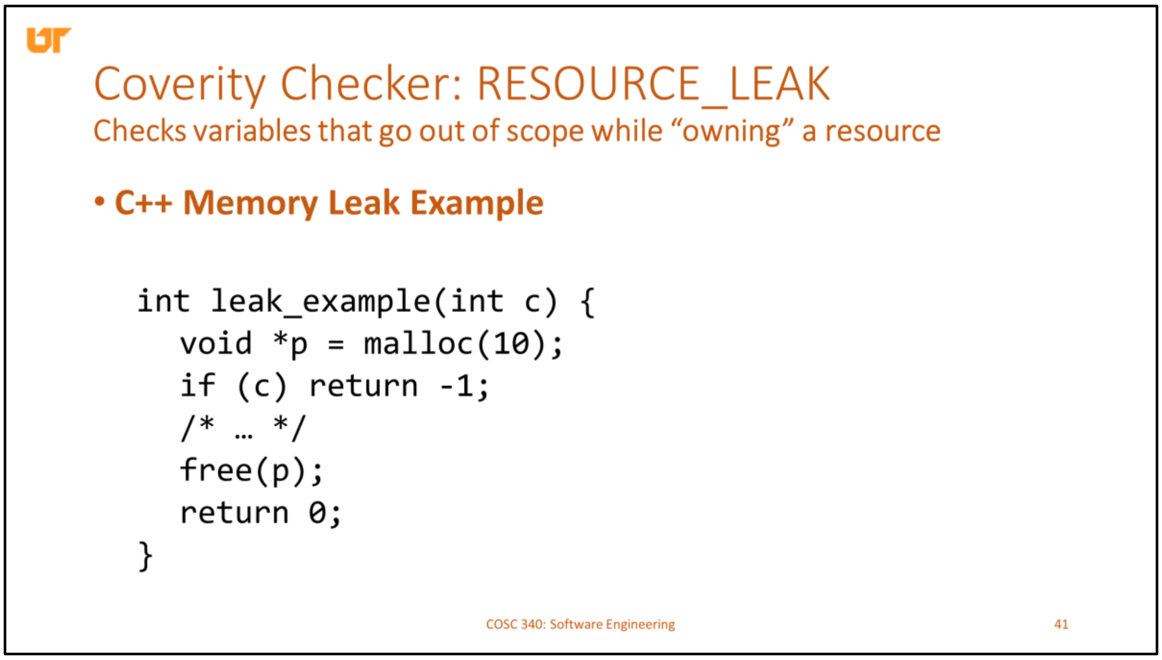
Also, just like with memory, if a FD leaks – there's really no way to free it until the process is completed

The other type of leak this checker detects is memory leaks. We've all seen memory leaks.

Very common on error paths.

Even small leaks can be problematic for long-running processes.

And leaks can also cause security issues. If you have a leak in program that an adversarial user can run, they could use it as a denial of service attack on your system.



Coverity Checker: RESOURCE_LEAK

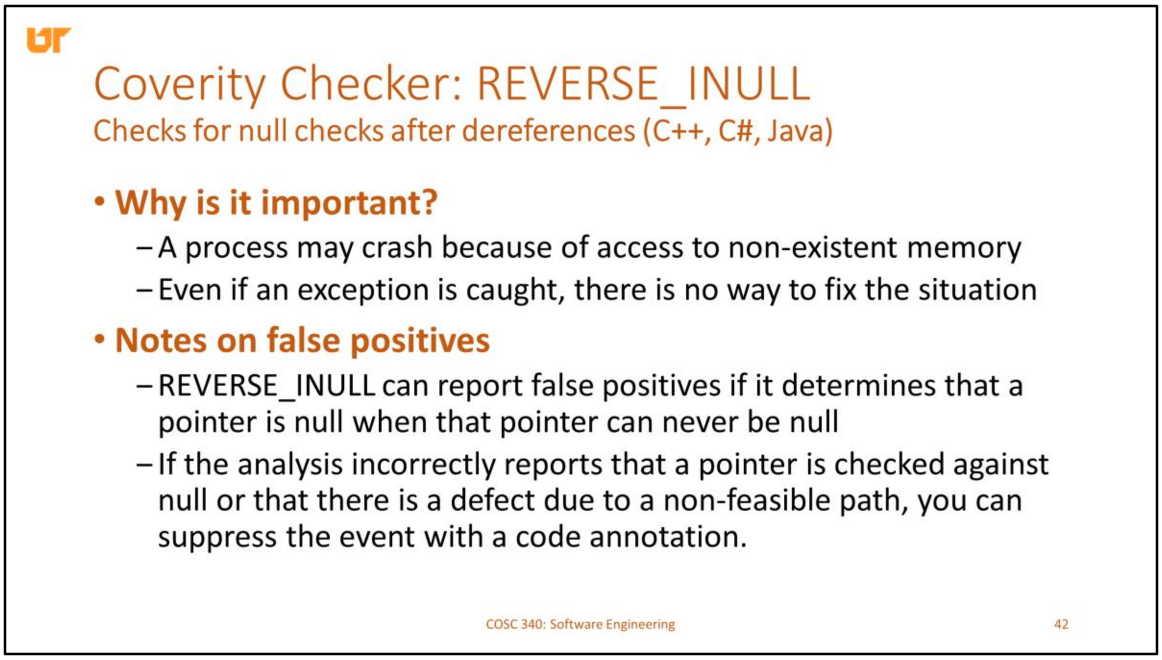
Checks variables that go out of scope while “owning” a resource

- **C++ Memory Leak Example**

```
int leak_example(int c) {
  void *p = malloc(10);
  if (c) return -1;
  /* … */
  free(p);
  return 0;
}
```

COSC 340: Software Engineering 41

Example shows the common case of not freeing on an error condition.



Coverity Checker: REVERSE_INULL

Checks for null checks after dereferences (C++, C#, Java)

- **Why is it important?**
  - A process may crash because of access to non-existent memory
  - Even if an exception is caught, there is no way to fix the situation
- **Notes on false positives**
  - REVERSE_INULL can report false positives if it determines that a pointer is null when that pointer can never be null
  - If the analysis incorrectly reports that a pointer is checked against null or that there is a defect due to a non-feasible path, you can suppress the event with a code annotation.

COSC 340: Software Engineering 42

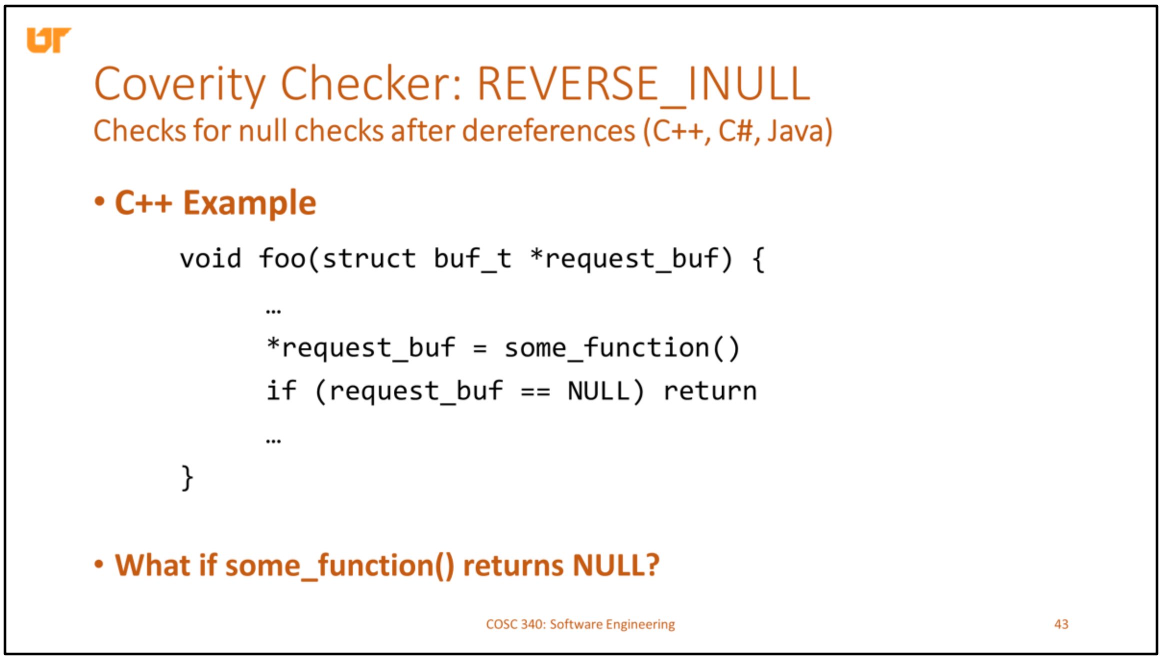
Next, is the reverse_inull checker. This was the third most common type of defect reported by coverity and findbugs.

So, basically, this checker checks for when you have a null check after you've already dereferenced the variable. It gets its name because the null check and dereference appear to be reversed in the code.

This is obviously important because dereferencing a null pointer will cause the program to crash, so you always want to make sure you do the null check before you dereference. Even in a high-level language, if you were to catch an exception for a null pointer, there's typically not much you can do at that point except crash.

There is a chance that this checker could report a significant number of false positives. In many cases, we have extra information so we know that a particular pointer is not going to be null. In these cases it's best to just remove the check entirely – or if it is useful to check – just move the check before the dereference.

Alternatively, there are options to suppress events with certain checkers by annotating your source code.



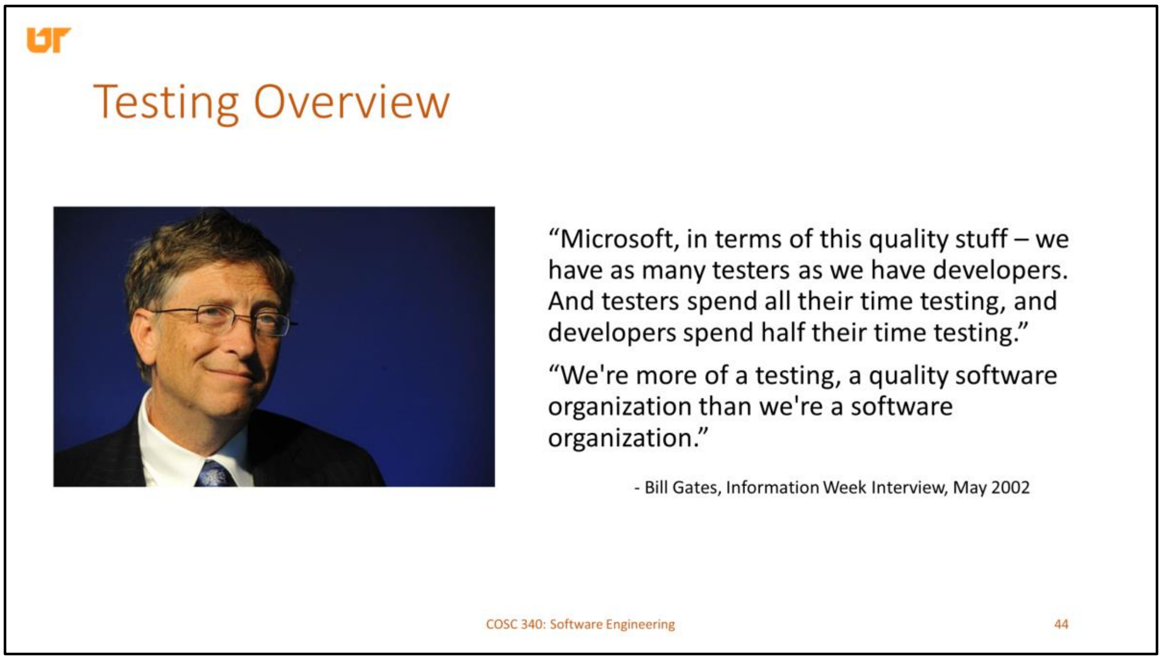
Coverity Checker: REVERSE_INULL

Checks for null checks after dereferences (C++, C#, Java)

- **C++ Example**

```
void foo(struct buf_t *request_buf) {
        …
        *request_buf = some_function()
        if (request_buf == NULL) return
        …
}
```

- **What if some_function() returns NULL?**



Testing Overview

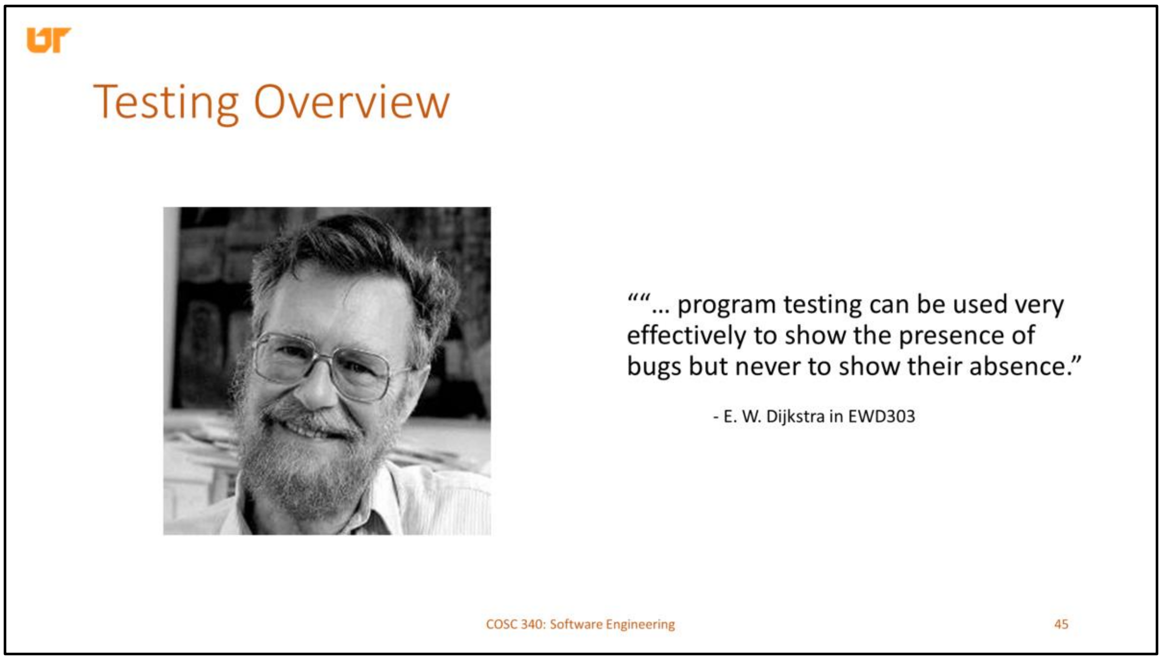
"Microsoft, in terms of this quality stuff – we have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing."

"We're more of a testing, a quality software organization than we're a software organization."

- Bill Gates, Information Week Interview, May 2002

COSC 340: Software Engineering 44

Microsoft has always prided itself on its testing practices.



Testing Overview

""... program testing can be used very effectively to show the presence of bugs but never to show their absence."

- E. W. Dijkstra in EWD303

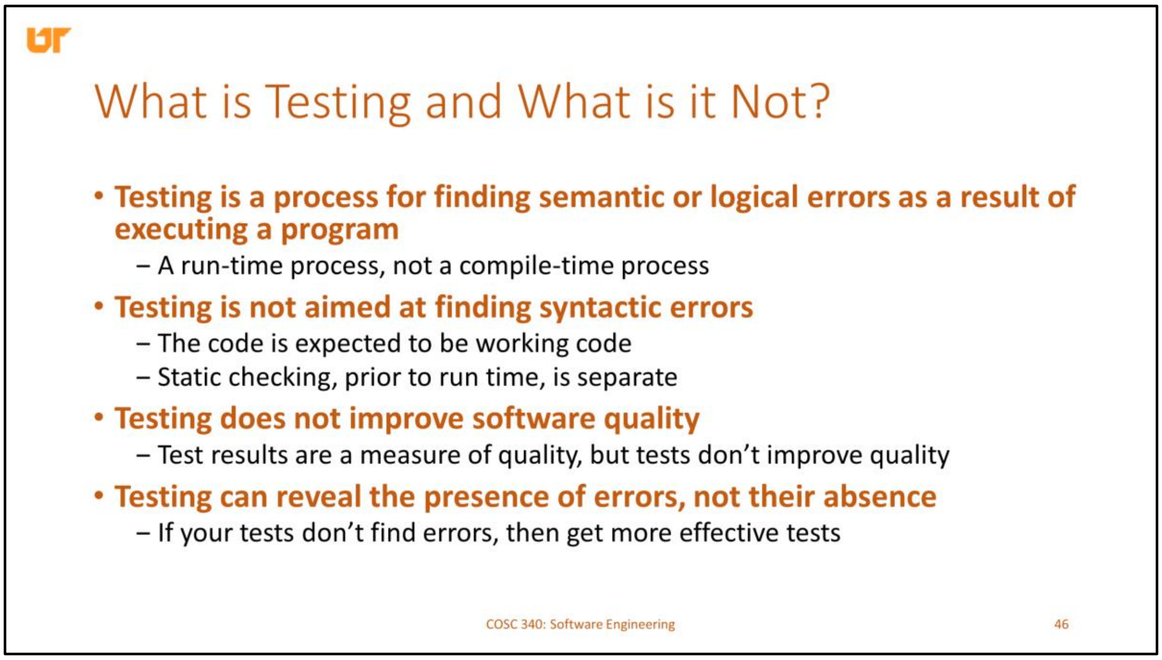
COSC 340: Software Engineering 45

When faced with a mechanism --be it hardware or software-- one can ask oneself "How can I convince myself of its being correct?" As long as we regard the mechanism as a black box, the only thing we can do is subject it to all possible inputs and check whether it produces the correct outputs. But for the kind of mechanisms we are considering this is absolutely out of the question.

I have a pet example to demonstrate this. At my University we have a machine and one should for instance like to know, whether the fixed point multiplication instruction works properly. The machine has a rather short word length of 27 bits, as a result there are only 2^{54} different fixed point multiplications possible. So, why not try them all? With 2^{14} multiplications per second, 2^{54} multiplications = 2^{40} sec = 10^{12} sec. = 10^{7} days = 30.000 years! It takes 30.000 years to have all possible multiplications performed just once.

One of the consequences of this number is that in the whole life time of the machine, the number of fixed point multiplications actually performed by our machine is a truly negligible fraction of the set of possible multiplications. From a simple-minded point of view we are only interested in the correct execution of the tiny set of multiplication the machine is actually called to perform. But because in programming we think not in terms of numerical values but in terms of variables, we have abstracted from the values actually processed by the arithmetic unit and we are only allowed to make this abstraction when the multiplier would do *any* multiplication correctly.

I make this point because it is often not realized that the at first sight extreme and ridiculous reliability requirements imposed by us on the hardware is a direct consequence of the fact that without it we could not afford this vital abstraction. Another consequence of the number of 30.000 years that sampling testing is hopelessly inadequate to convince ourselves of the correctness even of a simple piece of equipment as a multiplier: whole classes of in some sense critical cases can and will be missed! All this applies a fortiori to programs that claim to cope with many more cases and take more time for each single execution. The first moral of the story is that program testing can be used very effectively to show the presence of bugs but never to show their absence.



What is Testing and What is it Not?

- **Testing is a process for finding semantic or logical errors as a result of executing a program**
  - A run-time process, not a compile-time process
- **Testing is not aimed at finding syntactic errors**
  - The code is expected to be working code
  - Static checking, prior to run time, is separate
- **Testing does not improve software quality**
  - Test results are a measure of quality, but tests don't improve quality
- **Testing can reveal the presence of errors, not their absence**
  - If your tests don't find errors, then get more effective tests

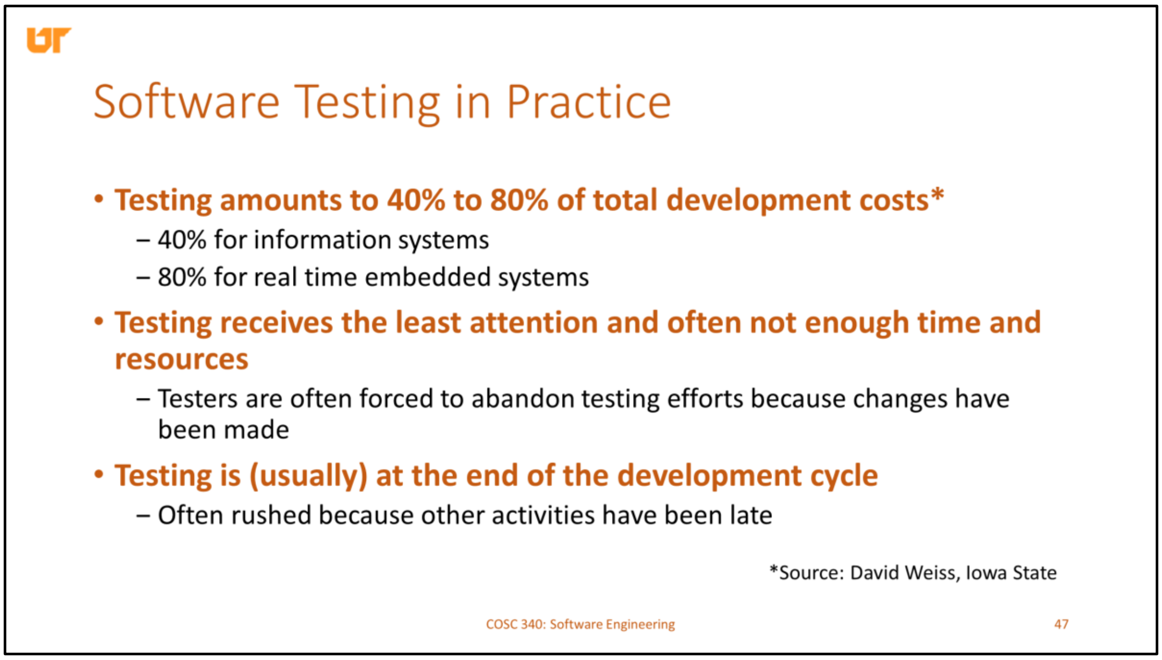
COSC 340: Software Engineering 46

Testing is a process for finding errors (semantic or logical) during execution.

We test code that we have already found to be syntactically correct – using a compiler. Static checking is performed before hand.

Testing does not improve quality. We can use tests to measure quality. We can find errors with testing – and it might lead us to eliminate some errors – but testing alone does not improve quality.

If you don't find errors, that does not mean your code works! You likely need more effective tests.


Software Testing in Practice

- **Testing amounts to 40% to 80% of total development costs***
  - 40% for information systems
  - 80% for real time embedded systems
- **Testing receives the least attention and often not enough time and resources**
  - Testers are often forced to abandon testing efforts because changes have been made
- **Testing is (usually) at the end of the development cycle**
  - Often rushed because other activities have been late

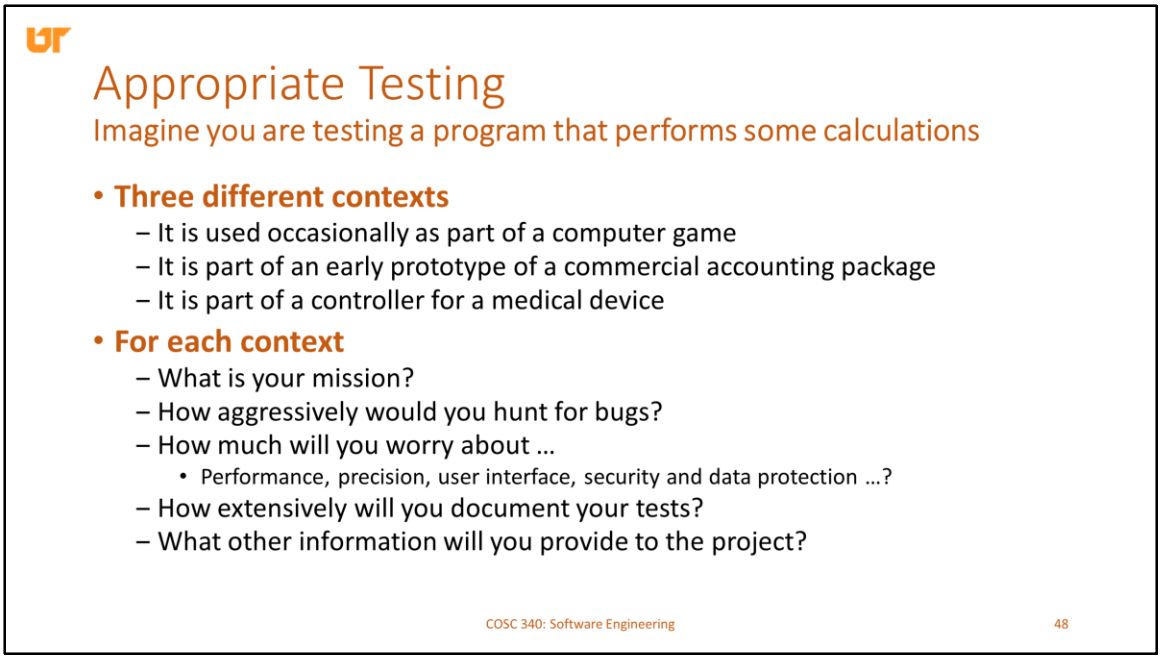
*Source: David Weiss, Iowa State

COSC 340: Software Engineering 47

You should expect testing to absorb a significant portion – if not the majority – of your development costs. Studies estimate testing requires 40% of development costs for information systems, in general, and 80% of costs for real time embedded systems (because you have to test the software for additional constraints, which require additional testing).

However, as you might expect, testing receives the least amount of attention during software development and often does not receive enough time or resources – which can result in the release of a defective product. The people developing the software are often responsible for testing the software as well – and they might have to abandon their testing efforts when something on the project changes.

One of the main reasons for these problems is because, naturally, testing is typically at the end of the development cycle. Since it occurs at the end, people often rush their testing efforts as other activities absorb their time of the project.



Appropriate Testing

Imagine you are testing a program that performs some calculations

- **Three different contexts**
  - It is used occasionally as part of a computer game
  - It is part of an early prototype of a commercial accounting package
  - It is part of a controller for a medical device
- **For each context**
  - What is your mission?
  - How aggressively would you hunt for bugs?
  - How much will you worry about …
    - Performance, precision, user interface, security and data protection …?
  - How extensively will you document your tests?
  - What other information will you provide to the project?

COSC 340: Software Engineering 48

One question you should always ask yourself for any software project you work on, is ‘what is the appropriate amount of testing that we should do for this project?’

The answer depends on the aspects of your individual project and the contexts in which your software will be used.

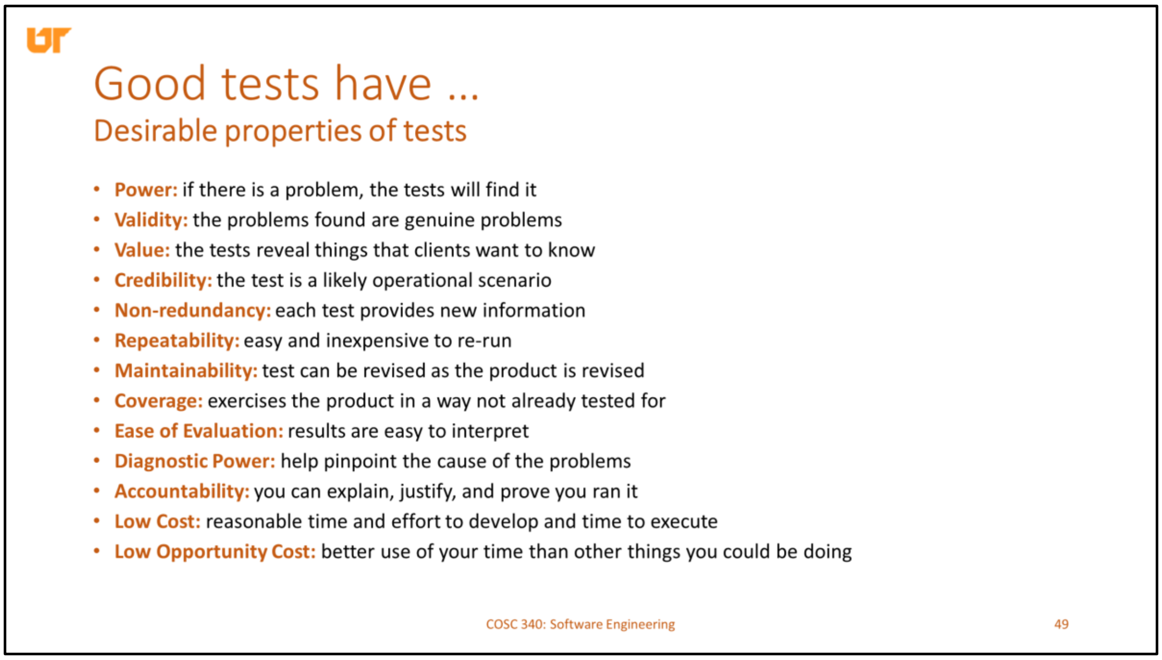
For example, consider you are testing a program that performs some mathematical calculations. In different contexts, you might test this program very differently.

For instance, if you’re developing the software to be used as part of a computer game – maybe you don’t care as much about accuracy – but performance is very important. So, you would write tests that make sure your software is accurate enough – but also ensure that you don’t drop under a specified frame rate.

If you’re developing the software as part of a prototype, you probably don’t care as much about performance, but you might have other concerns, such as user interface and functionality.

In other contexts, such as software for a medical device, or some other mission critical device, accuracy and reliability may be extremely important, so you would need to test the software very carefully to ensure these properties.

For each context, you would need to consider different questions, such as “what is your mission?”, “how buggy will you allow your software to be?” “how much will you worry about different properties – such as performance, precision, UI, and security”. And you should also consider how much information will you record during the testing process. Do test results need to be diligently recorded? Or can you simply record whether or not the tests passed?


Good tests have …

Desirable properties of tests

- **Power:** if there is a problem, the tests will find it
- **Validity:** the problems found are genuine problems
- **Value:** the tests reveal things that clients want to know
- **Credibility:** the test is a likely operational scenario
- **Non-redundancy:** each test provides new information
- **Repeatability:** easy and inexpensive to re-run
- **Maintainability:** test can be revised as the product is revised
- **Coverage:** exercises the product in a way not already tested for
- **Ease of Evaluation:** results are easy to interpret
- **Diagnostic Power:** help pinpoint the cause of the problems
- **Accountability:** you can explain, justify, and prove you ran it
- **Low Cost:** reasonable time and effort to develop and time to execute
- **Low Opportunity Cost:** better use of your time than other things you could be doing

COSC 340: Software Engineering 49

As with other aspects, there are certain properties that are desirable from a testing POV.

This slide gives a partial list of some of these desirable properties.

The power of a set of tests refers to their ability to find problems in your code.
Validity is whether the problems found by the tests are actual problems – that is – the test does not report a lot of false positives.
Tests are valuable if they reveal things that you and your customers would want to know.
A test is credible if it mimics some scenario that is likely to be encountered in the field.

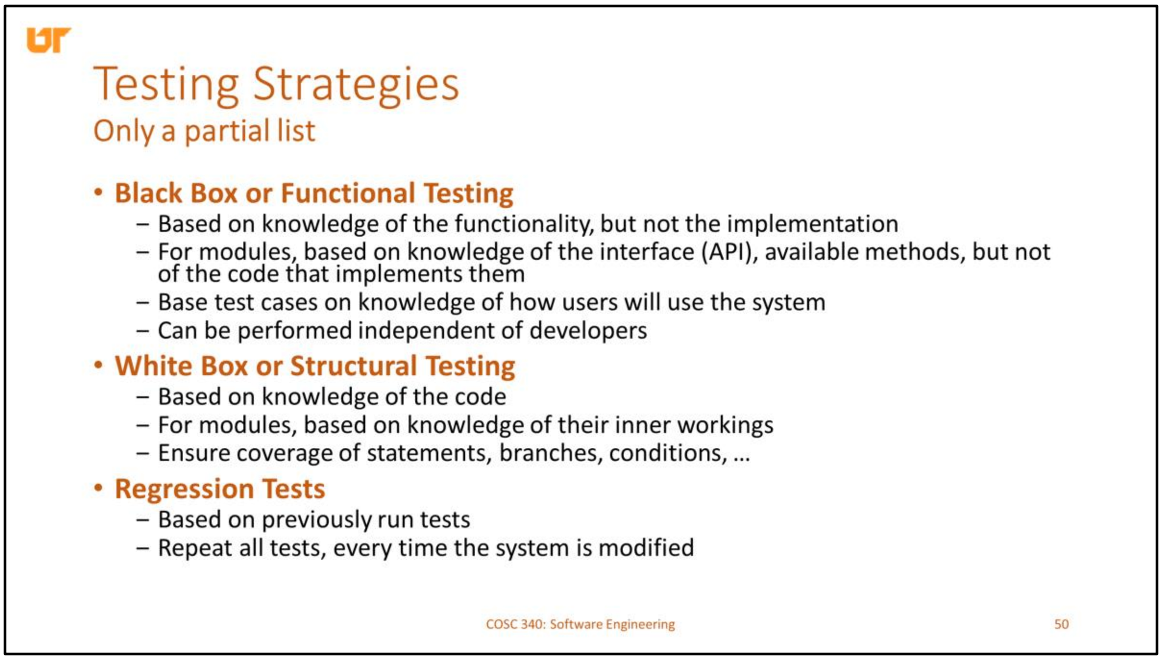
Some other important properties are test independence (or non-redundancy in your tests) as well as repeatability and maintainability of your tests.

A test or set of tests have good coverage if they exercise the product in all the different ways it might be used in practice.

It’s also important to structure your tests so that their results are easy to collect, explain, and interpret.

And, ideally, good tests will have all these properties, but also require relatively low development effort, execution time, and opportunity cost.

Obviously, writing tests with all of these properties might be difficult or infeasible depending on your project, but you should evaluate your tests based on these properties.


Testing Strategies
Only a partial list

- **Black Box or Functional Testing**
  - Based on knowledge of the functionality, but not the implementation
  - For modules, based on knowledge of the interface (API), available methods, but not of the code that implements them
  - Base test cases on knowledge of how users will use the system
  - Can be performed independent of developers
- **White Box or Structural Testing**
  - Based on knowledge of the code
  - For modules, based on knowledge of their inner workings
  - Ensure coverage of statements, branches, conditions, …
- **Regression Tests**
  - Based on previously run tests
  - Repeat all tests, every time the system is modified

COSC 340: Software Engineering 50

There are a wide range of strategies you can use to test your software. This slides shows few important classes of testing strategies.

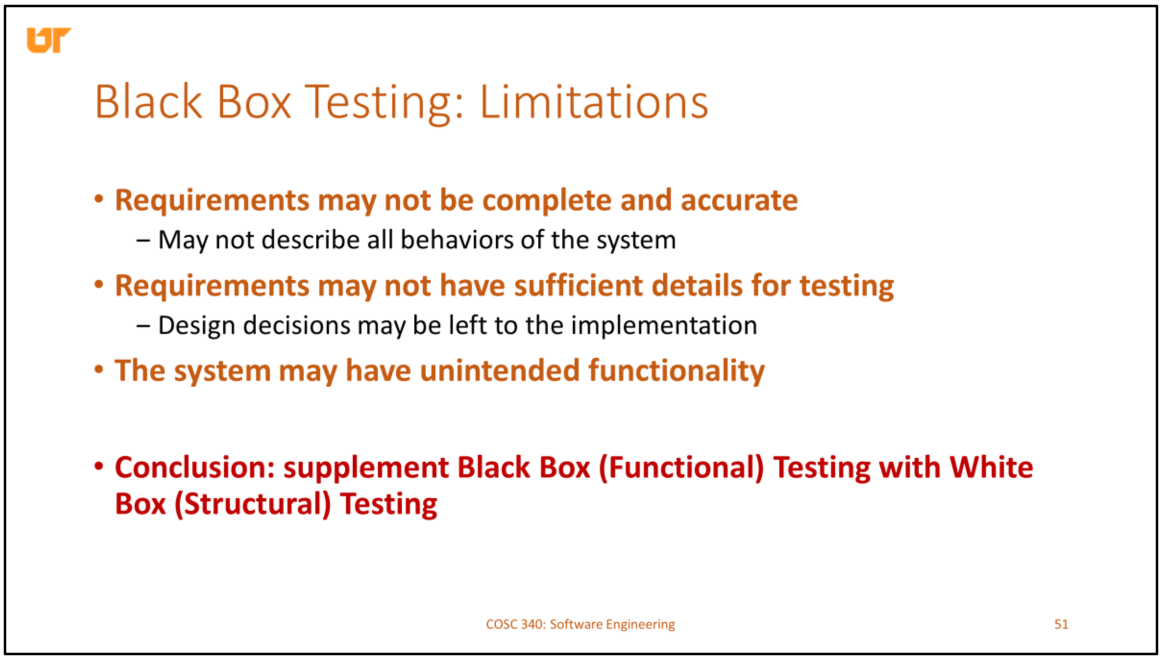
The first is black box or functional testing. Black Box testing refers to testing the functionality of a system, but not it's implementation. That is, you treat the implementation as a black box as you test the software's functionality.
If you are testing individual modules, you could test using only knowledge of their interface, but not the code that implements them.

For black box testing, you base your tests on your requirements and how you expect users will use the software. This sort of testing can often help expose discrepancies in your requirements or functional specifications. It also has the added advantage that tests can be done independently of the software developer – perhaps by another employee or team working in parallel.

Alternatively, we can also conduct white box or structural testing of the code. This just means you write tests for your product with knowledge of or in consideration of its implementation. So, for individual modules you can write tests considering the actual inner workings of the module (what might be in the .c or .cpp file and not just the .h file).

One nice thing about white box testing is that – since you have access to source code – you can ensure better coverage of statements in your code and make sure you have tests that test each branch or condition in your code.

Regression testing is another form of testing that can include either black box or white box tests. Regression testing just means you repeat all of your tests every time you update or modify the system. So, when you add a new feature, you not only run a test to evaluate the new feature, but you also re-run all of your old tests to make sure none of them broke. In this way, you can ensure that you can continuously deliver working code with each new feature.


Black Box Testing: Limitations

- **Requirements may not be complete and accurate**
  - May not describe all behaviors of the system
- **Requirements may not have sufficient details for testing**
  - Design decisions may be left to the implementation
- **The system may have unintended functionality**

- **Conclusion: supplement Black Box (Functional) Testing with White Box (Structural) Testing**

COSC 340: Software Engineering 51

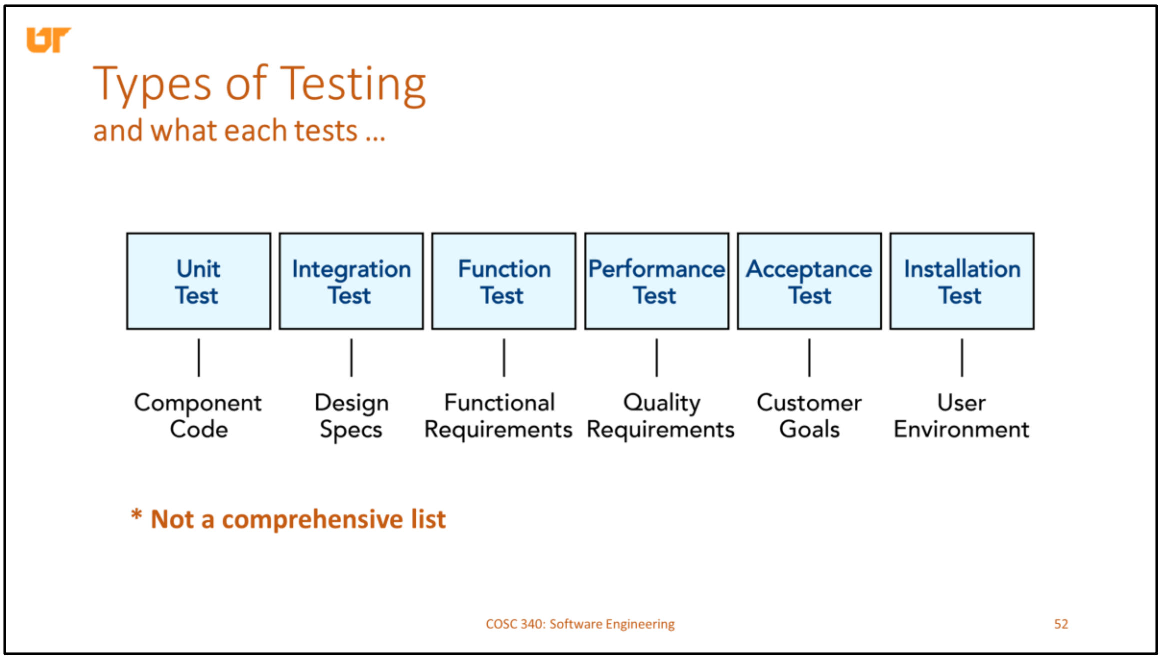
Black Box testing is beneficial because it allows you to test the product from the POV of the customer. However, there are a number of limitations to strictly black box testing your product.

For instance, black box tests are typically based off the requirements you and your customers have agreed upon for the system.

However, these requirements may be incomplete and may not actually describe all the functionality you should test. This can make writing black box tests that actually test all the different ways your customers might use your software very difficult.

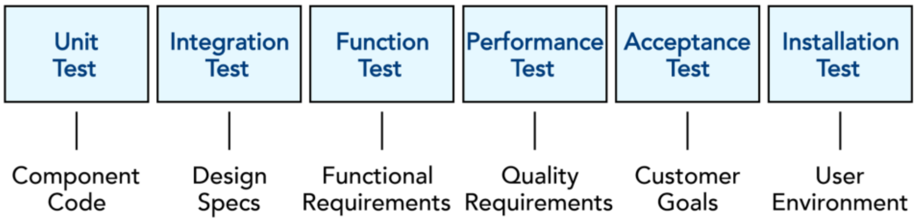
Additionally, various design decisions might be left to the implementation. If you're unaware of these decisions, you might miss some important test cases. For instance, maybe you're developing a calculator – and you know that in most cases – the user will only want to calculate with relatively small integer values. So, you use a 4-byte 'integer' type to represent numbers in the range -2^31 to 2^31-1. However, if the user inputs a very large (or very negative) integer outside this range, then you have a different system for representing arbitrarily large integers. If you're not aware of this design decision, you might not test integers outside the small integer range.

To address these limitations, it is recommended that you supplement black box testing with white box testing so that you can evaluate your product from the user's point of view – but also in consideration of all the design decisions hidden in your implementation.



Types of Testing

and what each tests …



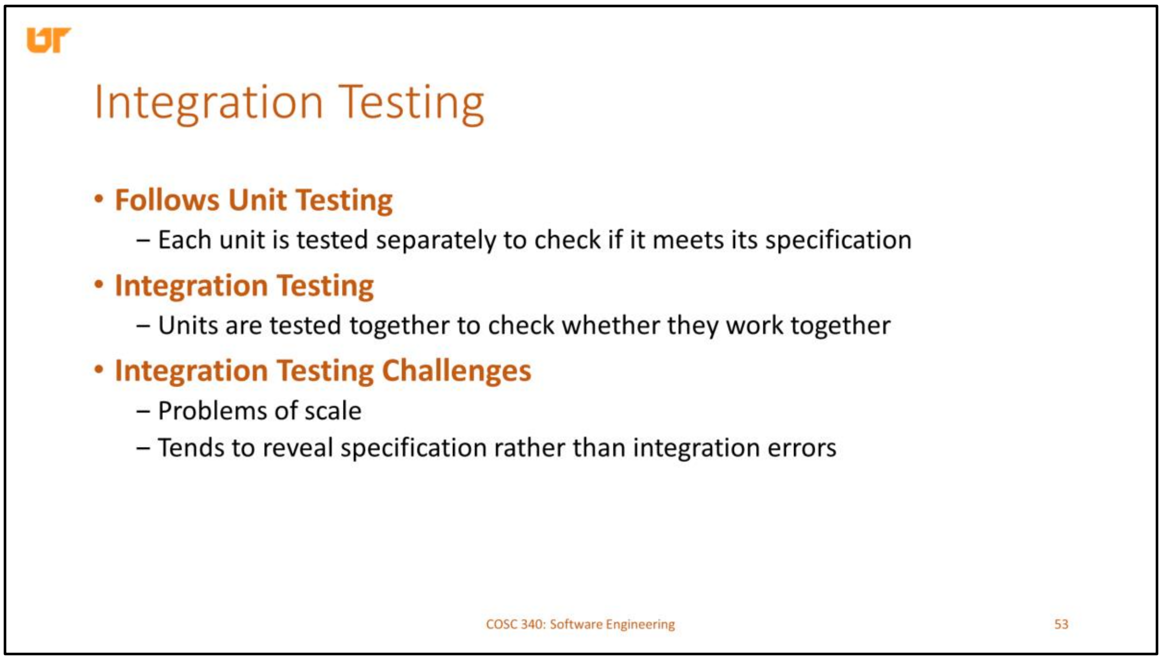
*** Not a comprehensive list**

COSC 340: Software Engineering 52

This figure shows a number of different types of testing and what each test will evaluate. We have seen some of these tests earlier when we discussed V-Processes.

The different types of tests often correspond to different stages of planning or development.

You need to conduct explicit tests to evaluate the implementation, design, functionality of your system, and whether or not it meets the customer's needs. Simply testing the implementation of individual methods in your code will not ensure that customer's will be satisfied with your product.



Integration Testing

- **Follows Unit Testing**
  - Each unit is tested separately to check if it meets its specification
- **Integration Testing**
  - Units are tested together to check whether they work together
- **Integration Testing Challenges**
  - Problems of scale
  - Tends to reveal specification rather than integration errors

COSC 340: Software Engineering 53

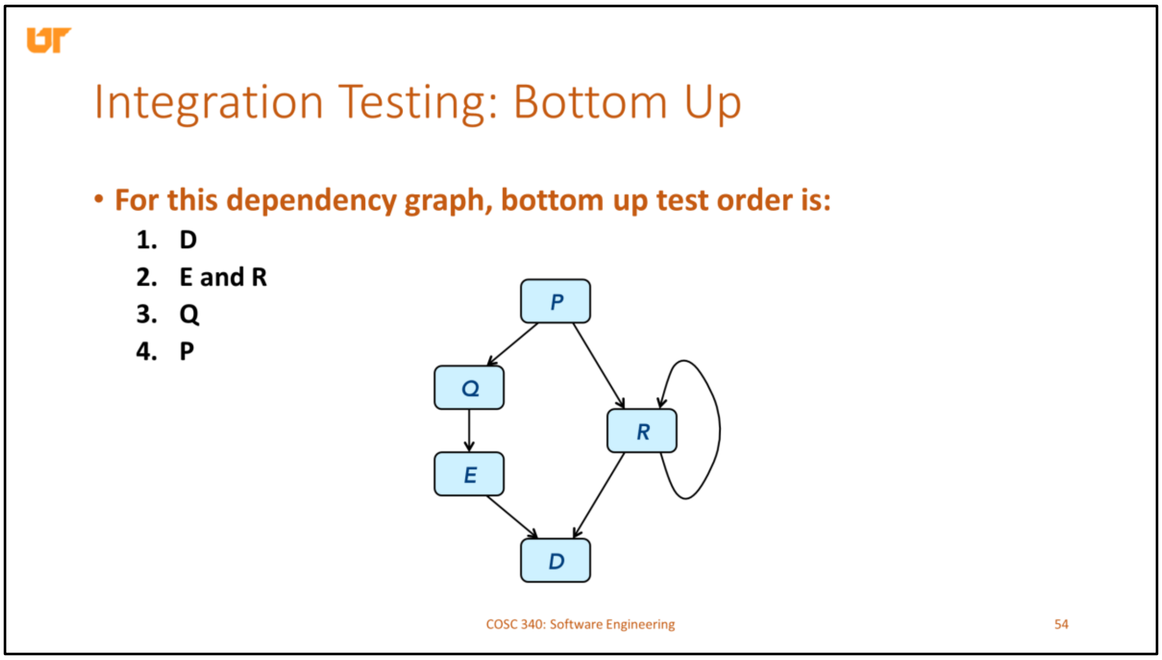
Let’s discuss a little more about integration testing.

You typically do unit testing first and follow it with integration testing. Unit testing tests the individual functions and modules in your code separately.

Once you are sure the individual modules or working properly, you test them together to ensure they work together properly. This is the idea behind integration testing.

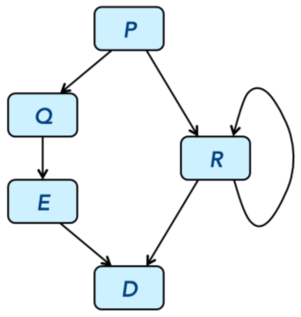
Writing good integration tests can be challenging – especially for large software projects that have many different components that might interact. You often cannot test all interactions – so you have to choose a representative subset.

Also, integration testing tends to reveal design errors rather than integration errors. It is at this stage where you may realize that two components that you thought could work together might not work together – or you may realize you need additional methods or modules to accomplish certain tasks.



Integration Testing: Bottom Up

- **For this dependency graph, bottom up test order is:**
  1. **D**
  2. **E and R**
  3. **Q**
  4. **P**

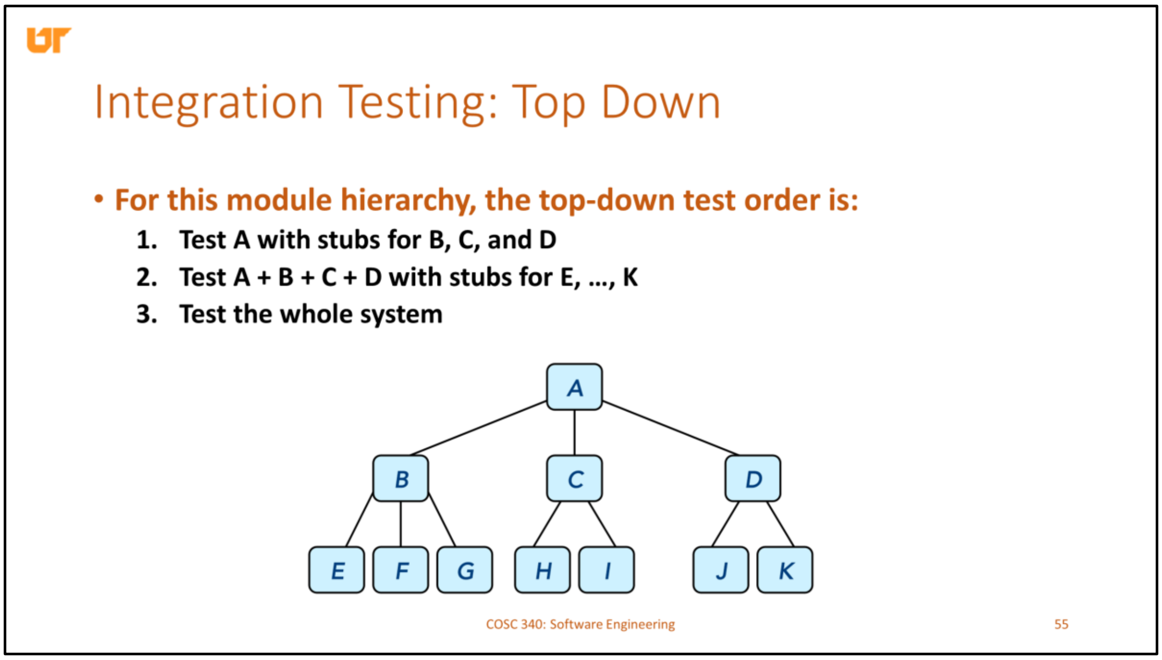


There are two main types of integration testing: bottom up and top down (also others – but would be variations of these).

In bottom-up integration testing, you test all the bottom-level units in your code that do not depend upon or call other parts of the code you test the higher-level units.

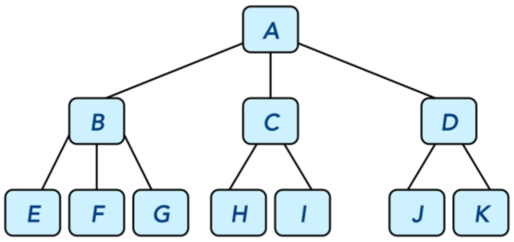
So, for instance, in this dependency graph, P might call or depend on Q and R, Q depends on E, and E and R both depend on D. So, first we would test D. When we know D works, we test both E and R (with D). Then, we can test Q (with D and E). Then, finally, we can test P knowing that Q, E, R, and D have been tested and should work at this point.

For bottom-up testing, you need some sort of driver to simulate the higher-level units during your testing of units at the bottom of the dependency graph.



Integration Testing: Top Down

- For this module hierarchy, the top-down test order is:
  1. Test A with stubs for B, C, and D
  2. Test A + B + C + D with stubs for E, …, K
  3. Test the whole system



Top-down integration testing is just the opposite of bottom-up.

With a top-down strategy, we test the units at the top of the hierarchy first, and then proceed to test the lower level units step-by-step afterwards.

With this approach, you need stubs to simulate the operation of the lower-level units in the dependency graph.

In this example, we would test A first and use stubs for units B, C, and D as we test A.

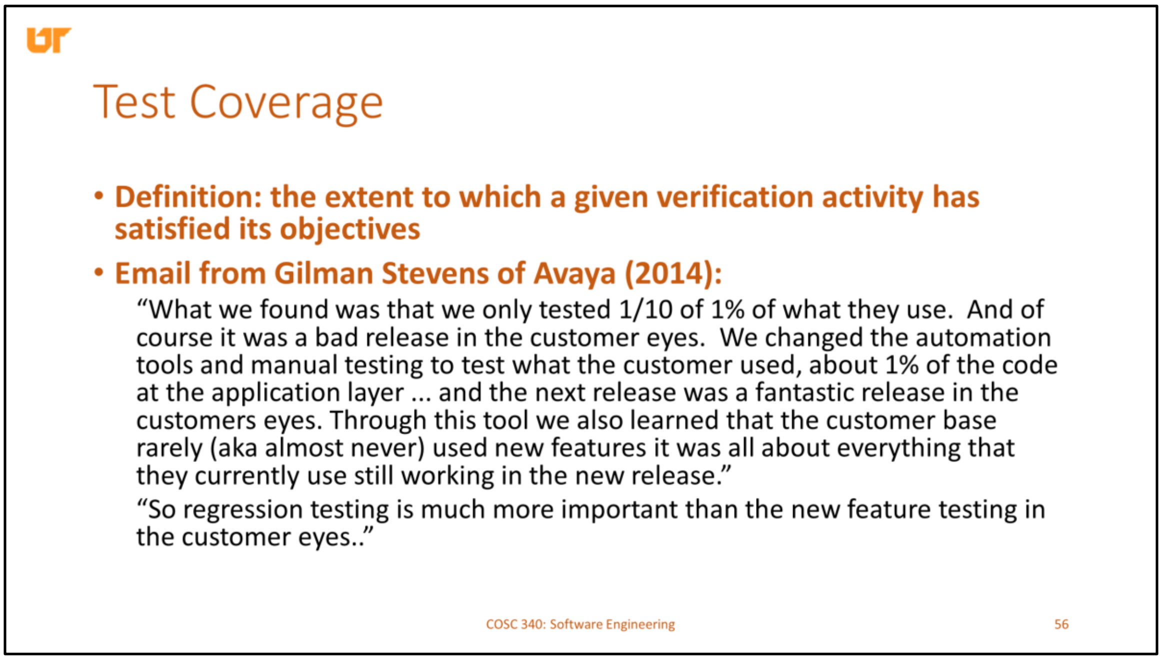
Then, when we know A is working, we can move to the next level and test A + B + C + D, and use stubs for the lowest level.

Finally, we test the whole system together.

Add a slide here:

After integration testing, comes system testing, where you test functional requirements, and after system testing, you could do acceptance testing to test whether or not the system meets the users requirements for acceptance. You can think of these tests in a hierarchy from most fine-grained and specific to implementation, to more abstract up to customer requirements.

Acceptance Tests
^
^
System Tests
^
^
Integration Tests
^
^
Unit Tests



Test Coverage

- **Definition: the extent to which a given verification activity has satisfied its objectives**
- **Email from Gilman Stevens of Avaya (2014):**
  "What we found was that we only tested 1/10 of 1% of what they use. And of course it was a bad release in the customer eyes. We changed the automation tools and manual testing to test what the customer used, about 1% of the code at the application layer ... and the next release was a fantastic release in the customers eyes. Through this tool we also learned that the customer base rarely (aka almost never) used new features it was all about everything that they currently use still working in the new release."
  "So regression testing is much more important than the new feature testing in the customer eyes.."

COSC 340: Software Engineering 56

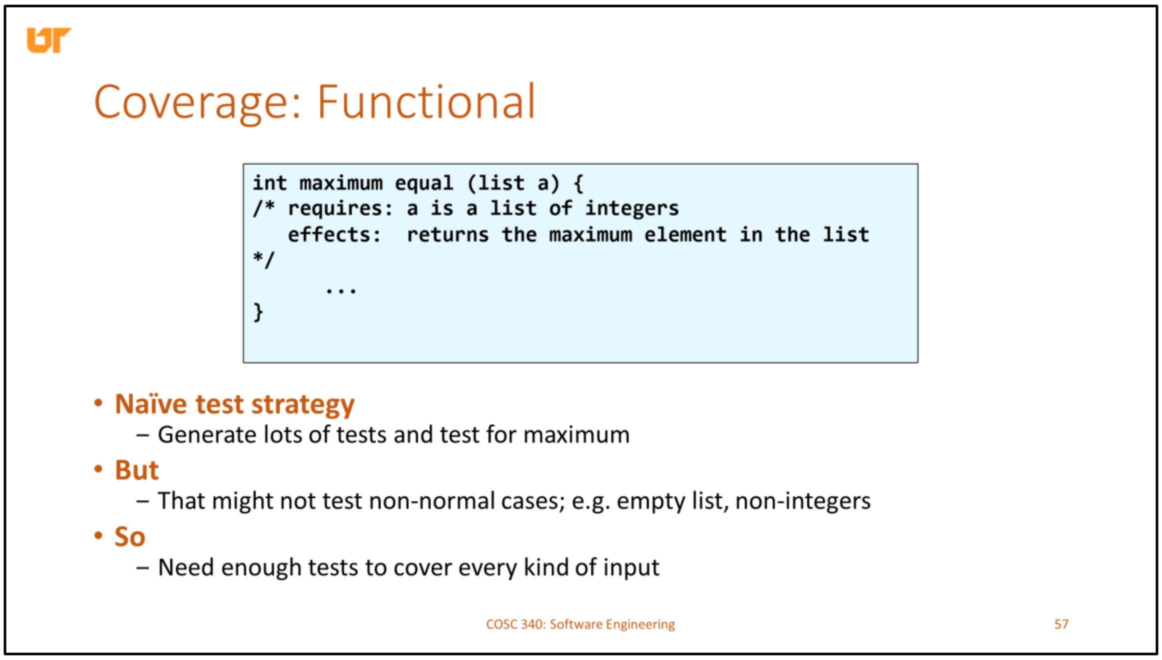
Coverage is the extent to which a given verification activity has satisfied its objectives.

Most often when we discuss coverage, we are talking about what percentage of the code or functionality a particular set of tests is able to evaluate.

The importance of coverage is illustrated by my friend Gilman Stevens, who had helped manage the development of communications software at Nortel and Avaya. He writes in an email about how, when he was at Nortel, they had used a code coverage tool in the field to evaluate how much of their software was actually being tested at the factory.

He found that only about 1/10 of 1% was being tested before it was released. And the customers were not satisfied with those releases. But, after they changed their testing tools based off the feedback from this study – they were able to test about 1% of all the code the customer's were using in the field. This 10x increase in coverage yielded much higher quality in the customer's eyes.

Using this coverage tool, they also found that their new code was almost never the cause for complaints. Rather, the new features could have broken some existing features the customer really cared about – which made them upset. So, in conclusion, regression testing was more important than actually having the new feature according to their customers.



Coverage: Functional

```
int maximum equal (list a) {
/* requires: a is a list of integers
   effects:  returns the maximum element in the list
*/
      ...
}
```

- **Naïve test strategy**
  - Generate lots of tests and test for maximum
- **But**
  - That might not test non-normal cases; e.g. empty list, non-integers
- **So**
  - Need enough tests to cover every kind of input

COSC 340: Software Engineering 57

Let’s first discuss the concept of coverage in terms of functional testing.

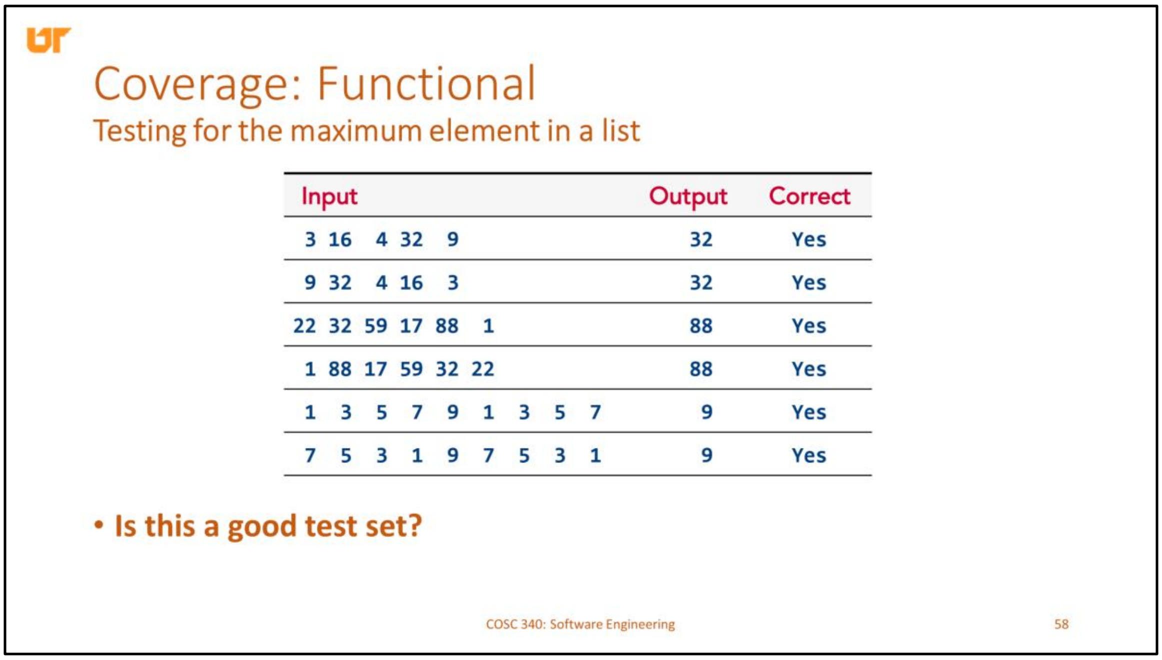
Remember, with functional testing, we want to test whether a particular piece of software performs the correct function for each possible input value.

So, suppose we are writing a test to test a function that takes a list of integers and returns the maximum element in the list.

To ensure good test coverage, we might say, well, let’s generate many different random lists of integers and compute their maximums. Then, we can manually evaluate if we got the right answer each time.

In addition to being time-consuming, that might not be the best strategy. Often times, in software, the tests where your code will break are those cases that are the typical input case. For instance, with this problem, we might want to test for empty lists or lists with non-integers to ensure the function behaves appropriately.

So, we need to make sure we include those other types of inputs that aren’t normal in our test set.



Coverage: Functional

Testing for the maximum element in a list

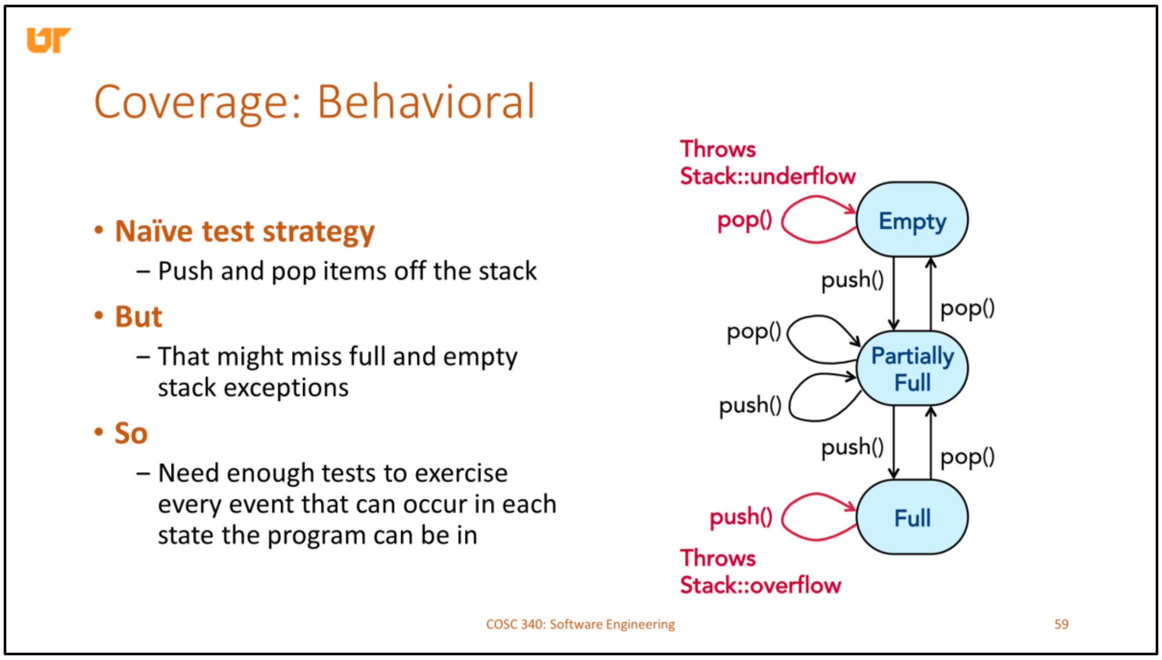
Input	Output	Correct
3 16 4 32 9	32	Yes
9 32 4 16 3	32	Yes
22 32 59 17 88 1	88	Yes
1 88 17 59 32 22	88	Yes
1 3 5 7 9 1 3 5 7	9	Yes
7 5 3 1 9 7 5 3 1	9	Yes

- **Is this a good test set?**

So, for instance, consider that we test the function with this set of lists. It seems like the function works every single time.

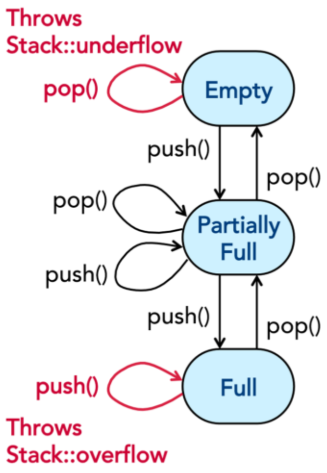
Would you consider this to be a good test set for this problem?

What other inputs would you give to this function to ensure good coverage of the method's functionalities?



Coverage: Behavioral

- **Naïve test strategy**
  - Push and pop items off the stack
- **But**
  - That might miss full and empty stack exceptions
- **So**
  - Need enough tests to exercise every event that can occur in each state the program can be in



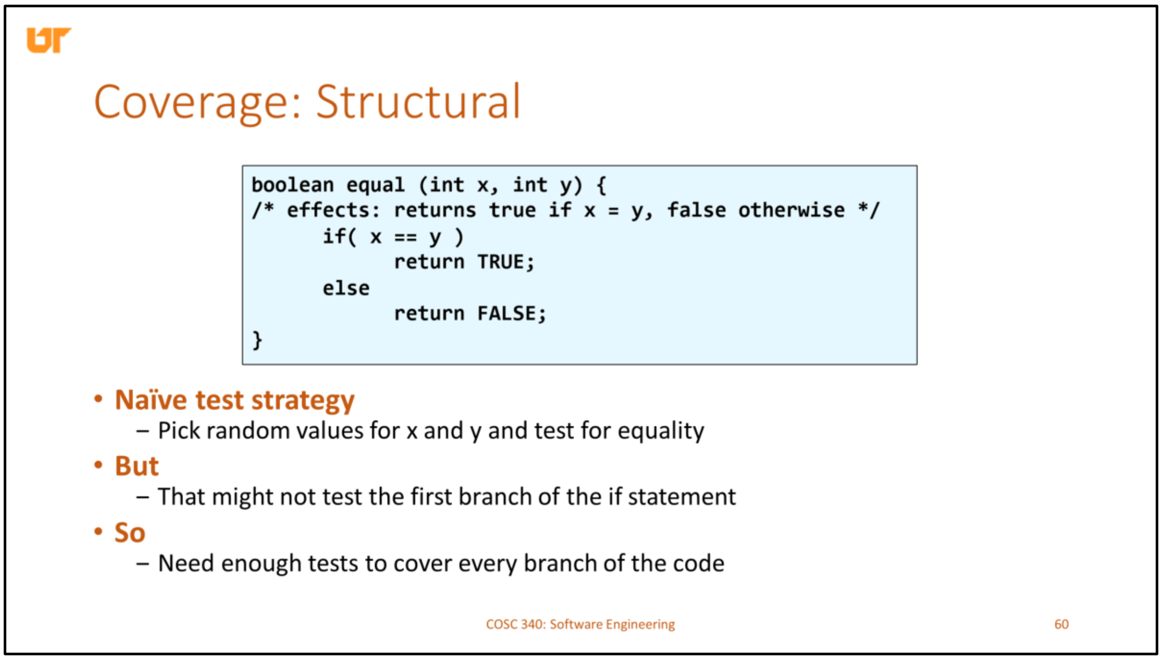
Coverage is also important in the context of behavioral testing. Behavioral tests aim to ensure the software behaves correctly given different inputs.

So, for instance, take the example of a program implementing a stack data structure.

To test this program, you might simply just try pushing and popping items off the stack randomly. Do that enough times and call it done.

The downside of this strategy is that you might miss the testing of some important states in your program. For instance, you want to make sure that you test error conditions, such as trying to insert when the stack is full and trying to remove when the stack is empty.

So, to ensure good coverage with your behavioral tests, you need to consider each state that your program can be in. In this example the stack might be in the partially full state, but it also might be in the empty state or in the full state. You need to write tests that evaluate each state of the program with different inputs.



Coverage: Structural

```
boolean equal (int x, int y) {
/* effects: returns true if x = y, false otherwise */
      if( x == y )
            return TRUE;
      else
            return FALSE;
}
```

- **Naïve test strategy**
  - Pick random values for x and y and test for equality
- **But**
  - That might not test the first branch of the if statement
- **So**
  - Need enough tests to cover every branch of the code

COSC 340: Software Engineering 60

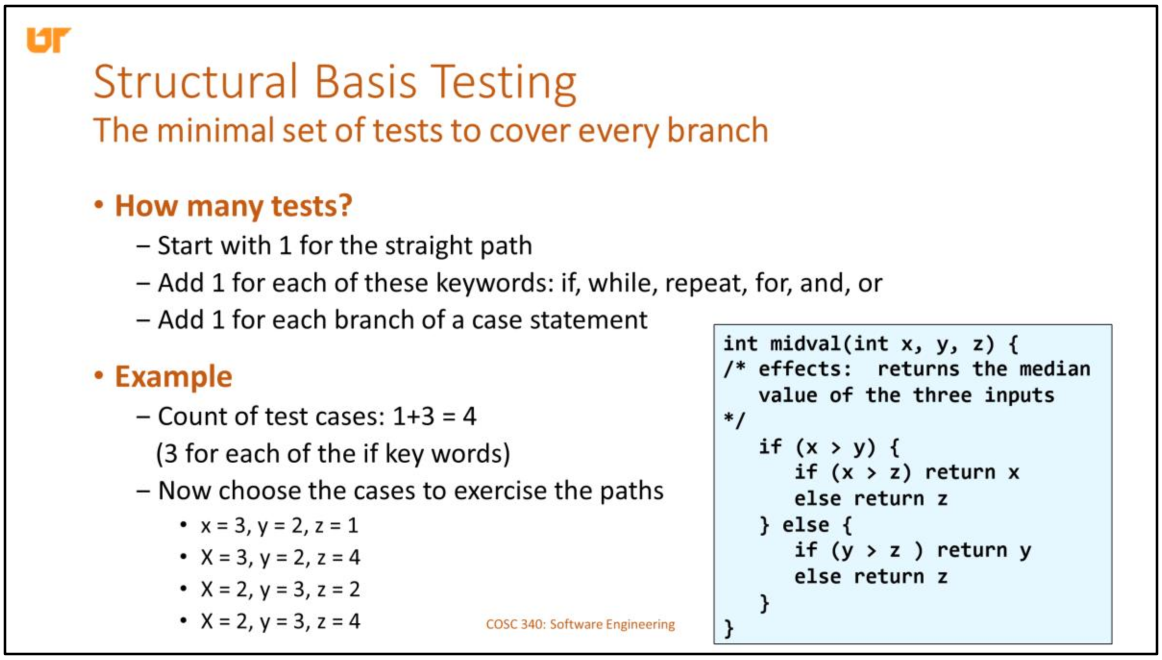
Coverage can also enable much more effective white box or structural testing where you test in consideration of the structure of the source code.

For instance, consider testing this equal function. (describe it)

A naïve way to test this code might be assign random values to x and y and ensure that it always returns the correct answer.

But – if you're just picking numbers randomly, there might be low probability that the numbers actually equal – so you might not test the first branch in the if statement. A better set of tests will cover every branch of the source code.

Ideally, you would have at least enough tests to cover every branch in your code.


Structural Basis Testing
The minimal set of tests to cover every branch

- **How many tests?**
  - Start with 1 for the straight path
  - Add 1 for each of these keywords: if, while, repeat, for, and, or
  - Add 1 for each branch of a case statement
- **Example**
  - Count of test cases: 1+3 = 4
    (3 for each of the if key words)
  - Now choose the cases to exercise the paths
    - x = 3, y = 2, z = 1
    - X = 3, y = 2, z = 4
    - X = 2, y = 3, z = 2
    - X = 2, y = 3, z = 4

```
int midval(int x, y, z) {
/* effects:  returns the median
   value of the three inputs
*/
   if (x > y) {
      if (x > z) return x
      else return z
   } else {
      if (y > z ) return y
      else return z
   }
}
```

COSC 340: Software Engineering

This is the idea behind structural basis testing.

Consider you wanted to generate tests to cover every single branch in your source code.

What is the minimum number of tests you would need?
1 for the straight path and add 1 for each branch

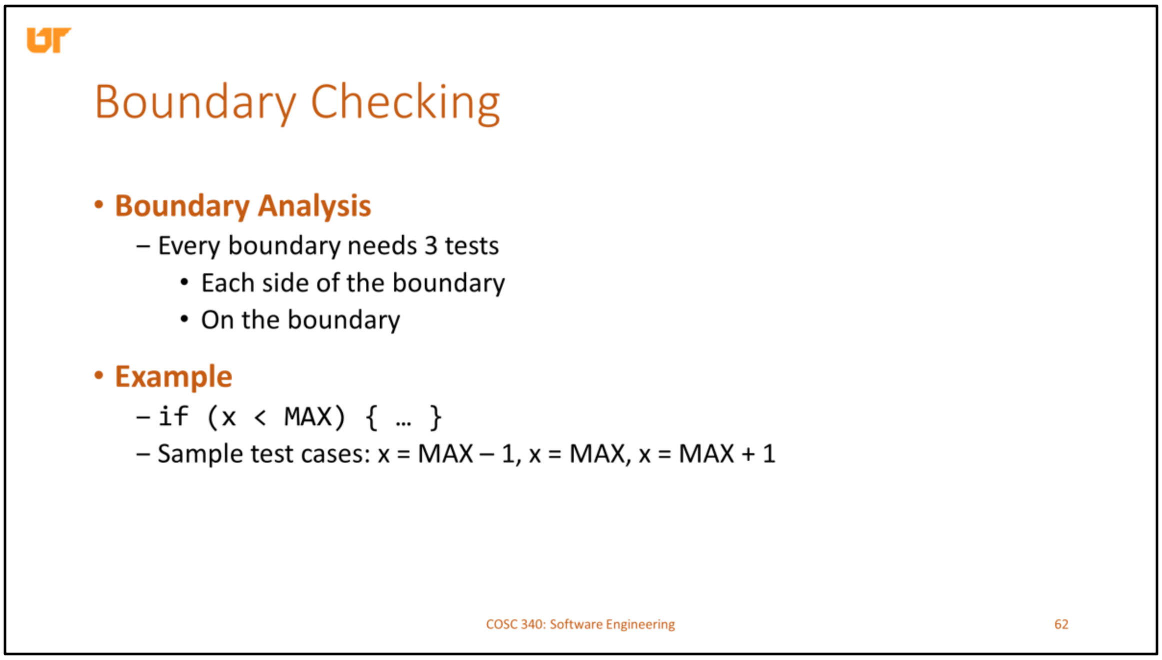
Consider the midval program (describe it)

We have three if statements – so the minimum number of tests is 1 + 3 = 4

Now choose the cases to exercise the paths?

Is this program correct?

We didn’t test for equal values – consider ‘3 3 2’ – depending on how median is defined – may not be right.



Boundary Checking

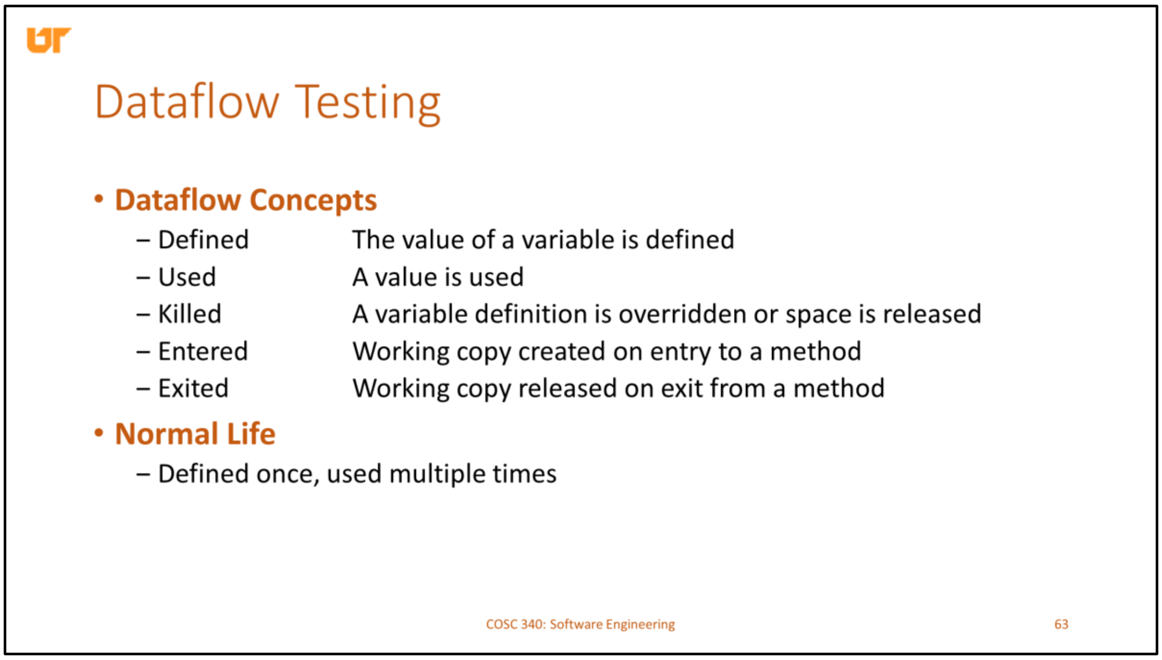
- **Boundary Analysis**
  - Every boundary needs 3 tests
    - Each side of the boundary
    - On the boundary
- **Example**
  - `if (x < MAX) { … }`
  - Sample test cases: x = MAX – 1, x = MAX, x = MAX + 1

COSC 340: Software Engineering 62

Relatedly, we have the concept of boundary analysis.

Here, for each boundary or check in your code, we generate tests for three cases:
1 for each side of the boundary
And 1 for if the input values are on the boundary.

Boundary checking would cover the case that inputs are equal.



Dataflow Testing

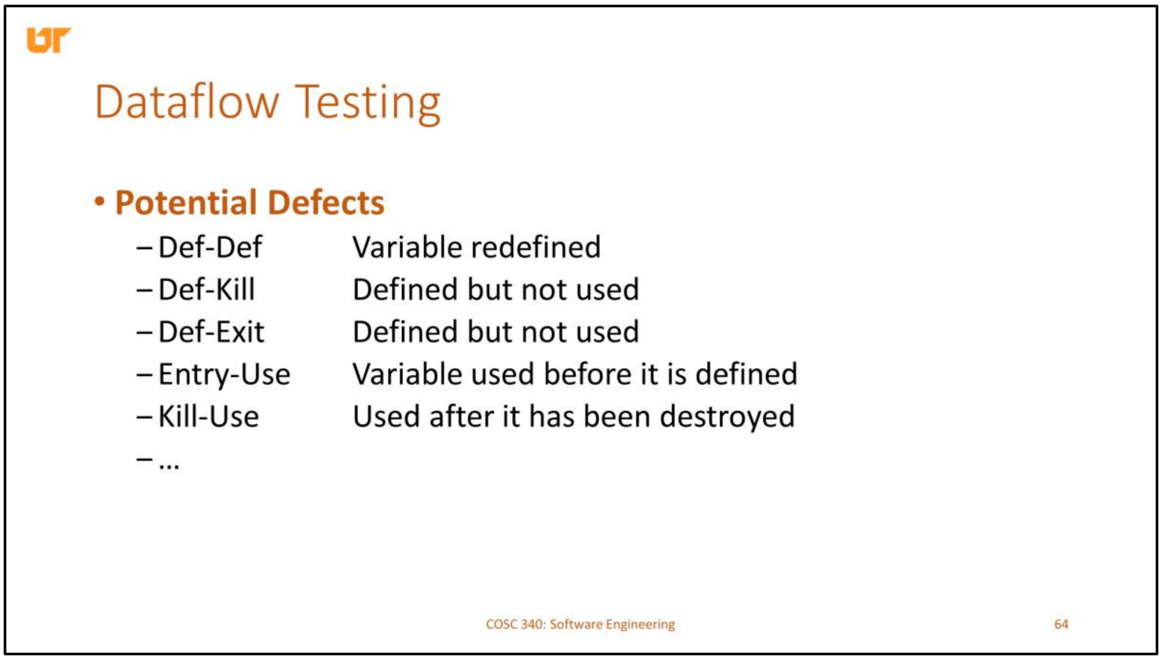
- **Dataflow Concepts**

– Defined	The value of a variable is defined
– Used	A value is used
– Killed	A variable definition is overridden or space is released
– Entered	Working copy created on entry to a method
– Exited	Working copy released on exit from a method

- **Normal Life**
  - Defined once, used multiple times

Next, we can also use program dataflow to generate tests with good coverage.

The idea here is to ensure that all program variables lead a normal life.

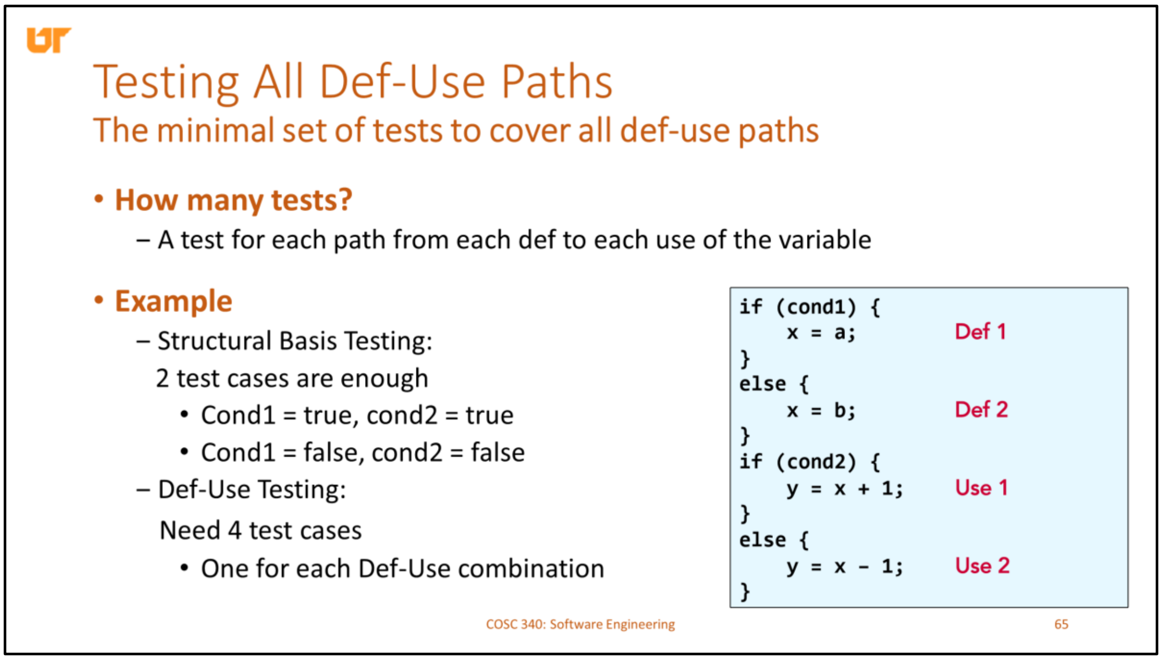


Dataflow Testing

- **Potential Defects**

– Def-Def	Variable redefined
– Def-Kill	Defined but not used
– Def-Exit	Defined but not used
– Entry-Use	Variable used before it is defined
– Kill-Use	Used after it has been destroyed
– …	

Some potential defects that would be found by data flow testing (read them).



Testing All Def-Use Paths

The minimal set of tests to cover all def-use paths

- **How many tests?**
  - A test for each path from each def to each use of the variable
- **Example**
  - Structural Basis Testing:
    2 test cases are enough
    - Cond1 = true, cond2 = true
    - Cond1 = false, cond2 = false
  - Def-Use Testing:
    Need 4 test cases
    - One for each Def-Use combination

```
if (cond1) {
    x = a;          Def 1
}
else {
    x = b;          Def 2
}
if (cond2) {
    y = x + 1;      Use 1
}
else {
    y = x - 1;      Use 2
}
```

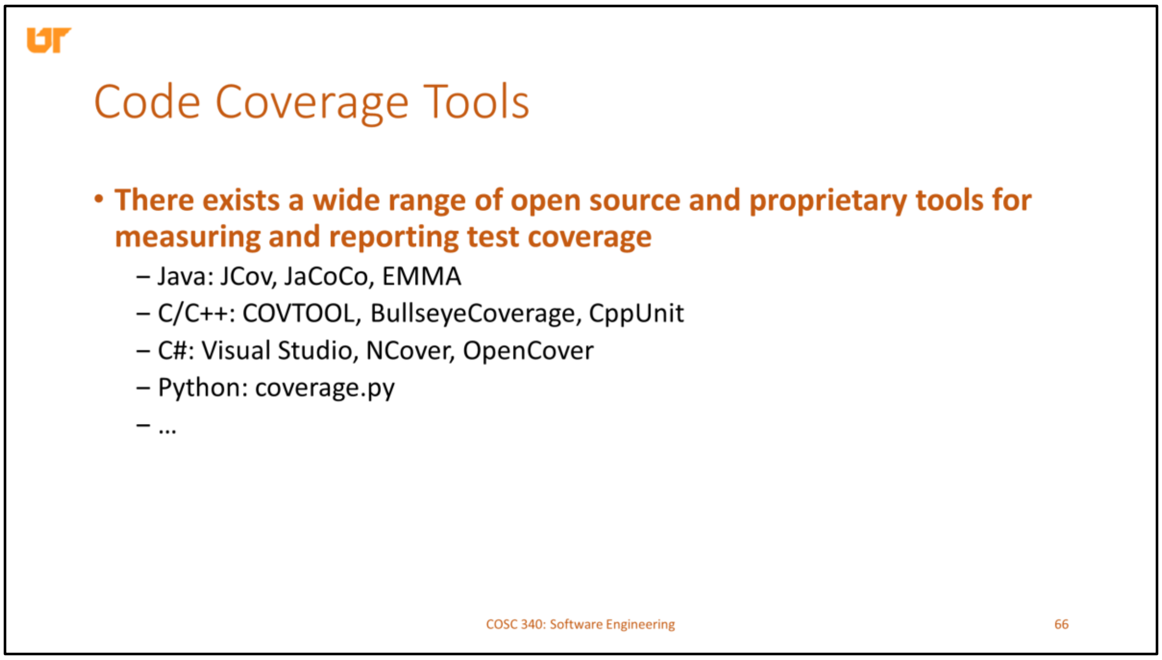
COSC 340: Software Engineering 65

To ensure good coverage, we would want to generate tests that cover all possible def-use paths for each program variable.

So, we generate tests that test each path from the definition of a variable to each use of that variable.

Consider this example.
Structured basis testing only requires two cases: (true, true) and (false, false)

Def-Use Testing requires four cases (true, true): def1-use1, (true, false): def1-use2, (false, true): def2-use1, and (false, false): def2-use2

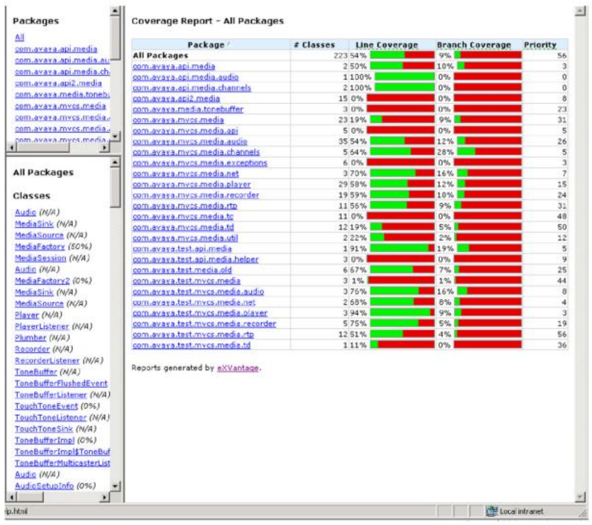


Code Coverage Tools

- **There exists a wide range of open source and proprietary tools for measuring and reporting test coverage**
  - Java: JCov, JaCoCo, EMMA
  - C/C++: COVTOOL, BullseyeCoverage, CppUnit
  - C#: Visual Studio, NCover, OpenCover
  - Python: coverage.py
  - …

There are a variety of open source and proprietary tools that can help you the effectiveness of your tests using coverage analysis.

Coverage Reports: Packages



Packages

All
com.avaya.api.media
com.avaya.api.media.au
com.avaya.api.media.ch
com.avaya.api2.media
com.avaya.media.toneb
com.avaya.mvcs.media
com.avaya.mvcs.media.
com.avaya.mvcs.media.
com.avaya.mvcs.media.

All Packages

Classes

Audio (N/A)
MediaSink (N/A)
MediaSource (N/A)
MediaFactory (50%)
MediaSession (N/A)
Audio (N/A)
MediaFactory2 (0%)
MediaSink (N/A)
MediaSource (N/A)
Player (N/A)
PlayerListener (N/A)
Plumber (N/A)
Recorder (N/A)
RecorderListener (N/A)
ToneBuffer (N/A)
ToneBufferFlushedEvent
ToneBufferListener (N/A)
TouchToneEvent (0%)
TouchToneListener (N/A)
TouchToneSink (N/A)
ToneBufferImpl (0%)
ToneBufferImpl$ToneBuf
ToneBufferMulticasterList
Audio (N/A)
AudioSetupInfo (0%)

Coverage Report - All Packages

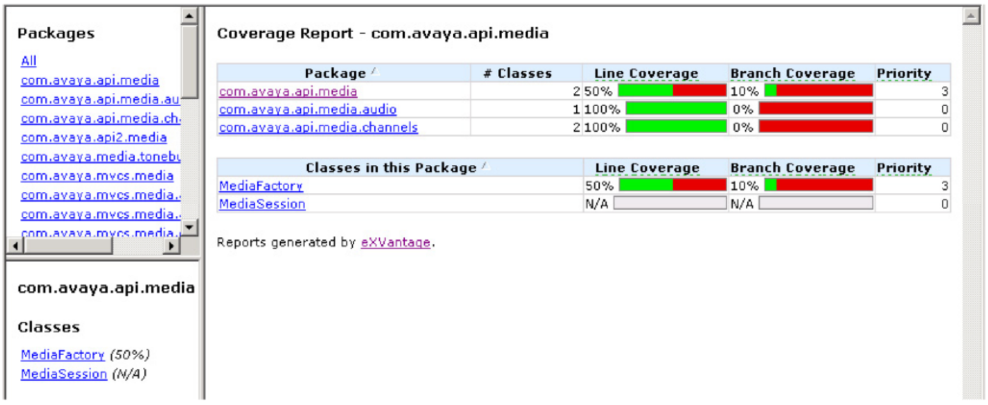
Package	# Classes	Line Coverage	Branch Coverage	Priority
All Packages	223	54%	9%	56
com.avaya.api.media	2	50%	10%	3
com.avaya.api.media.audio	1	100%	0%	0
com.avaya.api.media.channels	2	100%	0%	0
com.avaya.api2.media	15	0%	0%	8
com.avaya.media.tonebuffer	3	0%	0%	23
com.avaya.mvcs.media	23	19%	9%	31
com.avaya.mvcs.media.api	5	0%	0%	5
com.avaya.mvcs.media.audio	35	54%	12%	26
com.avaya.mvcs.media.channels	5	64%	28%	5
com.avaya.mvcs.media.exceptions	6	0%	0%	3
com.avaya.mvcs.media.net	3	70%	16%	7
com.avaya.mvcs.media.player	29	58%	12%	15
com.avaya.mvcs.media.recorder	19	59%	10%	24
com.avaya.mvcs.media.rtp	11	56%	9%	31
com.avaya.mvcs.media.tc	11	0%	0%	48
com.avaya.mvcs.media.td	12	19%	5%	50
com.avaya.mvcs.media.util	2	22%	2%	12
com.avaya.test.api.media	1	91%	19%	5
com.avaya.test.api.media.helper	3	0%	0%	9
com.avaya.test.media.old	6	67%	7%	25
com.avaya.test.mvcs.media	3	1%	1%	44
com.avaya.test.mvcs.media.audio	3	76%	16%	8
com.avaya.test.mvcs.media.net	2	68%	8%	4
com.avaya.test.mvcs.media.player	3	94%	9%	3
com.avaya.test.mvcs.media.recorder	5	75%	5%	19
com.avaya.test.mvcs.media.rtp	12	51%	4%	56
com.avaya.test.mvcs.media.td	1	11%	0%	36

Reports generated by eXVantage.

This screenshot shows the output of a coverage tool for Java.

It shows the portion of lines and branches covered by a particular test for each package in the source code.

Coverage Reports: Classes



Packages

All
com.avaya.api.media
com.avaya.api.media.au
com.avaya.api.media.ch
com.avaya.api2.media
com.avaya.media.tonebu
com.avaya.mvcs.media
com.avaya.mvcs.media.
com.avaya.mvcs.media.
com.avaya.mvcs.media.

com.avaya.api.media

Classes

MediaFactory (50%)
MediaSession (N/A)

Coverage Report - com.avaya.api.media

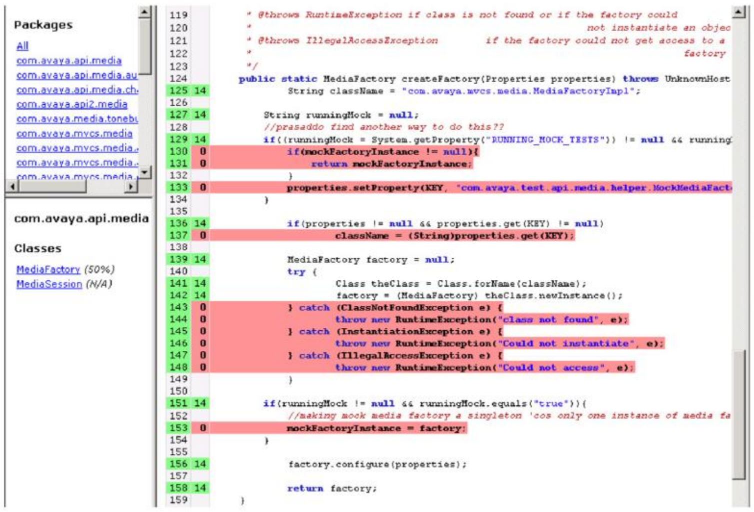
Package	# Classes	Line Coverage	Branch Coverage	Priority
com.avaya.api.media	2	50%	10%	3
com.avaya.api.media.audio	1	100%	0%	0
com.avaya.api.media.channels	2	100%	0%	0

Classes in this Package	Line Coverage	Branch Coverage	Priority
MediaFactory	50%	10%	3
MediaSession	N/A	N/A	0

Reports generated by eXVantage.

We can also view this information at the level of individual classes.

Coverage Reports: Source Code


Packages

All
com.avaya.api.media
com.avaya.api.media.au
com.avaya.api.media.ch
com.avaya.api2.media
com.avaya.media.tonebu
com.avaya.mvcs.media
com.avaya.mvcs.media.
com.avaya.mvcs.media.
com.avaya.mvcs.media.

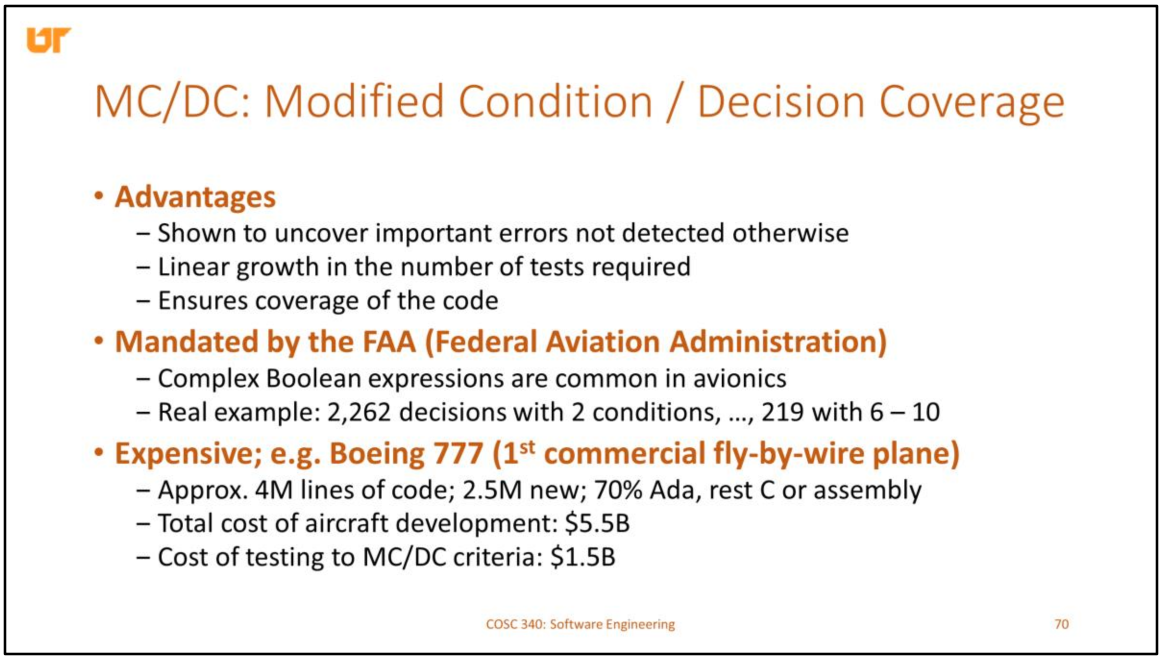
com.avaya.api.media

Classes

MediaFactory *(50%)*
MediaSession *(N/A)*

```
119        * @throws RuntimeException if class is not found or if the factory could
120        *                                                      not instantiate an objec
121        * @throws IllegalAccessException       if the factory could not get access to a
122        *                                                                       factory
123        */
124      public static MediaFactory createFactory(Properties properties) throws UnknownHost
125 14           String className = "com.avaya.mvcs.media.MediaFactoryImpl";
126
127 14       String runningMock = null;
128          //prasaddo find another way to do this??
129 14       if((runningMock = System.getProperty("RUNNING_MOCK_TESTS")) != null && running
130  0           if(mockFactoryInstance != null){
131  0               return mockFactoryInstance;
132              }
133  0           properties.setProperty(KEY, "com.avaya.test.api.media.helper.MockMediaFact
134          }
135
136 14           if(properties != null && properties.get(KEY) != null)
137  0                   className = (String)properties.get(KEY);
138
139 14           MediaFactory factory = null;
140              try {
141 14                   Class theClass = Class.forName(className);
142 14                   factory = (MediaFactory) theClass.newInstance();
143  0           } catch (ClassNotFoundException e) {
144  0                   throw new RuntimeException("class not found", e);
145  0           } catch (InstantiationException e) {
146  0                   throw new RuntimeException("Could not instantiate", e);
147  0           } catch (IllegalAccessException e) {
148  0                   throw new RuntimeException("Could not access", e);
149              }
150
151 14       if(runningMock != null && runningMock.equals("true")){
152              //making mock media factory a singleton 'cos only one instance of media fa
153  0           mockFactoryInstance = factory;
154          }
155
156 14           factory.configure(properties);
157
158 14           return factory;
159      }
```

And even at the level of individual source files. Lines highlighted in red were not reached during the test.



MC/DC: Modified Condition / Decision Coverage

- **Advantages**
  - Shown to uncover important errors not detected otherwise
  - Linear growth in the number of tests required
  - Ensures coverage of the code
- **Mandated by the FAA (Federal Aviation Administration)**
  - Complex Boolean expressions are common in avionics
  - Real example: 2,262 decisions with 2 conditions, …, 219 with 6 – 10
- **Expensive; e.g. Boeing 777 (1st commercial fly-by-wire plane)**
  - Approx. 4M lines of code; 2.5M new; 70% Ada, rest C or assembly
  - Total cost of aircraft development: $5.5B
  - Cost of testing to MC/DC criteria: $1.5B

COSC 340: Software Engineering 70

Today, I want to talk about another type of code coverage that is often used in safety critical software and that is modified condition / decision coverage or MC/DC.

The basic idea behind MCDC is that if a choice can be made in your code, then all the possible factors (or conditions) that can contribute to that choice (or decision) must be tested.

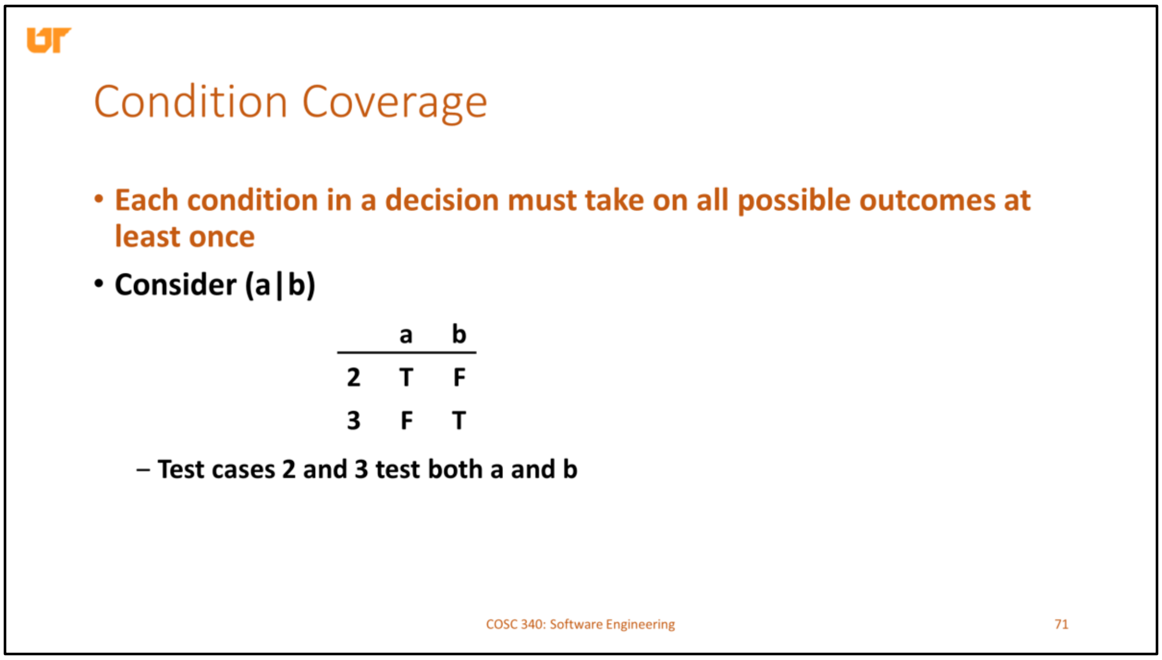
It’s been shown to be a very powerful testing technique that can uncover important errors that would go undetected with other techniques. I’ll expand more on this point a bit later.

MCDC was designed to be used with safety critical software or software that absolutely could not fail. For such software, some people used to advocate for exhaustive testing of all decisions in the code. That is, for a decision with n inputs, you would test all combinations of all inputs to ensure the correct outcome was always produced. Obviously, the problem with exhaustive testing is that it can produce an exponential number of tests.

One of the main advantages of MCDC is that the number of tests required is linear relative to the number of inputs to each decision in your code. So, for instance, in a decision with n inputs, the minimum number of tests required by MCDC is n+1. This is much better than the exhaustive where the number of tests is 2^n.

MCDC testing is widely used in practice for safety critical software. In fact, the FAA mandates MCDC testing for avionics software. This code can be quite complex – and may contain many complex Boolean expressions that would be impractical to test with exhaustive testing.

However, MCDC testing is quite expensive compared to some of the simpler coverage analysis tools. It does require much more testing than some of the other approaches we've discussed. In the development of the Boeing 777, which was the 1st commercial airplane with fully electronic flight controls, the MCDC testing cost more 25% of the aircraft's total development budget.


Condition Coverage

- Each condition in a decision must take on all possible outcomes at least once
- Consider (a|b)

	a	b
2	T	F
3	F	T

  - Test cases 2 and 3 test both a and b

COSC 340: Software Engineering 71

To better understand MCDC, we need to discuss a couple other types of coverage.

Condition coverage requires that each condition in a decision must take on all possible outcomes at least once.

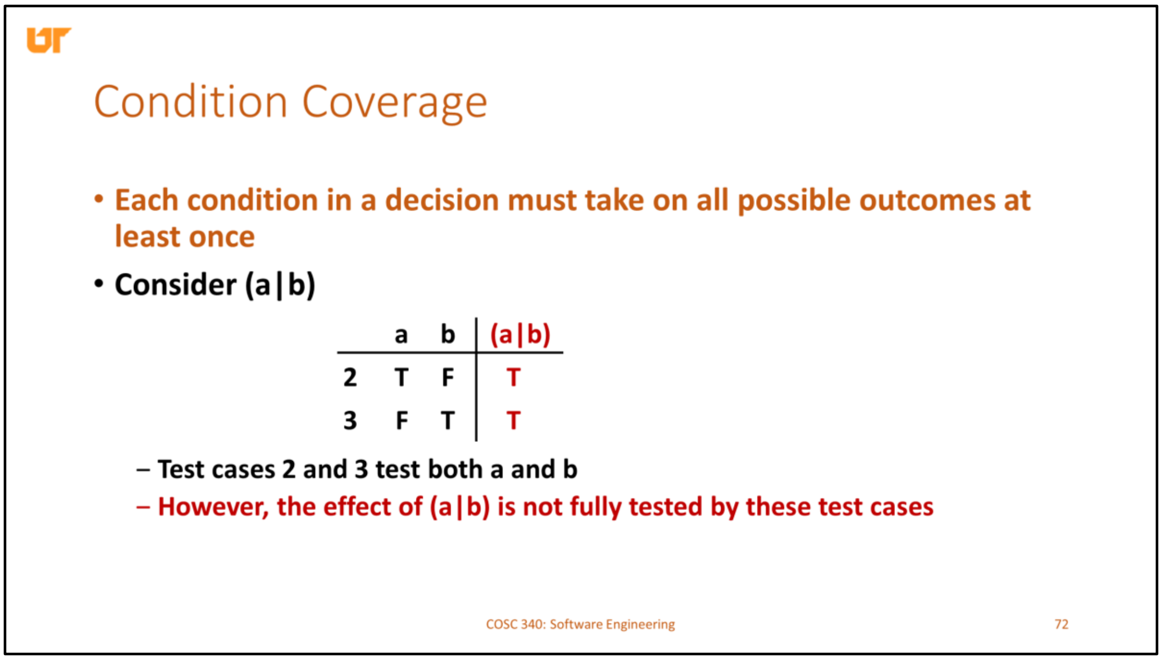
Suppose we have code that makes a decision based off the condition (a|b)

Now, exhaustive coverage would require that we test all possible combinations of a and b

```
   a  b
---------
1   T  T
2   T  F
3   F  T
4   F  F
```

But condition coverage just says that each condition must take on all possible outcomes at least once. So, we only need to include tests 2 and 3 because a is true in

2 and false in 3, and b is false in 2 and true in 3.



Condition Coverage

- **Each condition in a decision must take on all possible outcomes at least once**
- **Consider (a|b)**

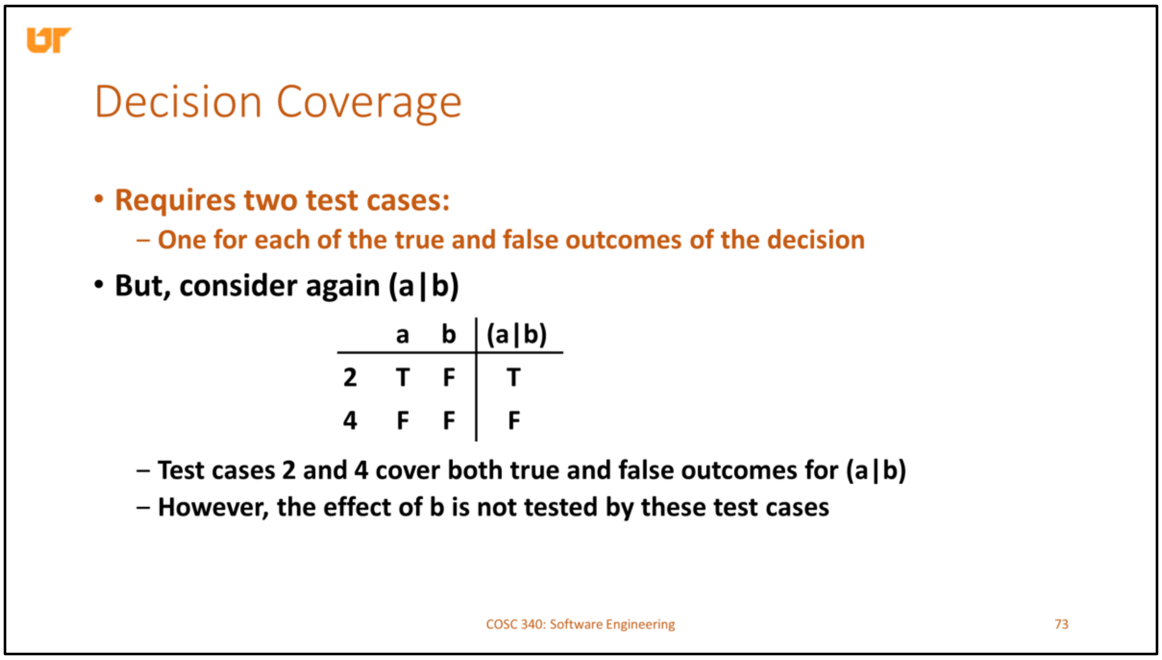
	a	b	(a|b)
2	T	F	T
3	F	T	T

  - **Test cases 2 and 3 test both a and b**
  - **However, the effect of (a|b) is not fully tested by these test cases**

COSC 340: Software Engineering 72

So, condition coverage allows for fewer tests than exhaustive coverage.

But, the downside of condition coverage is that the decision might not take all possible outcomes with these tests. So, you might miss testing some important parts of your program.



Decision Coverage

- **Requires two test cases:**
  - **One for each of the true and false outcomes of the decision**
- **But, consider again (a|b)**

	a	b	(a|b)
2	T	F	T
4	F	F	F

  - **Test cases 2 and 4 cover both true and false outcomes for (a|b)**
  - **However, the effect of b is not tested by these test cases**

COSC 340: Software Engineering 73

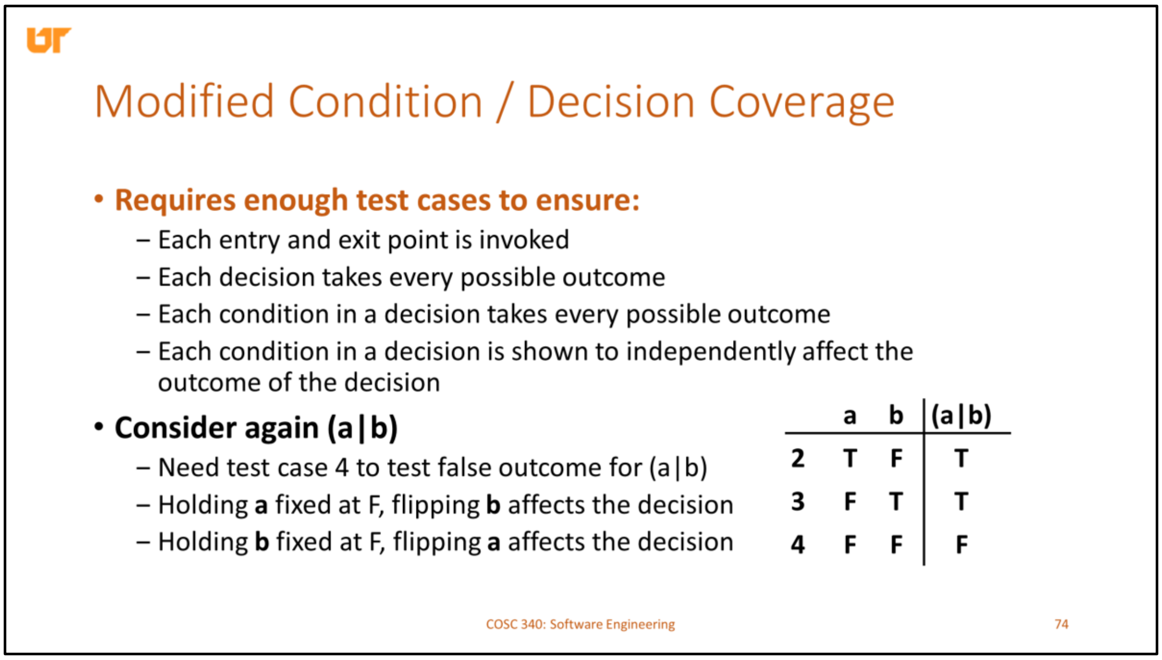
Another type of coverage is decision coverage.

Decision coverage just says that you need a test to test all possible outcomes of each decision.

If we consider again (a|b), we only need to include 1 test where outcome is true and one test where the outcome is false.

So, we could choose tests 2 and 4 to ensure decision coverage.

However, the downside here is that you might not test the effect of all the conditions on each test. For instance, these tests do not ensure that the program will behave correctly when b is true.



Modified Condition / Decision Coverage

- **Requires enough test cases to ensure:**
  - Each entry and exit point is invoked
  - Each decision takes every possible outcome
  - Each condition in a decision takes every possible outcome
  - Each condition in a decision is shown to independently affect the outcome of the decision
- **Consider again (a|b)**
  - Need test case 4 to test false outcome for (a|b)
  - Holding **a** fixed at F, flipping **b** affects the decision
  - Holding **b** fixed at F, flipping **a** affects the decision

	a	b	(a|b)
2	T	F	T
3	F	T	T
4	F	F	F

COSC 340: Software Engineering 74

To overcome the drawbacks of condition coverage and decision coverage, we can use modified condition / decision coverage.

There are four criteria for MCDC coverage. We need to run enough test cases to ensure that:

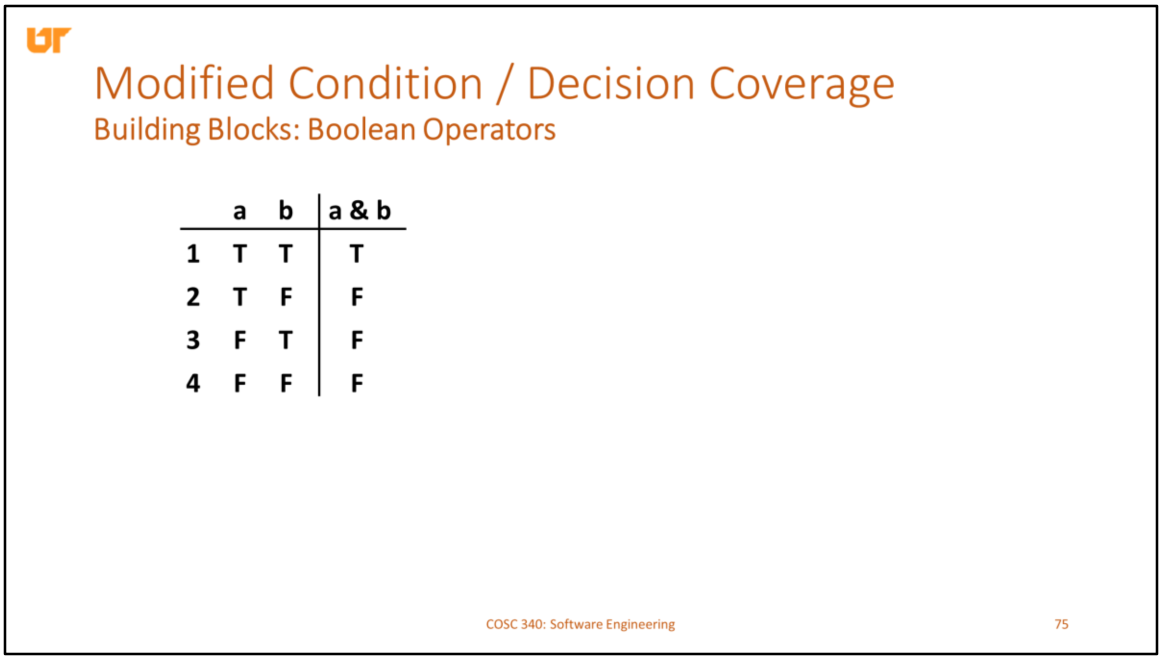
Each entry and exit point is invoked
Each decision takes every possible outcome
Each condition in a decision takes every possible outcome
Each condition in a decision is shown to independently affect the outcome of the decision.

Now, you might think we only need the first three. Why do we need to make sure that we have test cases where each condition in a decision independently affects the outcome? The basic idea is that, in many cases, some conditions of a decision may be masked by the other conditions. With MCDC, for each condition, you hold all the other conditions fixed and choose test cases where changing the condition will affect the outcome of the decision.

This property makes MCDC much stronger than decision coverage or condition coverage alone.

Let’s consider again the example of (a|b) as input to some decision. We need to choose test case 4 to test the false outcome.

Now, for the true outcome we have three choices. We only need to choose tests where flipping the value of the condition independently affects the result of the decision. So, for instance, If we hold a fixed at False, then flipping b affects the decision (so we need to choose tests 3 and 4 to cover that case). Holding b fixed at false, we see that flipping a affects the decision (so we need to choose tests 2 and 4 to cover that case). In this case, we don’t need test 1 to ensure MCDC.



Modified Condition / Decision Coverage

Building Blocks: Boolean Operators

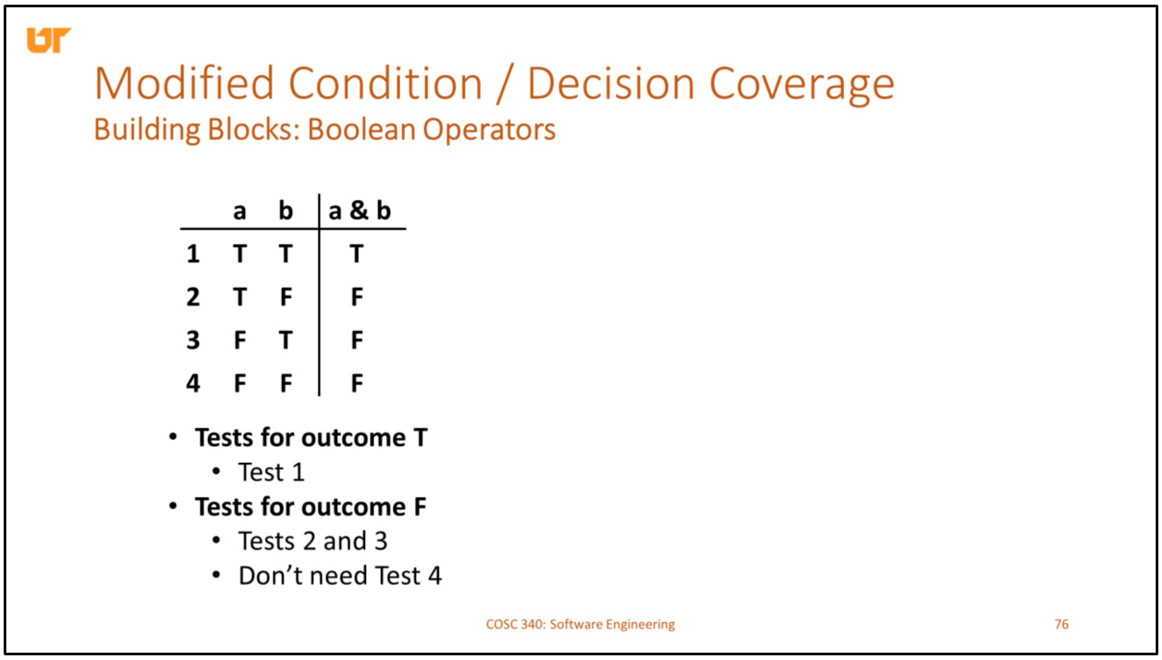
	a	b	a & b
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

Let’s do another example with a basic Boolean operator.

Suppose we have the condition in our code a&b. To determine the tests we need to run for MCDC, we first draw the truth table.

Now, we need to cover all possible outcomes for the decision, so right off the bat, we know we need to include test 1 to cover the case where the decision is true. For the false case, we have three possibilities. If we were to fix a at true, we see that flipping b affects the outcome, so we need to include tests 1 & 2. And if we fix b at true, flipping a also affects the outcome, so we need to include both 1 and 3.

In this case, we do not need to include test 4.

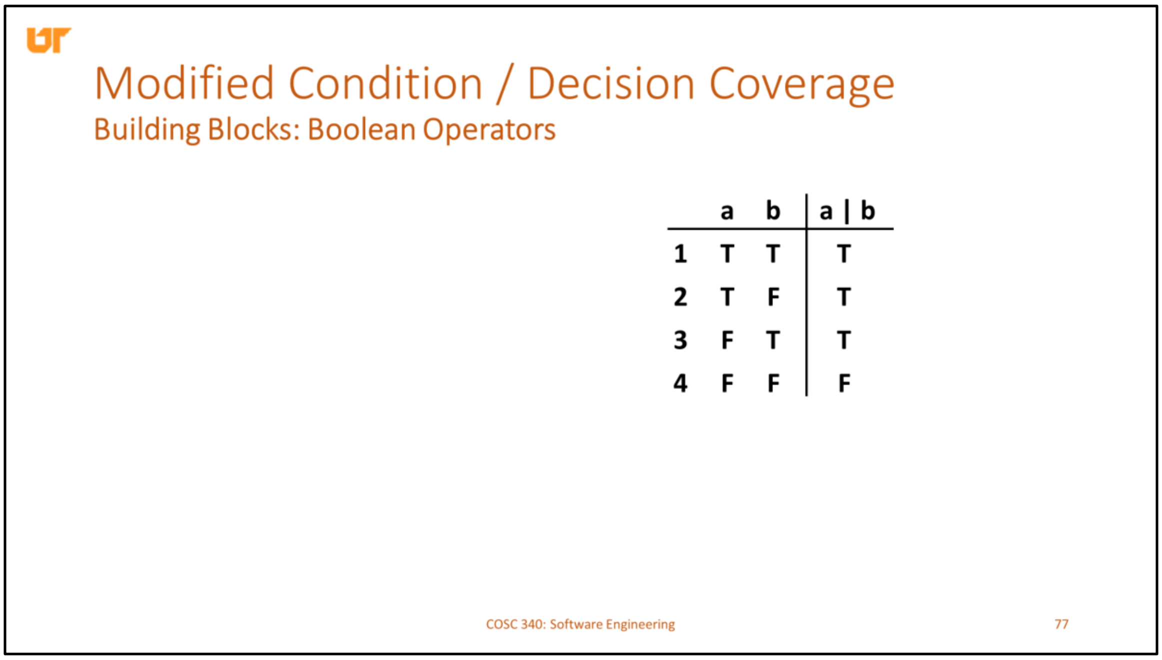


Modified Condition / Decision Coverage

Building Blocks: Boolean Operators

	a	b	a & b
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

- **Tests for outcome T**
  - Test 1
- **Tests for outcome F**
  - Tests 2 and 3
  - Don’t need Test 4



Modified Condition / Decision Coverage

Building Blocks: Boolean Operators

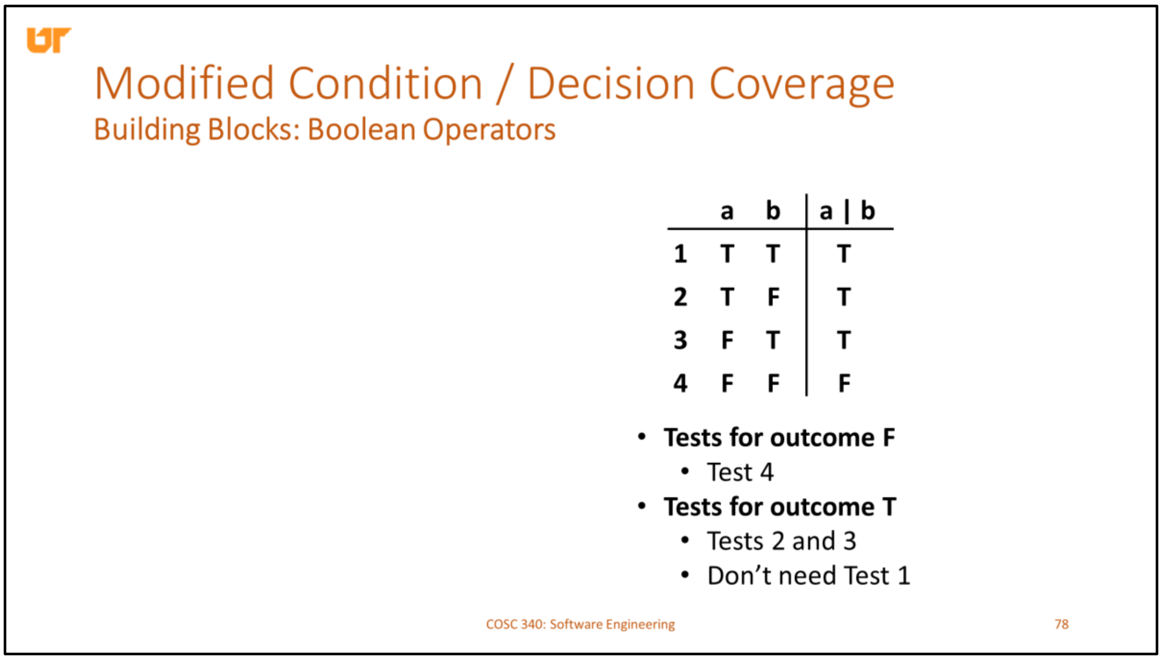
	a	b	a | b
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

COSC 340: Software Engineering 77

Same example as before.

Must select test 4 to get the case where the outcome will be false.
If we fix a at false, flipping b affects the outcome, so we need to include 3 and 4
If we fix b at false, flipping a affects the outcome, so we need to include 2 and 4.

Do not need to include 1.



Modified Condition / Decision Coverage

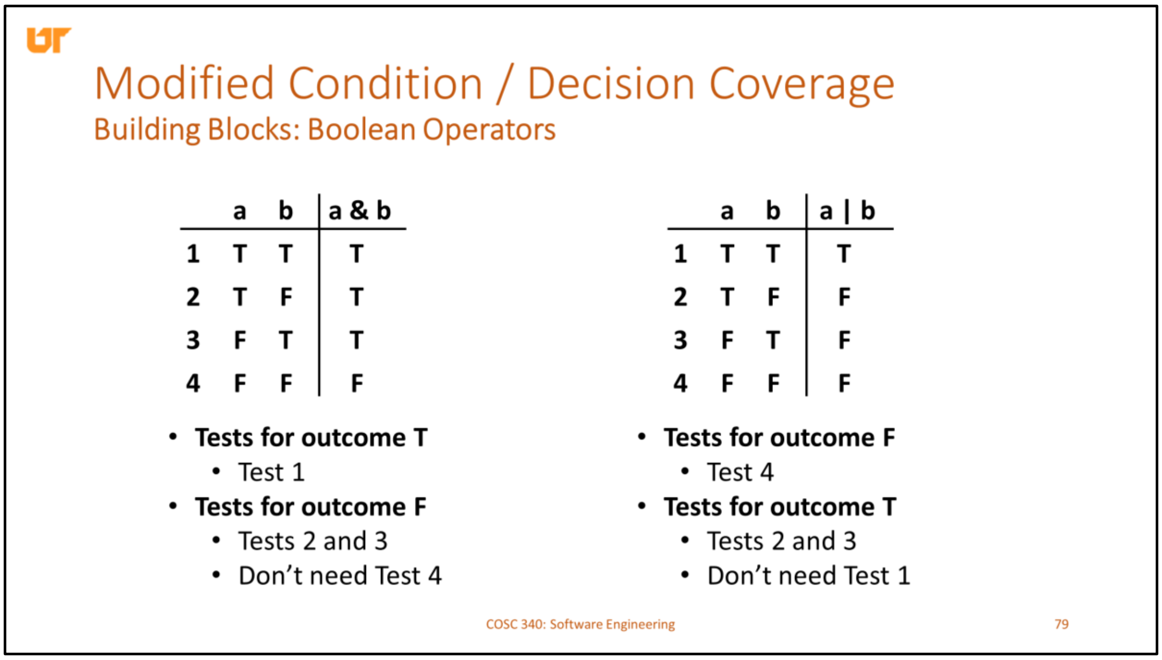
Building Blocks: Boolean Operators

	a	b	a | b
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

- **Tests for outcome F**
  - Test 4
- **Tests for outcome T**
  - Tests 2 and 3
  - Don’t need Test 1

COSC 340: Software Engineering 78

Need to include tests 2, 3, and 4. No need to include test 1.



Modified Condition / Decision Coverage

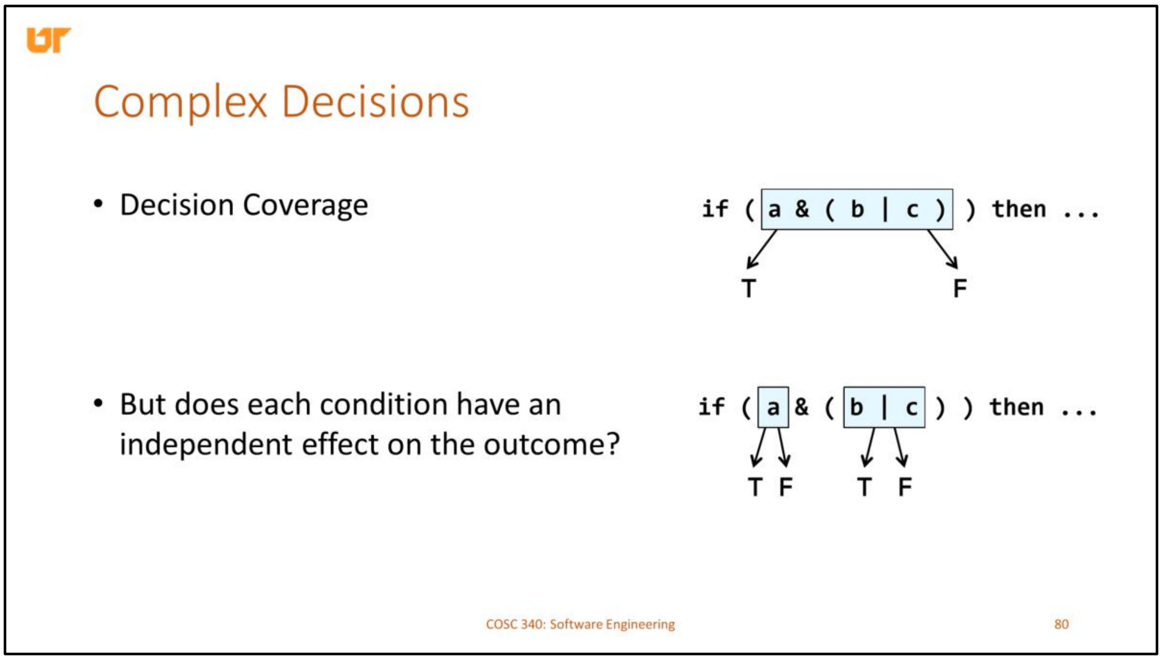
Building Blocks: Boolean Operators

	a	b	a & b
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

- **Tests for outcome T**
  - Test 1
- **Tests for outcome F**
  - Tests 2 and 3
  - Don’t need Test 4

	a	b	a | b
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

- **Tests for outcome F**
  - Test 4
- **Tests for outcome T**
  - Tests 2 and 3
  - Don’t need Test 1



Complex Decisions

- Decision Coverage
- But does each condition have an independent effect on the outcome?

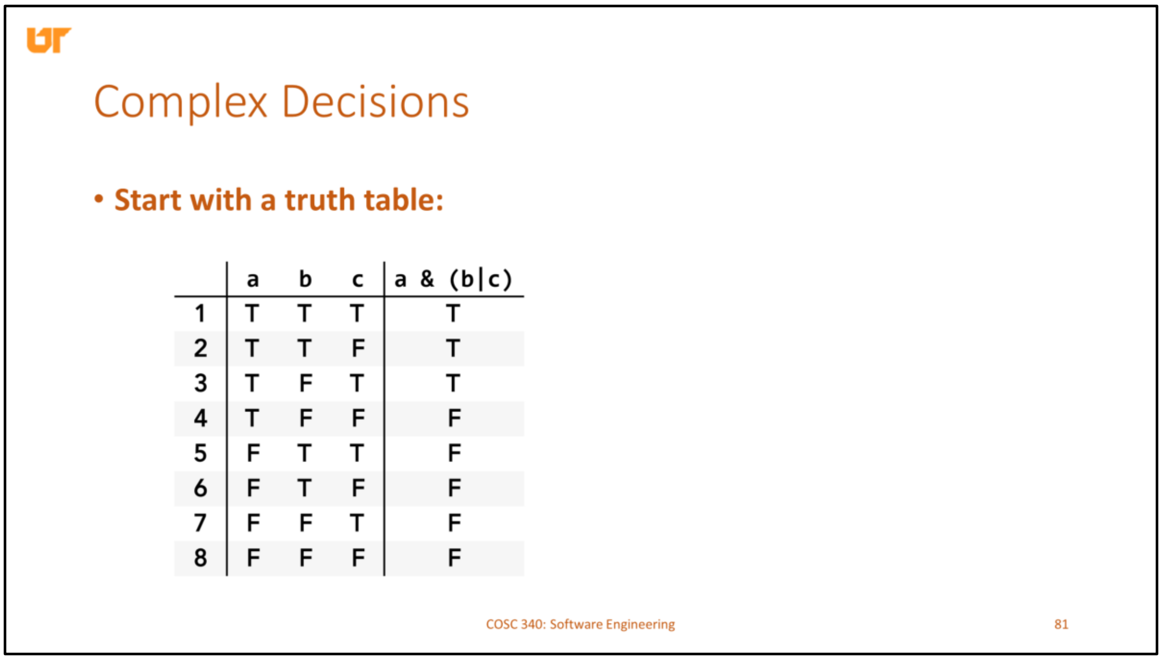
```
if ( a & ( b | c ) ) then ...
```

Now, what if we want to do MCDC for more complex decisions?

For instance, what about the condition (a&(b|c))?

Determining decision coverage is simple – we know we need to include tests that cover each possible outcome of the decision.

But how do we determine if each condition has an independent effect on the outcome of the decision? And which tests do we need to include to show that each condition has an independent effect on the outcome of the decision?



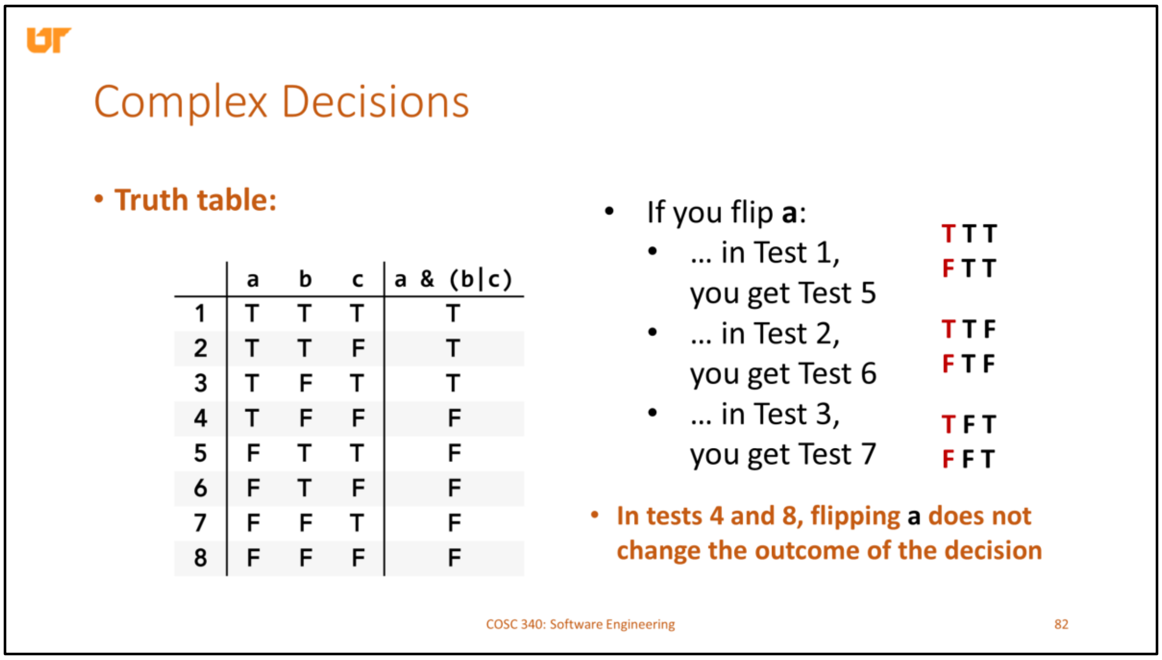
Complex Decisions

- **Start with a truth table:**

	a	b	c	a & (b|c)
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

COSC 340: Software Engineering 81

As with many things in life, the first step is to draw up the truth table.



Complex Decisions

- **Truth table:**

	a	b	c	a & (b|c)
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

- If you flip **a**:
  - … in Test 1, you get Test 5 — T T T / F T T
  - … in Test 2, you get Test 6 — T T F / F T F
  - … in Test 3, you get Test 7 — T F T / F F T
- **In tests 4 and 8, flipping a does not change the outcome of the decision**

COSC 340: Software Engineering 82

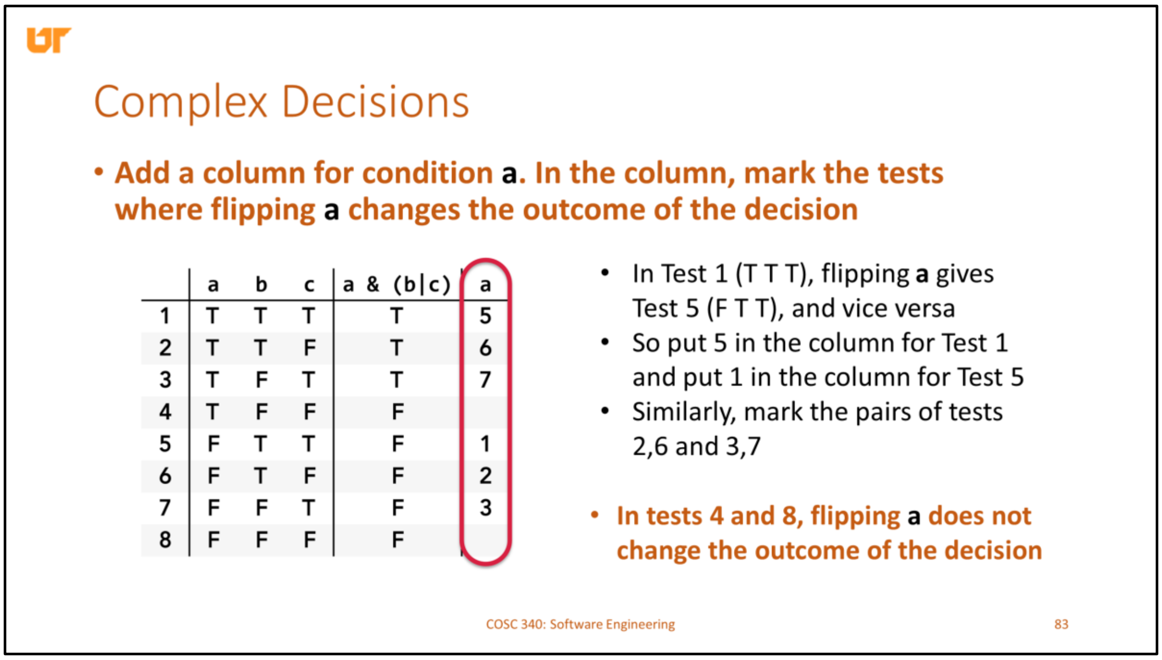
From here, we see that if we flip the value of a between true and false, it often affects the outcome of the decision.

Observe that:

If we flip the value of a in test 1, we get test 5, which does lead to a different outcome.
If we flip the value of a in test 2, we get test 6, which also has a different outcome than test 2.
And if we flip the value of a in test 3, we get test 7 and the outcomes of 3 and 7 are different.

So, a independently affects the outcome of tests 1 and 5, tests 2 and 6, and tests 3 and 7.

If we were to flip the value of a in test 4, that does correspond to a different test (test 8), but tests 4 and 8 lead to the same outcome. So, in regards to those tests, a does not matter.



Complex Decisions

- **Add a column for condition a. In the column, mark the tests where flipping a changes the outcome of the decision**

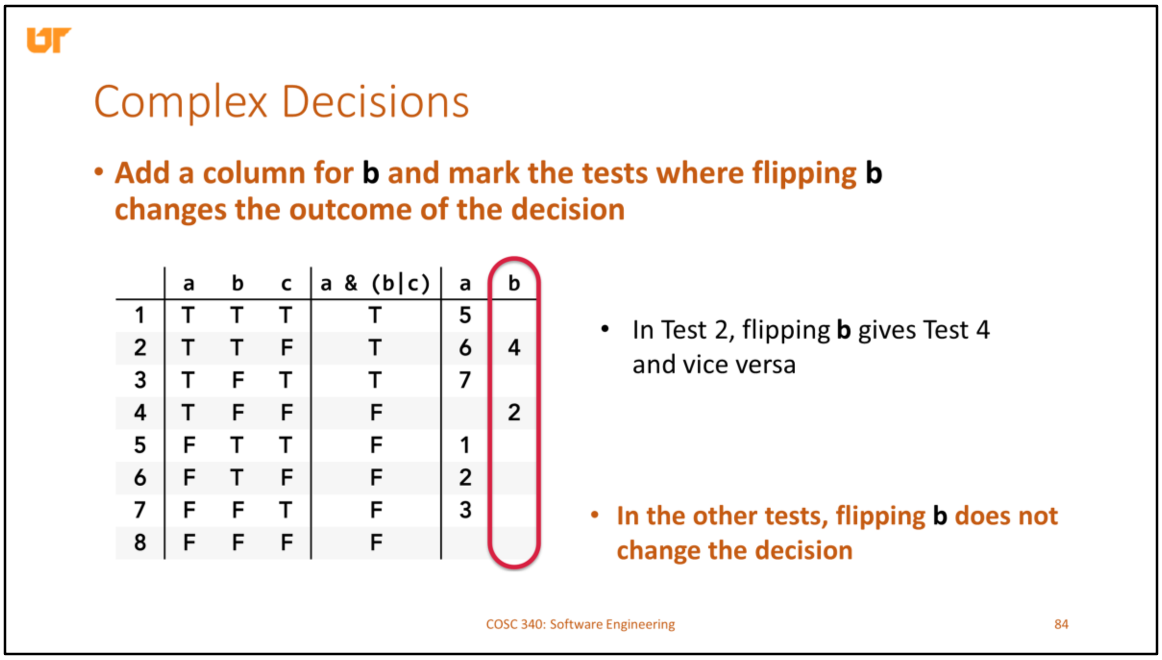
	a	b	c	a & (b|c)	a
1	T	T	T	T	5
2	T	T	F	T	6
3	T	F	T	T	7
4	T	F	F	F	
5	F	T	T	F	1
6	F	T	F	F	2
7	F	F	T	F	3
8	F	F	F	F	

- In Test 1 (T T T), flipping **a** gives Test 5 (F T T), and vice versa
- So put 5 in the column for Test 1 and put 1 in the column for Test 5
- Similarly, mark the pairs of tests 2,6 and 3,7

- **In tests 4 and 8, flipping a does not change the outcome of the decision**

Now, the next step is to take those tests where flipping the value of a affects the outcome of the decision and mark them in a separate column in the table.

In this column, we just write the corresponding test number for each test that affects the outcome of the decision.



Complex Decisions

- **Add a column for b and mark the tests where flipping b changes the outcome of the decision**

	a	b	c	a & (b|c)	a	b
1	T	T	T	T	5	
2	T	T	F	T	6	4
3	T	F	T	T	7	
4	T	F	F	F		2
5	F	T	T	F	1	
6	F	T	F	F	2	
7	F	F	T	F	3	
8	F	F	F	F		

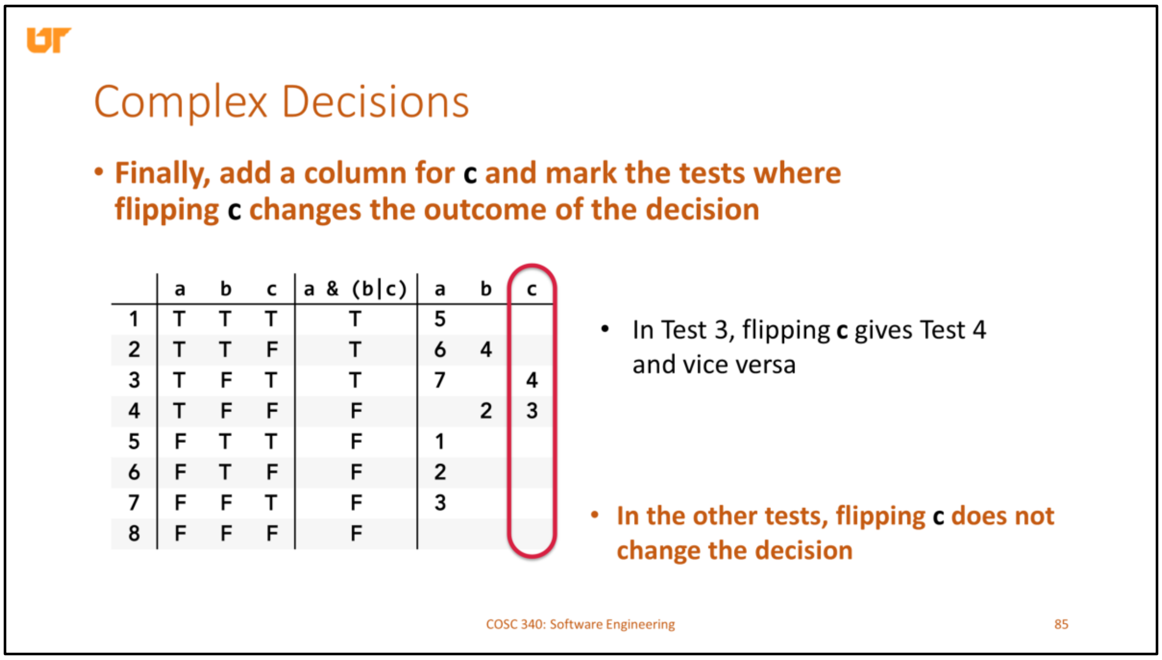
- In Test 2, flipping **b** gives Test 4 and vice versa

- **In the other tests, flipping b does not change the decision**

COSC 340: Software Engineering 84

We do the same for input b. For b, we find that flipping b in test 2 gives us test 4, which has a different result than test 2.

Flipping the value of b in the other tests does not affect the outcome of this decision.



Complex Decisions

- **Finally, add a column for c and mark the tests where flipping c changes the outcome of the decision**

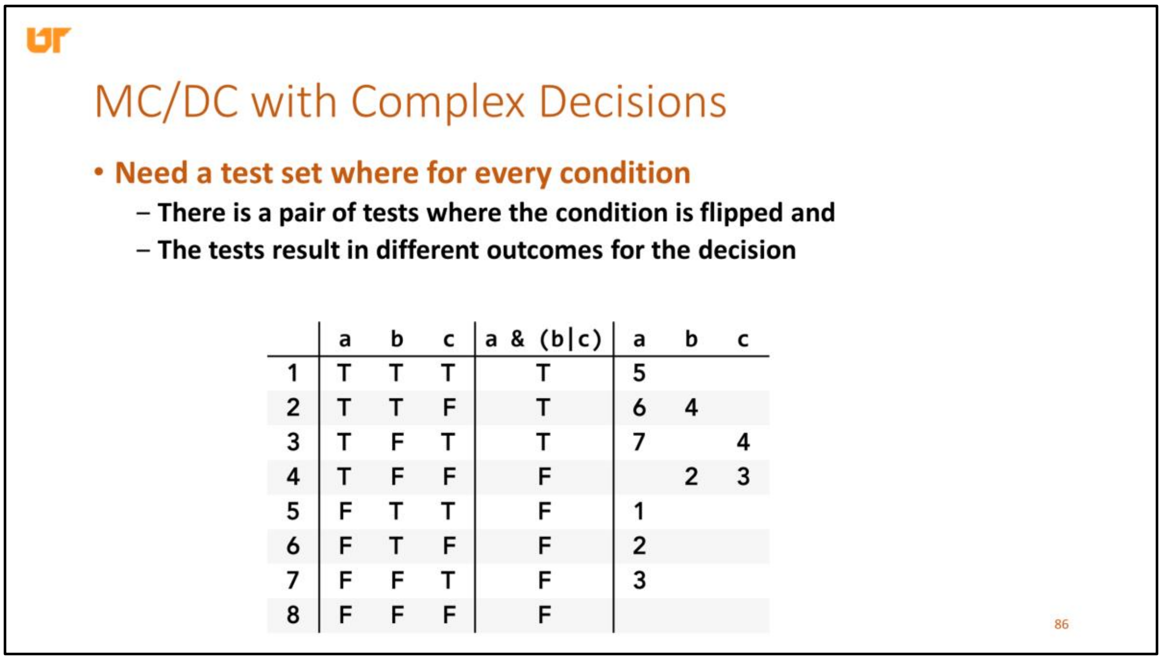
	a	b	c	a & (b|c)	a	b	c
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

- In Test 3, flipping **c** gives Test 4 and vice versa

- **In the other tests, flipping c does not change the decision**

COSC 340: Software Engineering 85

And we do the same for c.



MC/DC with Complex Decisions

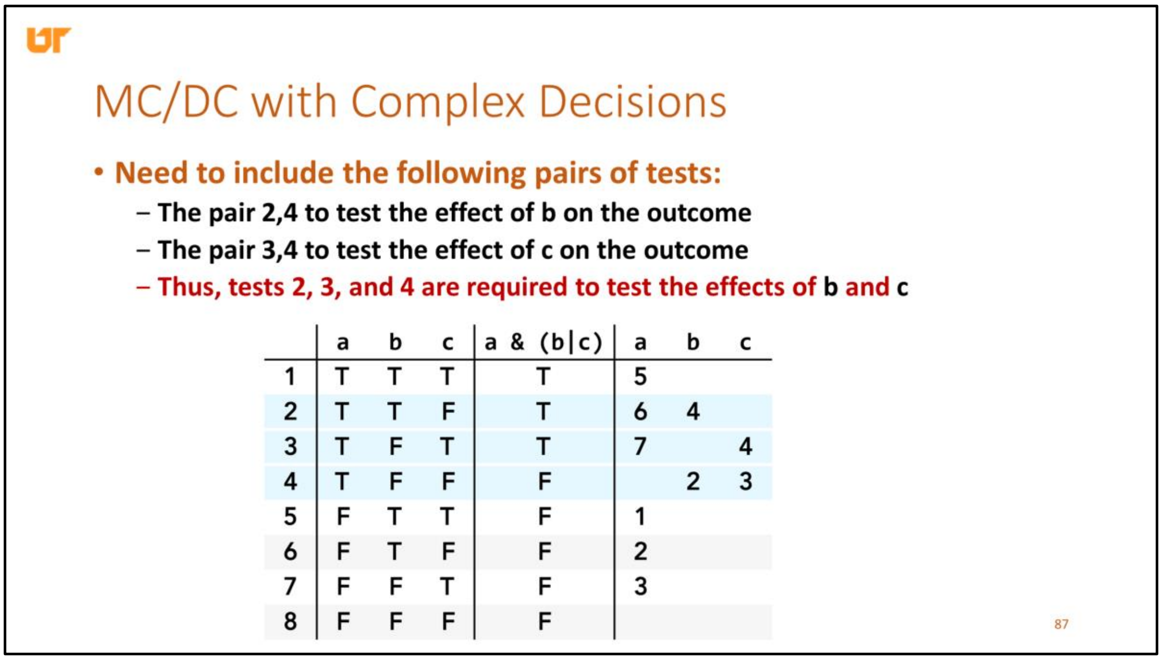
- **Need a test set where for every condition**
  - **There is a pair of tests where the condition is flipped and**
  - **The tests result in different outcomes for the decision**

	a	b	c	a & (b|c)	a	b	c
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

86

Now, we construct our test set by, for each condition, finding pairs of tests where the condition is flipped and the test results in different outcomes for the decision.

So, for condition b, we need to include both tests 4 and 2 because flipping the value of b in those tests results in a different outcome.
And, for condition c, we need to include both tests 4 and 3 because flipping the value of c in those tests leads to a different outcome.



MC/DC with Complex Decisions

- **Need to include the following pairs of tests:**
  - **The pair 2,4 to test the effect of b on the outcome**
  - **The pair 3,4 to test the effect of c on the outcome**
  - **Thus, tests 2, 3, and 4 are required to test the effects of b and c**

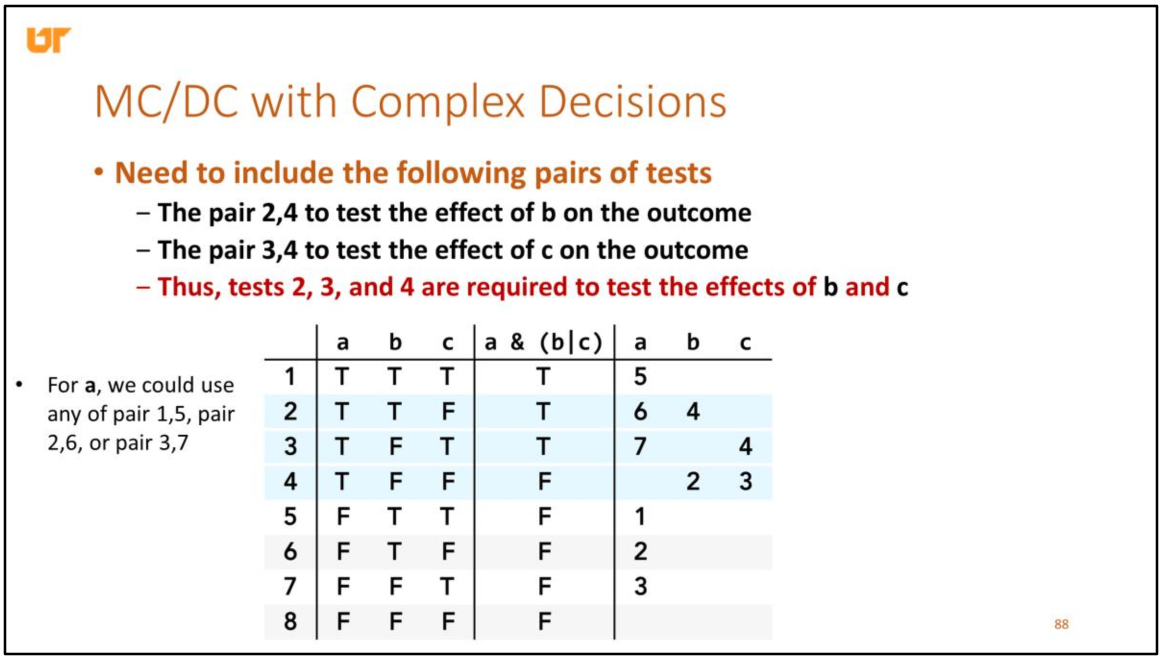
	a	b	c	a & (b|c)	a	b	c
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

87

So, to test the effects of b and c, we need to include tests 2, 3, and 4.

For a, we have several options. We only need to include one pair of tests that shows that a independently affects the outcome of the decision.

So, we could include tests 1 and 5, or we could include tests 2 and 6, or we could include tests 3 and 7.



MC/DC with Complex Decisions

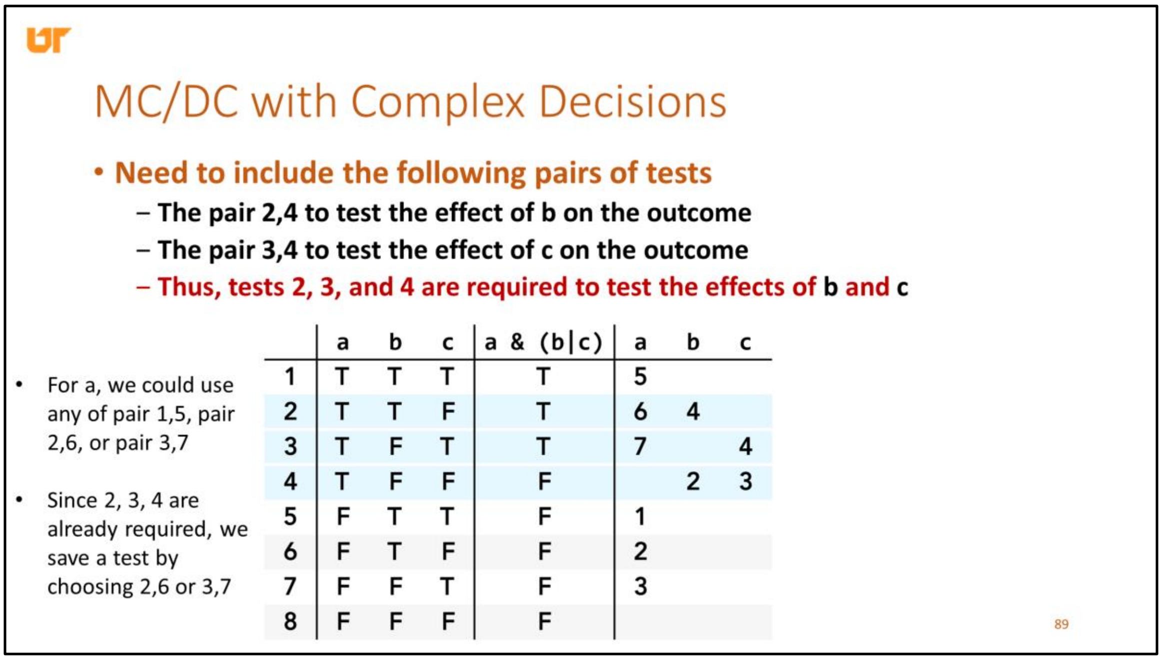
- **Need to include the following pairs of tests**
  - **The pair 2,4 to test the effect of b on the outcome**
  - **The pair 3,4 to test the effect of c on the outcome**
  - **Thus, tests 2, 3, and 4 are required to test the effects of b and c**

- For **a**, we could use any of pair 1,5, pair 2,6, or pair 3,7

	a	b	c	a & (b|c)	a	b	c
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

88

Continue.



MC/DC with Complex Decisions

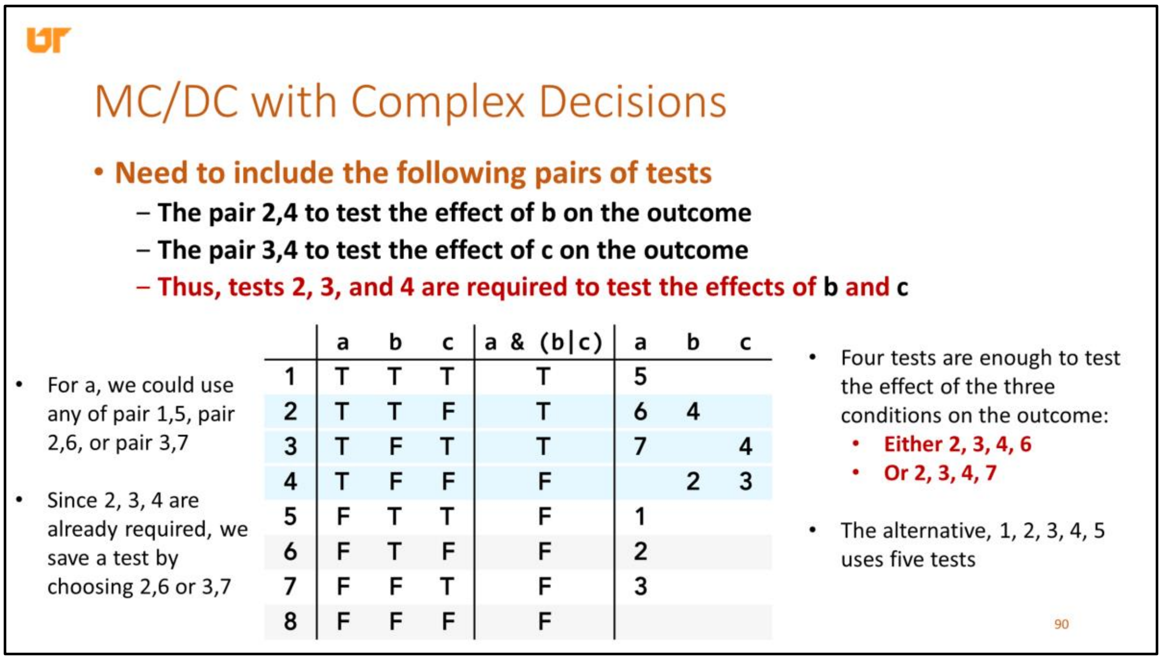
- **Need to include the following pairs of tests**
  - **The pair 2,4 to test the effect of b on the outcome**
  - **The pair 3,4 to test the effect of c on the outcome**
  - **Thus, tests 2, 3, and 4 are required to test the effects of b and c**

- For a, we could use any of pair 1,5, pair 2,6, or pair 3,7
- Since 2, 3, 4 are already required, we save a test by choosing 2,6 or 3,7

	a	b	c	a & (b|c)	a	b	c
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

89

Now, since 2, 3, and 4 are already required for b and c, we can save ourselves one test by choosing either 2 and 6 or 3 and 7



MC/DC with Complex Decisions

- **Need to include the following pairs of tests**
  - **The pair 2,4 to test the effect of b on the outcome**
  - **The pair 3,4 to test the effect of c on the outcome**
  - **Thus, tests 2, 3, and 4 are required to test the effects of b and c**

- For a, we could use any of pair 1,5, pair 2,6, or pair 3,7
- Since 2, 3, 4 are already required, we save a test by choosing 2,6 or 3,7

	a	b	c	a & (b|c)	a	b	c
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

- Four tests are enough to test the effect of the three conditions on the outcome:
  - **Either 2, 3, 4, 6**
  - **Or 2, 3, 4, 7**
- The alternative, 1, 2, 3, 4, 5 uses five tests

90

So, to ensure modified condition / decision coverage with this decision, we only need four tests. We could choose either 2, 3, 4, and 6 or 2, 3, 4, and 7.

We could also use 1, 2, 3, 4, and 5 – but this one requires an additional test.

Combinatorial Testing

NIST National Institute of Standards and Technology

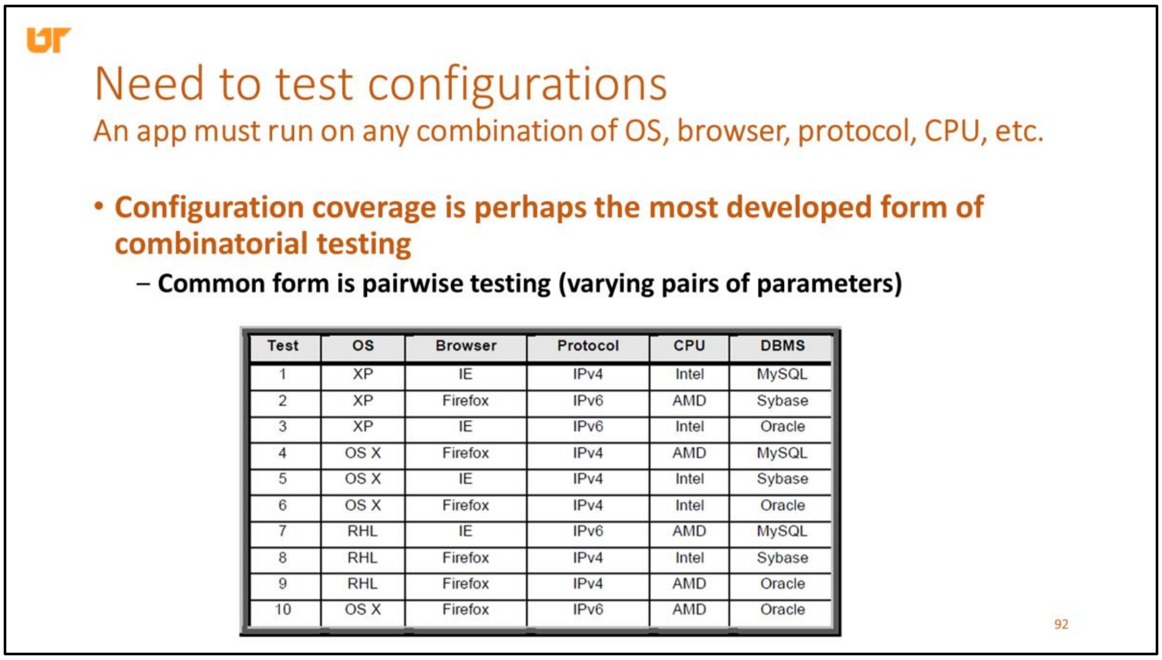
Introduction to Combinatorial Testing

Rick Kuhn
National Institute of Standards and Technology
Gaithersburg, MD

Carnegie-Mellon University, 7 June 2011

Lastly, I want to talk about a more recent testing method called combinatorial testing that was developed and made popular by Rick Kuhn from NIST.

By thinking of inputs as combinations of parameters, we may be able to significantly reduce the number of tests we need to run.



Need to test configurations

An app must run on any combination of OS, browser, protocol, CPU, etc.

- **Configuration coverage is perhaps the most developed form of combinatorial testing**
  - **Common form is pairwise testing (varying pairs of parameters)**

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	IE	IPv4	Intel	Sybase
6	OS X	Firefox	IPv4	Intel	Oracle
7	RHL	IE	IPv6	AMD	MySQL
8	RHL	Firefox	IPv4	Intel	Sybase
9	RHL	Firefox	IPv4	AMD	Oracle
10	OS X	Firefox	IPv6	AMD	Oracle

92

The basic idea from combinatorial testing is that you often want to test different combinations of parameters.

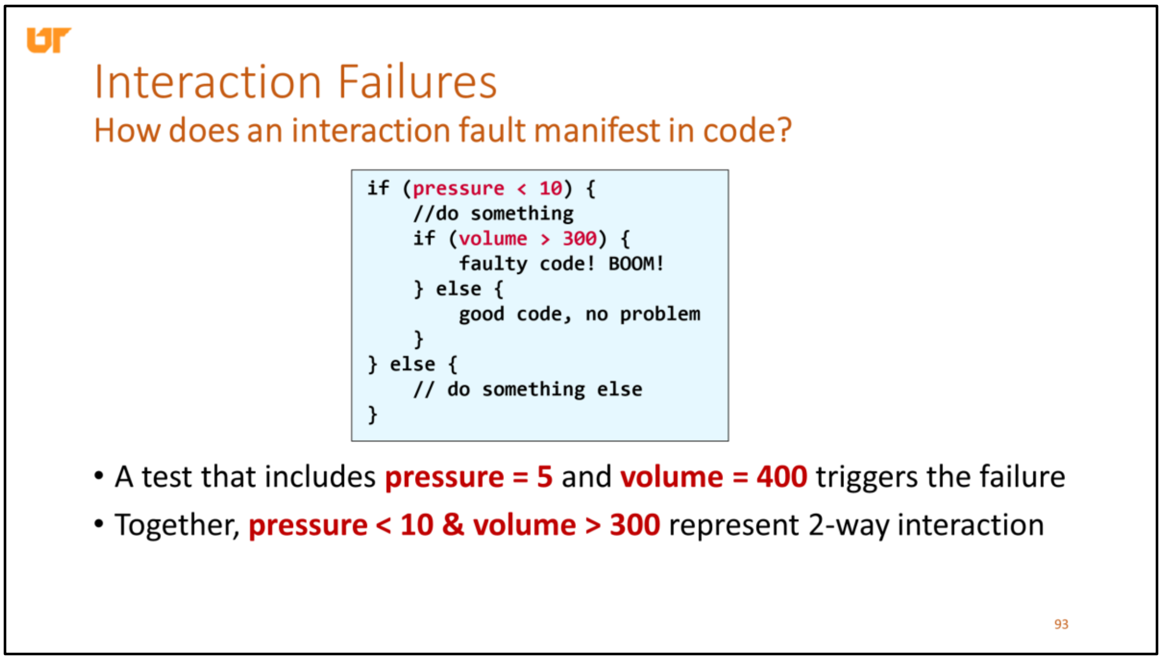
These different combinations of parameters might interact in your code in ways you might not have expected and cause errors.

An example of where combinatorial testing would be useful is in ensuring configuration coverage.

For many applications, we need to test different combinations of configurations. For example, you might have a web app that you want to be able to run in different browsers, on different operating systems, running on different types of processors. Even more complicated, you might want to be able to use it with different database backends or connect to it using different protocols.

Testing all possible combinations of these parameters could be challenging.

One of the most common shortcuts for configuration testing is to use pairwise testing. With pairwise testing, you select two input parameters at a time and test all possible discrete combinations of those parameters. So, you might test the app with each type of browser on the different operating systems. Then, test the app on the different browsers on different architectures. This is faster than testing all possible combinations.



Interaction Failures

How does an interaction fault manifest in code?

```
if (pressure < 10) {
    //do something
    if (volume > 300) {
        faulty code! BOOM!
    } else {
        good code, no problem
    }
} else {
    // do something else
}
```

- A test that includes **pressure = 5** and **volume = 400** triggers the failure
- Together, **pressure < 10 & volume > 300** represent 2-way interaction

93

Let’s discuss the rationale behind combinatorial testing.

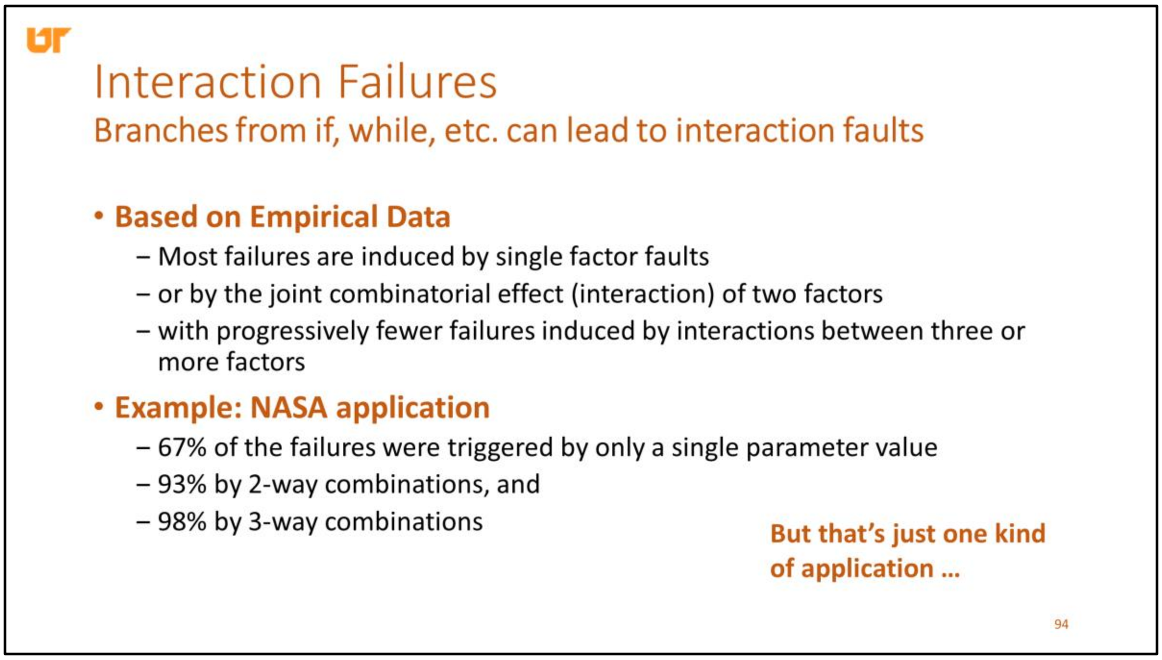
Combinatorial testing can help detect interaction failures.

An interaction failure occurs when two inputs together interact to induce a failure.

So, for example, in this code, if pressure < 10, we might or might not induce a failure. But, if pressure < 10 and volume > 300, we will reach this faulty code and some bad things will happen.

The condition “pressure < 10 and volume > 300” represents a two-way interaction that can induce the failure.

During testing, you want to include tests that cover this interaction.



Interaction Failures

Branches from if, while, etc. can lead to interaction faults

- **Based on Empirical Data**
  - Most failures are induced by single factor faults
  - or by the joint combinatorial effect (interaction) of two factors
  - with progressively fewer failures induced by interactions between three or more factors
- **Example: NASA application**
  - 67% of the failures were triggered by only a single parameter value
  - 93% by 2-way combinations, and
  - 98% by 3-way combinations

But that's just one kind of application …

94

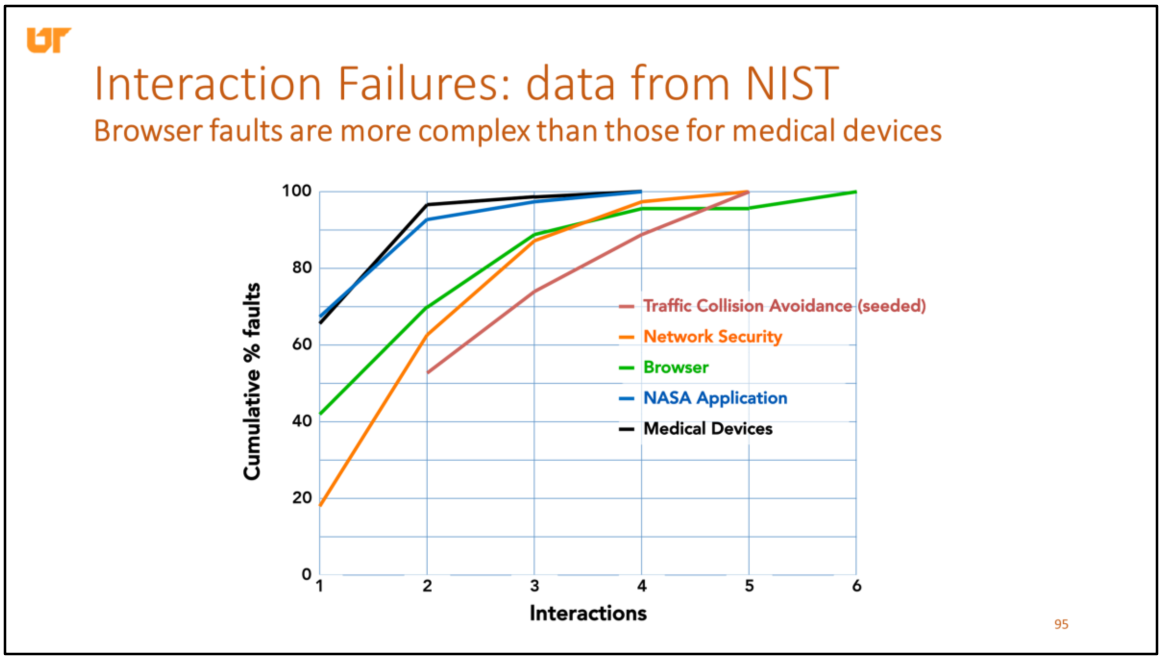
These interaction failures occur due to the way different branches in your code can interact.

So, given that these interaction faults can cause issues, an obvious question to ask is how often do faults occur due to different combinations of inputs.

Rick Kuhn at NIST studied the root cause of different faults for different kinds of applications.

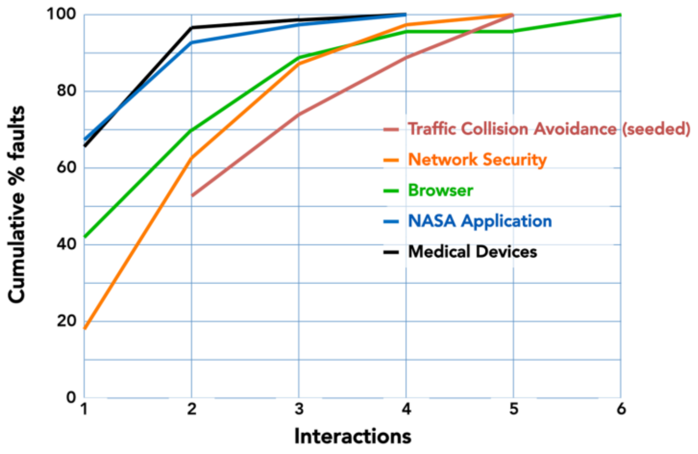
He found that most failures are induced by single factor faults (that is – you only need to vary the value of one parameter to induce the fault) or by 2-way interactions (just the combination of two factors). Progressively fewer failures are induced by interactions between 3 or more factors.

For example, when he looked at software produced by NASA … (read it)



Interaction Failures: data from NIST

Browser faults are more complex than those for medical devices



95

This figure is from Kuhn’s study.

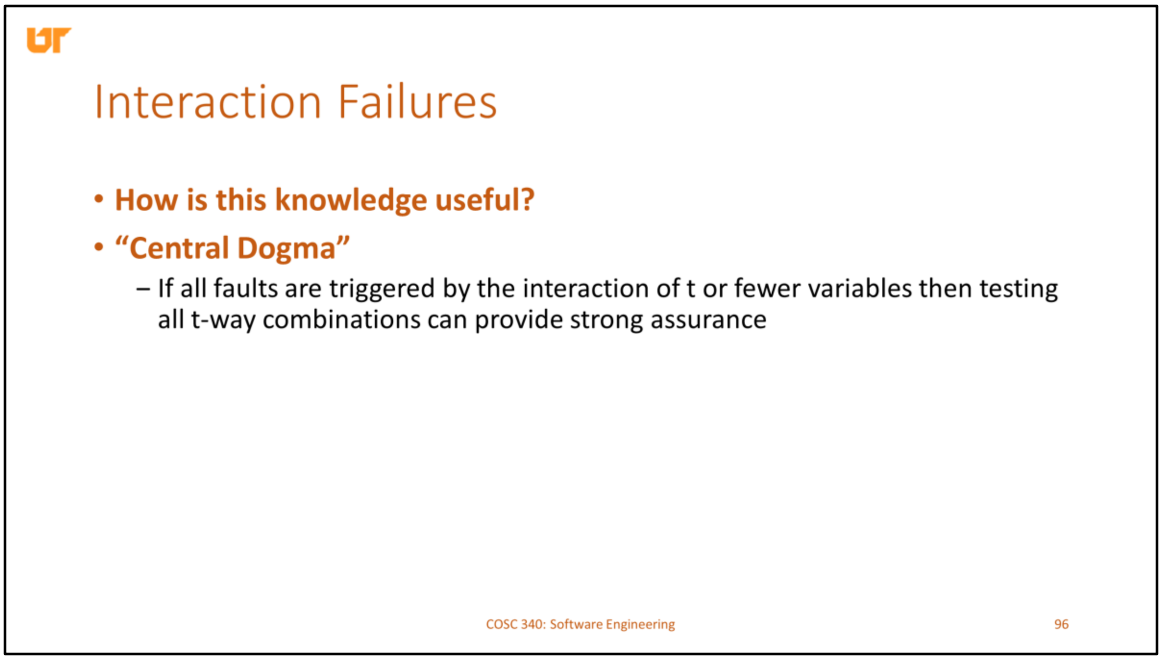
It shows the percentage of faults induced by tests with different numbers of interactions.

So, for instance, with medical devices, you can induce about 66% of failures with only 1-way interactions (i.e. by varying the value of just one parameter). About 95% of interactions are induced by 1-way or 2-way interactions.

And only a very small number of failures are induced by 3 or more interactions between program inputs.

This data is interesting for several reasons. We see that faults might be more complex for different types of software. For instance, for browsers, generating all tests with 2-way interactions only induces about 70% of all of the faults.

Perhaps surprisingly, faults for network security software, browser software, and this traffic collision software they tested are significantly more complex than faults for NASA applications and medical devices.


Interaction Failures

- **How is this knowledge useful?**
- **"Central Dogma"**
  - If all faults are triggered by the interaction of t or fewer variables then testing all t-way combinations can provide strong assurance

COSC 340: Software Engineering 96

So, how could this knowledge be useful for testing?

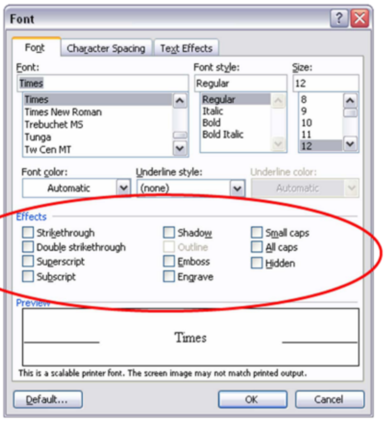
Well, Kuhn describes this finding as the rationale for the 'central dogma' of his work.

He says if all faults are triggered by the interaction of t or fewer variables, then testing all t-way combinations of your program inputs can provide strong assurance that it is working properly.

Combination of Effects

Text formatting example

- 10 effects, each can be turned on or off
- How many tests to test all combinations?


Font

Font | Character Spacing | Text Effects

Font: Times

Times, Times New Roman, Trebuchet MS, Tunga, Tw Cen MT

Font style: Regular

Regular, Italic, Bold, Bold Italic

Size: 12

8, 9, 10, 11, 12

Font color: Automatic

Underline style: (none)

Underline color: Automatic

Effects

- [ ] Strikethrough
- [ ] Double strikethrough
- [ ] Superscript
- [ ] Subscript
- [ ] Shadow
- [ ] Outline
- [ ] Emboss
- [ ] Engrave
- [ ] Small caps
- [ ] All caps
- [ ] Hidden

Preview

Times

This is a scalable printer font. The screen image may not match printed output.

Let's look at an example of how this dogma could be useful.

For instance, let's consider text formatting in Microsoft Word.

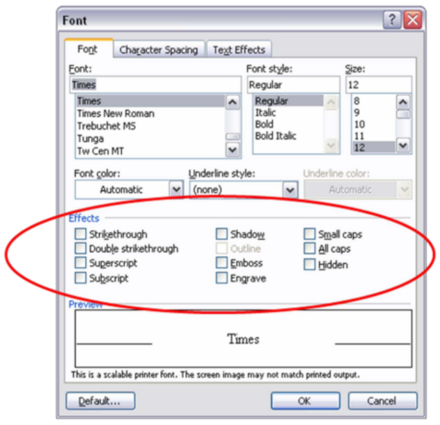
In Microsoft word, there are 10 different font effects you can apply to your text. Each effect can be applied in combination with the other effects. So, for instance, you can apply strikethrough along with the shadow effect along with small caps to your font.

If you wanted to thoroughly test your code, you might want to test all different combinations of these 10 effects. (how many tests would that be?) 2^10 = 1024

Combination of Effects

Text formatting example

- 10 effects, each can be turned on or off
- How many tests to test all combinations?
  - All combinations is 2^{10} = 1,024
- Might not be able to do so many tests
- Instead, look only at 3-way interactions


Font: Times
Font style: Regular
Size: 12
Font color: Automatic
Underline style: (none)
Underline color: Automatic
This is a scalable printer font. The screen image may not match printed output.

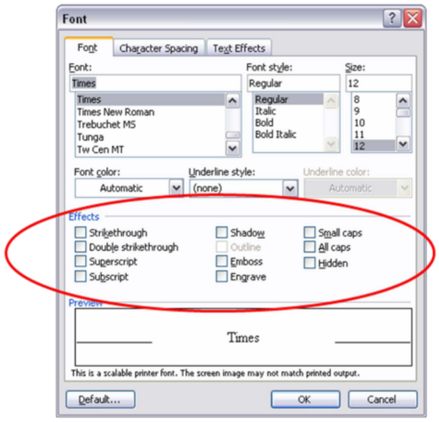
But, suppose that you're not able to test every single combination. Maybe your testing budget is limited – or you just don't want to spend the time running all those tests.

Well, because of Kuhn, we know that the vast majority of faults can be induced by 3-way interactions. So, we decide we only want to look at 3-way interactions.

Combination of Effects

Text formatting example

- **How many tests would it take to test all 3-way interactions?**
  - **How many combinations of 3 effects?**


Font

Font | Character Spacing | Text Effects

Font: Times
Times, Times New Roman, Trebuchet MS, Tunga, Tw Cen MT

Font style: Regular
Regular, Italic, Bold, Bold Italic

Size: 12
8, 9, 10, 11, 12

Font color: Automatic
Underline style: (none)
Underline color: Automatic

Effects
- ☐ Strikethrough
- ☐ Double strikethrough
- ☐ Superscript
- ☐ Subscript
- ☐ Shadow
- ☐ Outline
- ☐ Emboss
- ☐ Engrave
- ☐ Small caps
- ☐ All caps
- ☐ Hidden

Preview
Times

This is a scalable printer font. The screen image may not match printed output.

Now, the question is, how many tests would it take to test all 3-way interactions?

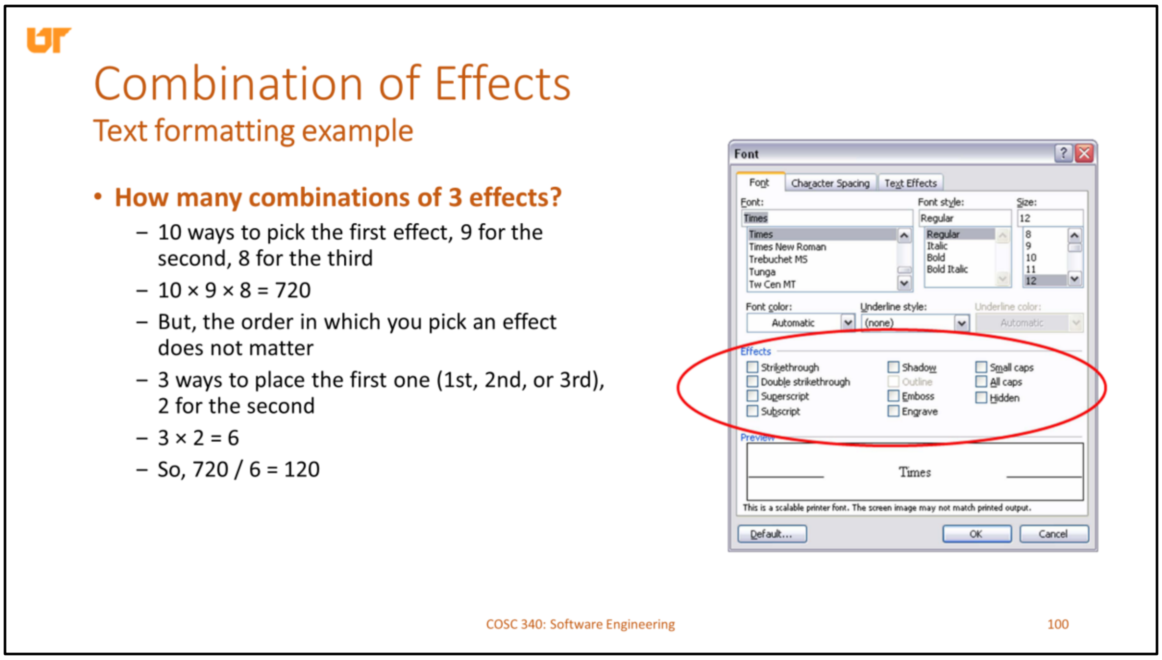
Well, first we need to determine how many combinations of 3 different effects can we have?

This is actually a simple counting problem.

10 ways to pick the first effect, 9 for the second, 8 for the third: $10 \times 9 \times 8 = 720$

But, the order in which you pick an effect does not matter
 3 ways to place the first one (1st, 2nd, or 3rd), 2 for the second
 $3 \times 2 = 6$

So, $720 / 6 = 120$



Combination of Effects

Text formatting example

- **How many combinations of 3 effects?**
  - 10 ways to pick the first effect, 9 for the second, 8 for the third
  - 10 × 9 × 8 = 720
  - But, the order in which you pick an effect does not matter
  - 3 ways to place the first one (1st, 2nd, or 3rd), 2 for the second
  - 3 × 2 = 6
  - So, 720 / 6 = 120

Font

Font | Character Spacing | Text Effects

Font: Times
Font style: Regular
Size: 12

Font color: Automatic
Underline style: (none)
Underline color: Automatic

Effects
- [ ] Strikethrough
- [ ] Double strikethrough
- [ ] Superscript
- [ ] Subscript
- [ ] Shadow
- [ ] Outline
- [ ] Emboss
- [ ] Engrave
- [ ] Small caps
- [ ] All caps
- [ ] Hidden

Preview

Times

This is a scalable printer font. The screen image may not match printed output.

Default... | OK | Cancel

COSC 340: Software Engineering 100

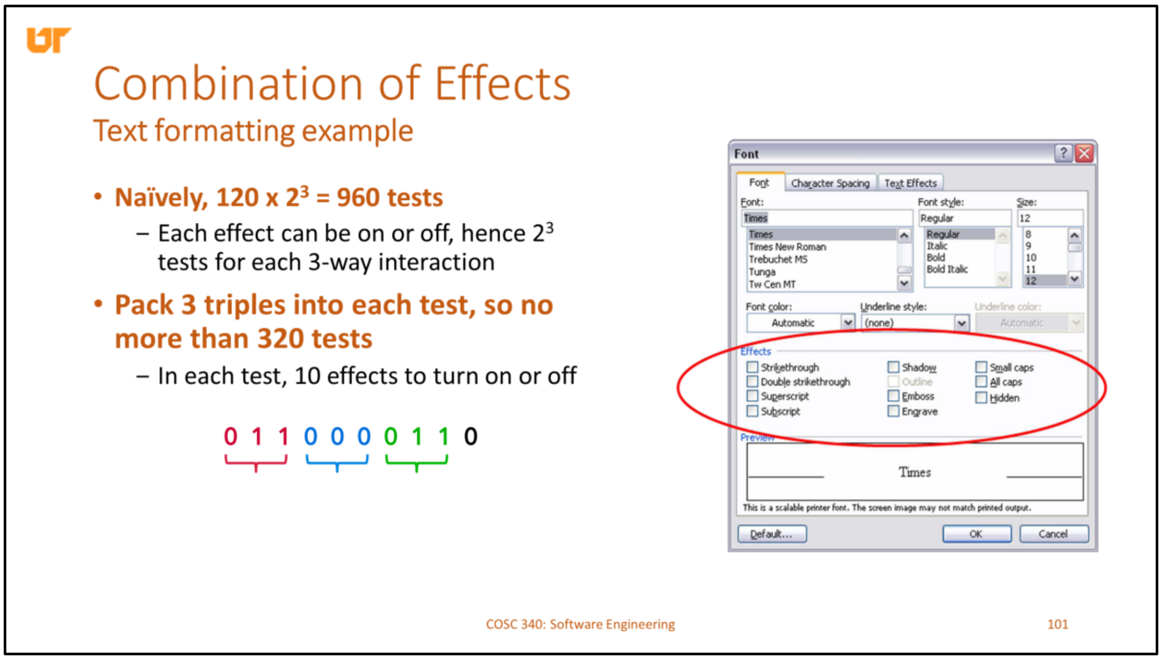
Now, the question is, how many tests would it take to test all 3-way interactions?

This is actually a simple counting problem.

10 ways to pick the first effect, 9 for the second, 8 for the third: 10 × 9 × 8 = 720

But, the order in which you pick an effect does not matter
 3 ways to place the first one (1st, 2nd, or 3rd), 2 for the second
 3 × 2 = 6

So, 720 / 6 = 120

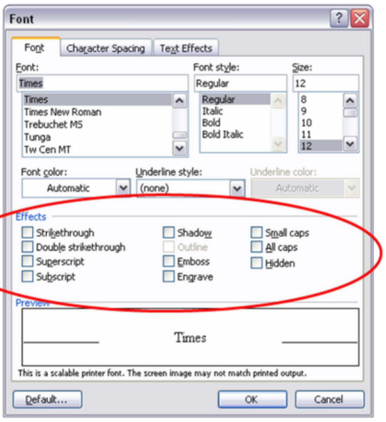


Combination of Effects

Text formatting example

- **Naïvely, 120 x 2^3 = 960 tests**
  - Each effect can be on or off, hence 2^3 tests for each 3-way interaction
- **Pack 3 triples into each test, so no more than 320 tests**
  - In each test, 10 effects to turn on or off

$$\underbrace{0\ 1\ 1}\ \underbrace{0\ 0\ 0}\ \underbrace{0\ 1\ 1}\ 0$$



Font: Times
Font style: Regular
Size: 12
Font color: Automatic
Underline style: (none)
Underline color: Automatic

Effects
- ☐ Strikethrough ☐ Shadow ☐ Small caps
- ☐ Double strikethrough ☐ Outline ☐ All caps
- ☐ Superscript ☐ Emboss ☐ Hidden
- ☐ Subscript ☐ Engrave

This is a scalable printer font. The screen image may not match printed output.

COSC 340: Software Engineering 101

OK – so there are 120 combinations of three effects. So, how many tests do we need to run for these 120 different combinations?

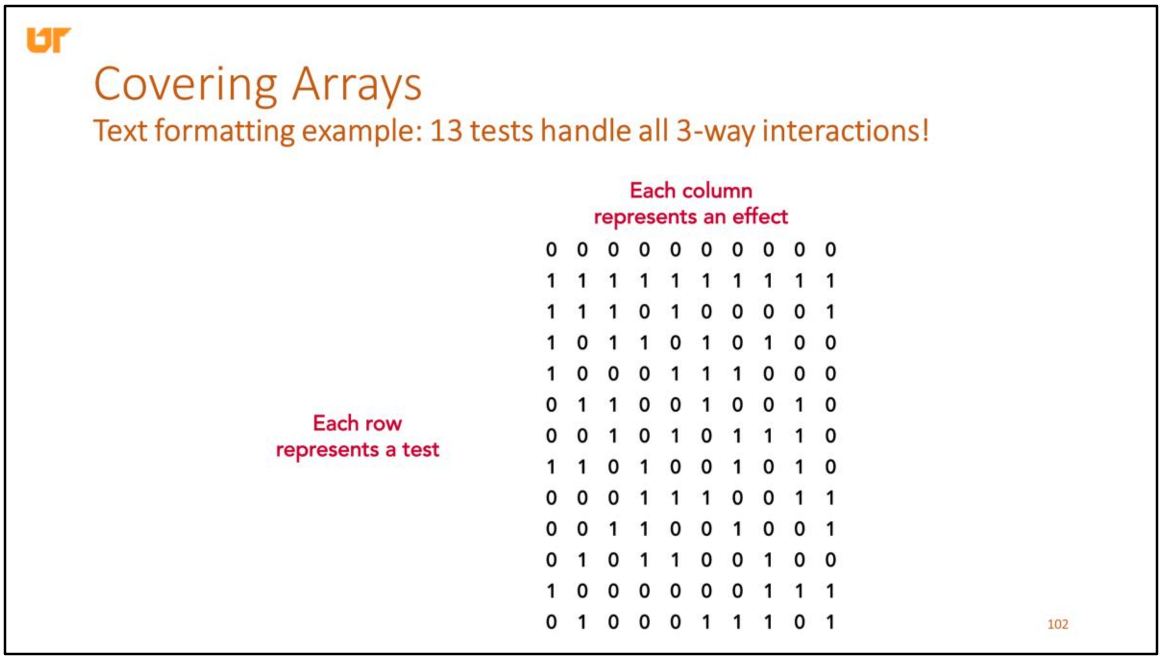
Well, naively we could compute that we need to run 8 tests for each combination of effects. Because each effect can be on or off – there are 2^3 possible tests for each combination of effects, and so we have 120 * 2^3 = 960 tests we need to run.

But, we could easily construct our tests so that different values for different combinations are tested simultaneously. So, basically we have ten possible effects that we can modify for each test, so we pack three triples of effects into each test. Thus, we can divide the number of tests we have to run by 3.

Now, using this same concept, observe that each test can exercise many different triples of parameters.

For instance, if we label each bit as effects a – i, it's clear that this test tests 0 1 1 for the combination a, b, and c, and 0, 0, 0 for d, e, and f, and 0, 1, 1 for g, h, i. But also notice, it tests 1 0 0 for the combination c, d, and e. There are many triples that are packed into this one test.

So, now, the question becomes, what’s the smallest number of tests we need to exhaustively test all 3-way interactions.



Covering Arrays

Text formatting example: 13 tests handle all 3-way interactions!

Each column represents an effect

Each row represents a test

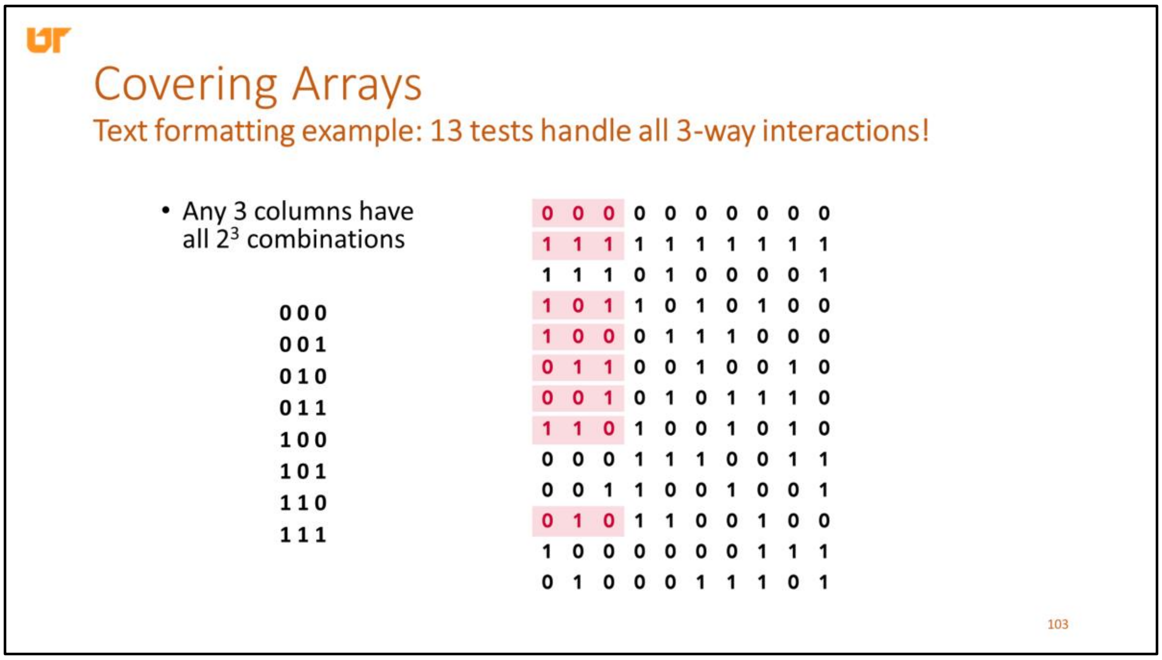
0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

102

A covering array is a way of representing all the tests that cover a set of 3-way interactions.

Each row represents a test. Each column represents an effect.

In this example, only 13 tests are required to handle all 3-way interactions.



Covering Arrays

Text formatting example: 13 tests handle all 3-way interactions!

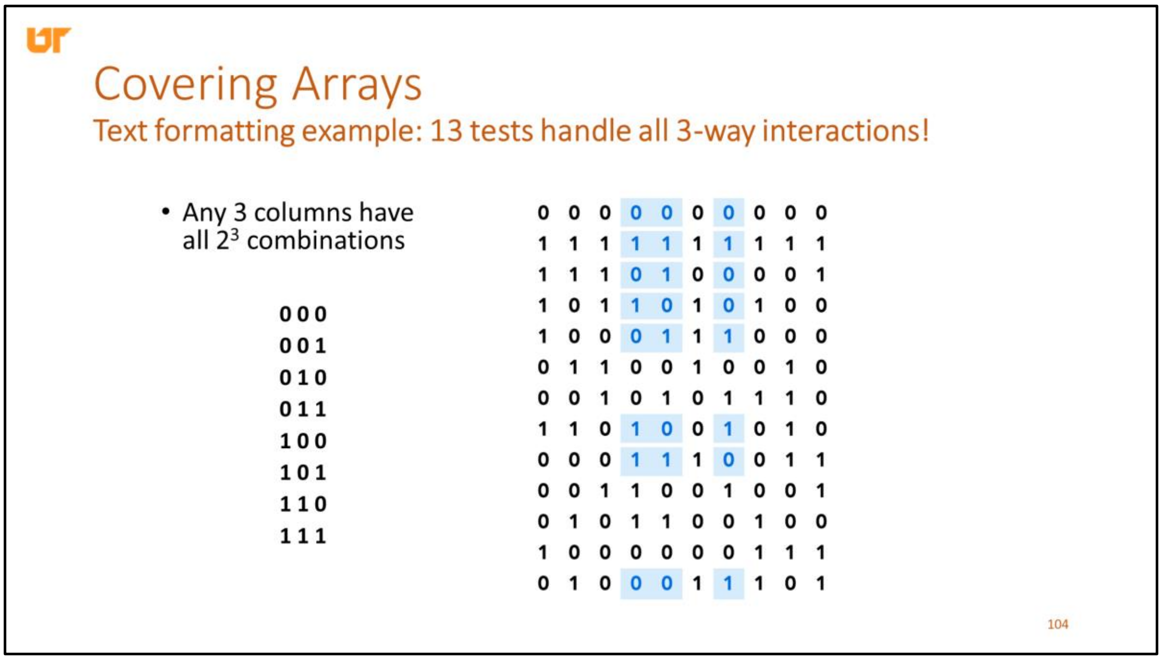
- Any 3 columns have all 2^3 combinations

```
000
001
010
011
100
101
110
111
```

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

103

The only requirement for this covering array is that any three columns contain all possible on/off configurations for the three effects.



Covering Arrays

Text formatting example: 13 tests handle all 3-way interactions!

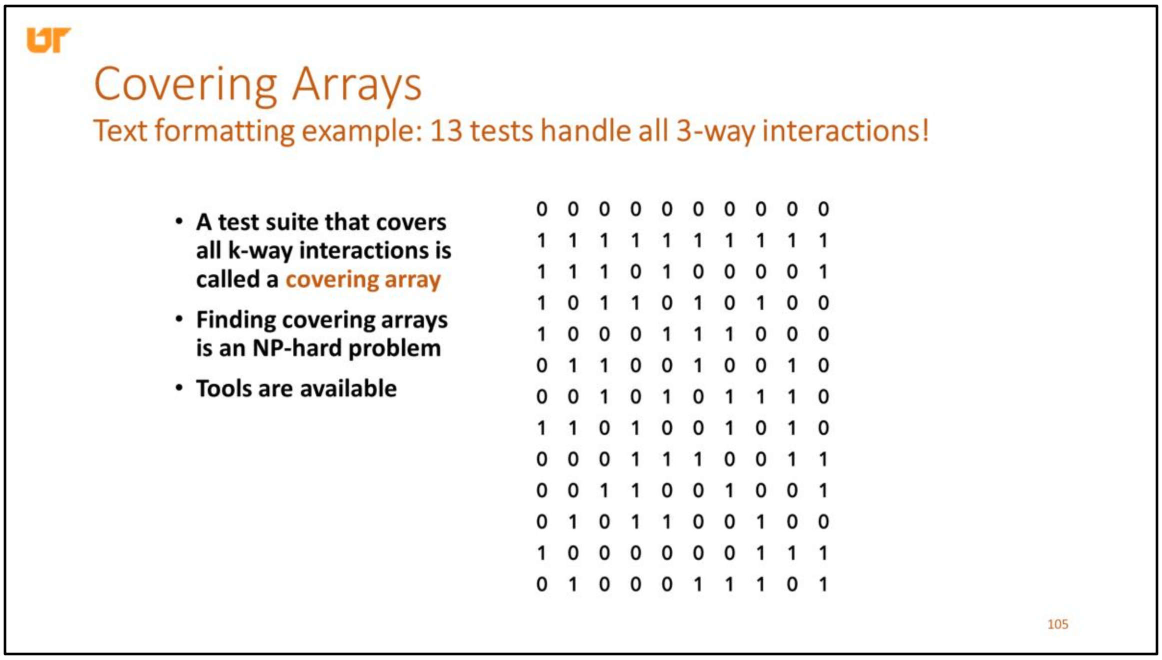
- Any 3 columns have all 2^3 combinations

000
001
010
011
100
101
110
111

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

104

The only requirement for this covering array is that any three columns contain all possible on/off configurations for the three effects.



Covering Arrays

Text formatting example: 13 tests handle all 3-way interactions!

- **A test suite that covers all k-way interactions is called a covering array**
- **Finding covering arrays is an NP-hard problem**
- **Tools are available**

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

105

Unfortunately, finding the minimal covering array is an np-hard problem.

Tools exist that are actually pretty good at finding small covering sets (but may not be optimal).