Black Box Software Testing Foundations
Lecture 1
Overview and Basic Definitions

Cem Kaner J.D., PH.D.

Professor Emeritus, Software Engineering, Florida Institute of Technology

Notice

The practices recommended and discussed in this course are useful for an introduction to testing, but more experienced testers will adopt additional practices. I am writing this course with the mass-market software development industry in mind. Mission-critical and life-critical software development efforts involve specific and rigorous procedures that are not described in this course.

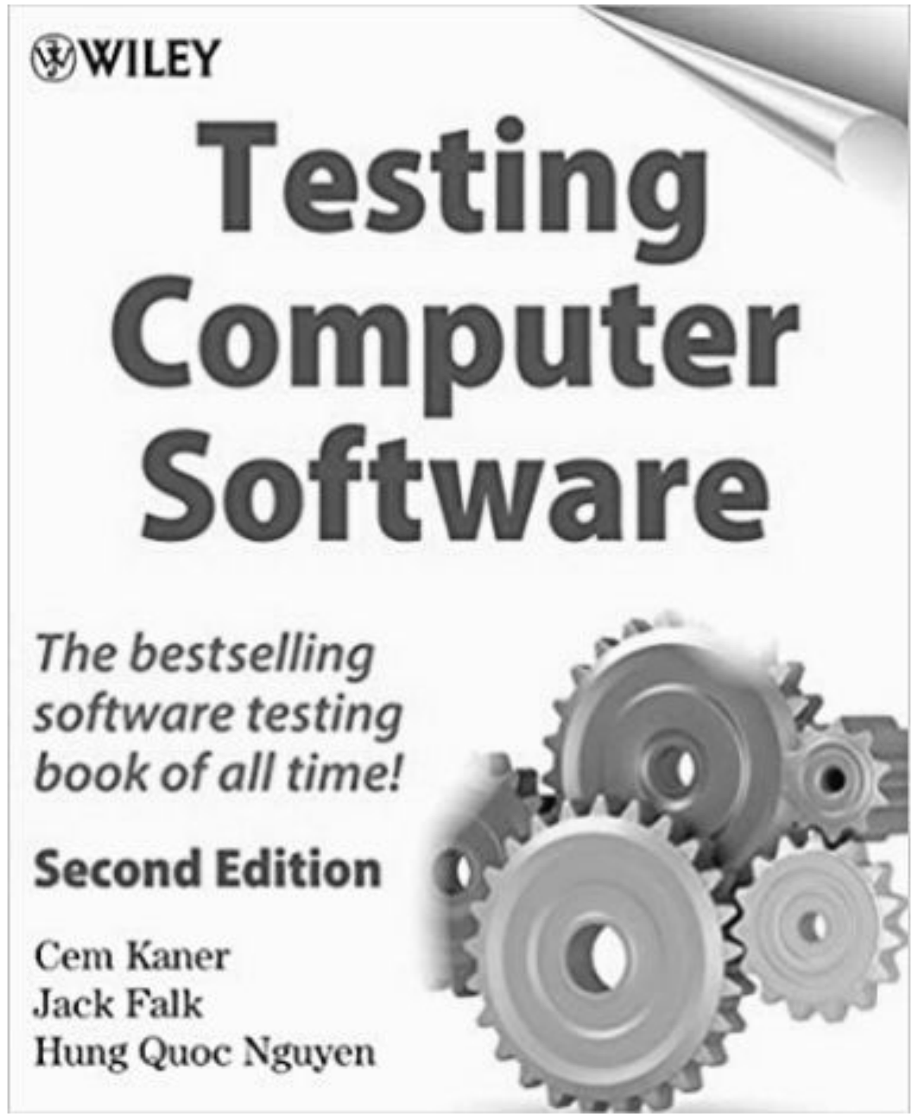
Some of the BBST-series courses include some legal information, but you are not my legal client. I do not provide legal advice in the notes or in the course. If you ask a BBST instructor a question about a specific situation, the instructor might use your question as a teaching tool, and answer it in a way that s/he believes would “normally” be true but such an answer may be inappropriate for your particular situation or incorrect in your jurisdiction. Neither I nor any instructor in the BBST series can accept any responsibility for actions that you might take in response to comments about the law made in this course. If you need legal advice, please consult your own attorney.

About Cem Kaner

https://kaner.com

My job titles are Professor of Software Engineering at the Florida Institute of Technology, and Research Fellow at Satisfice, Inc. I'm also an attorney, whose work focuses on same theme as the rest of my career: satisfaction and safety of software customers and workers. I`ve worked as a programmer, tester, writer, teacher, user interface designer, software salesperson, organization development consultant, as a manager of software testing, user documentation, and software development, and as an attorney focusing on the law of software quality. These have provided many insights into relationships between computers, software, developers, and customers. I studied Experimental Psychology for my Ph.D., with a dissertation on Psychophysics (essentially perceptual measurement). This field nurtured my interest in human factors (usability of computer systems) and the development of useful, valid software metrics. I recently received ACM's Special Interest Group on Computers and Society “Making a Difference” award, which is “presented to an individual who is widely recognized for work related to the interaction of computers and society. The recipient is a leader in promoting awareness of ethical and social issues in computing.”

About Cem Kaner



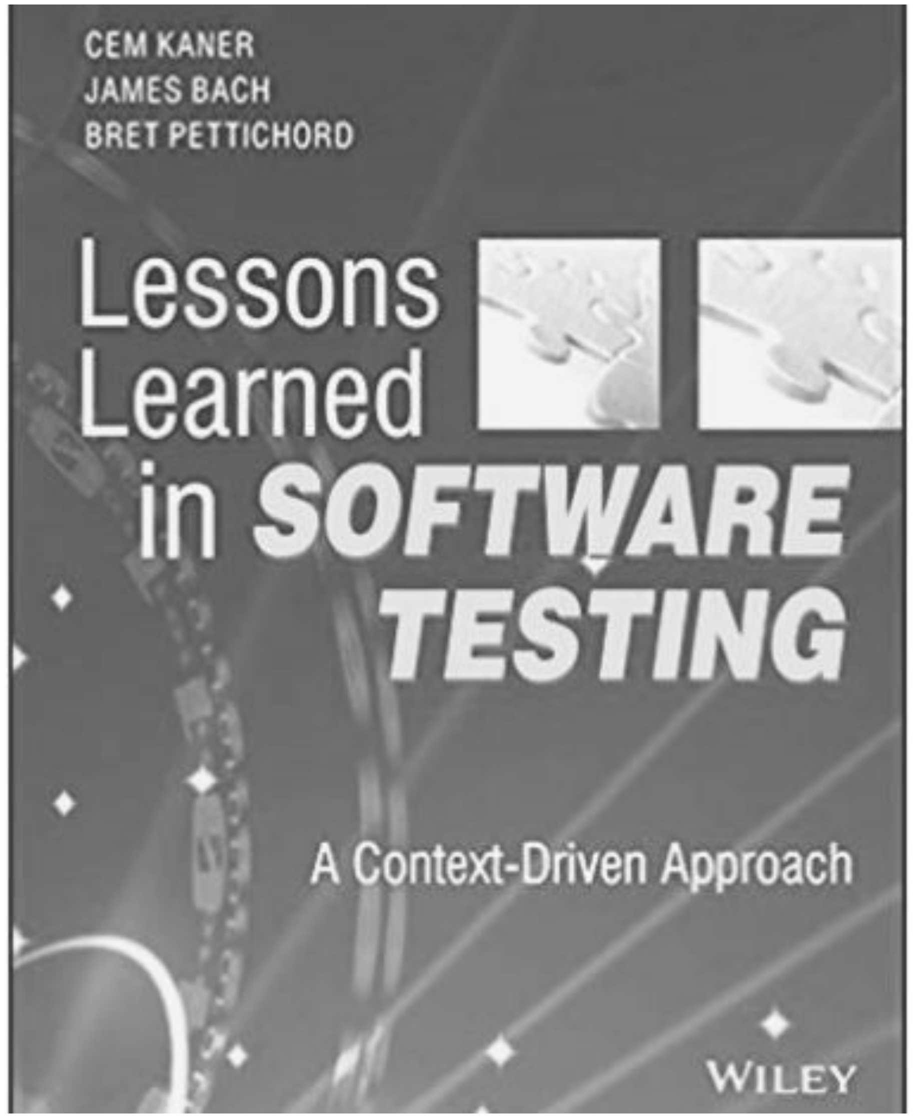
WILEY

Testing Computer Software

The bestselling software testing book of all time!

Second Edition

Cem Kaner
Jack Falk
Hung Quoc Nguyen

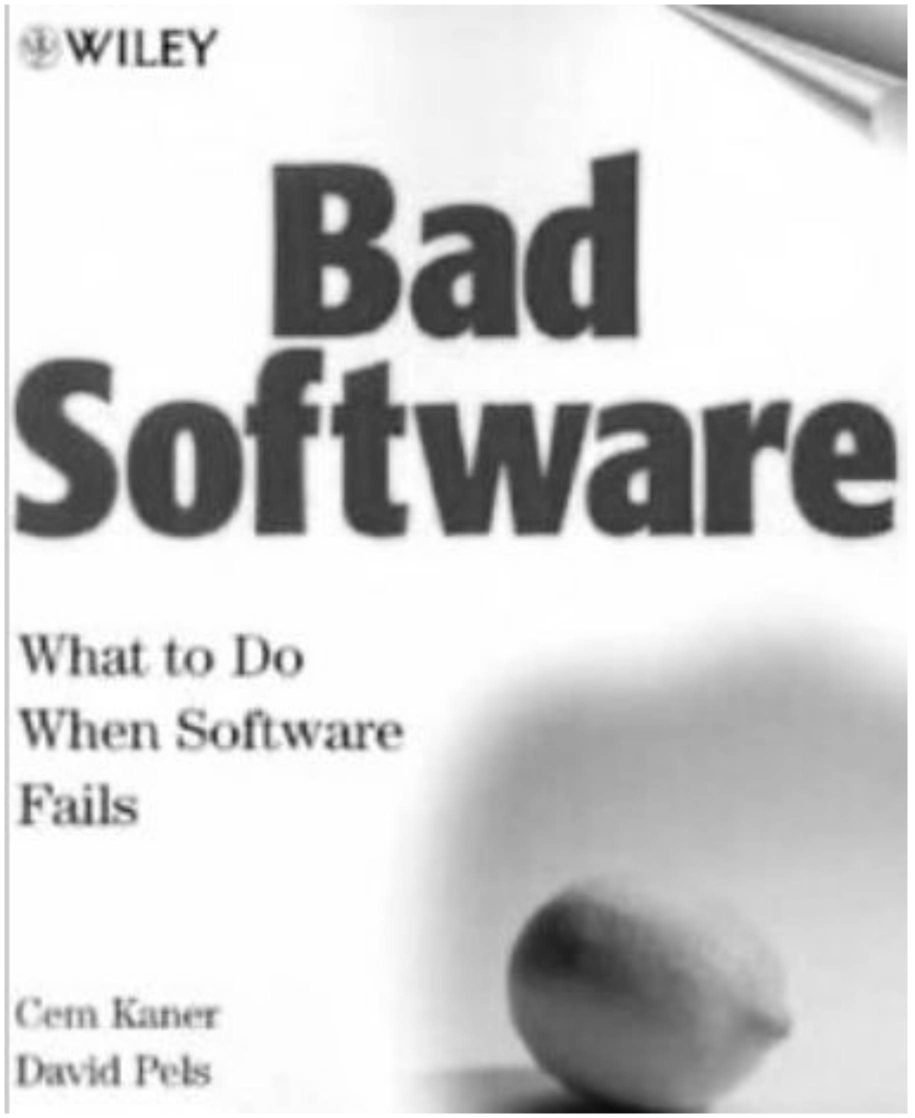


CEM KANER
JAMES BACH
BRET PETTICHORD

Lessons Learned in SOFTWARE TESTING

A Context-Driven Approach

WILEY



WILEY

Bad Software

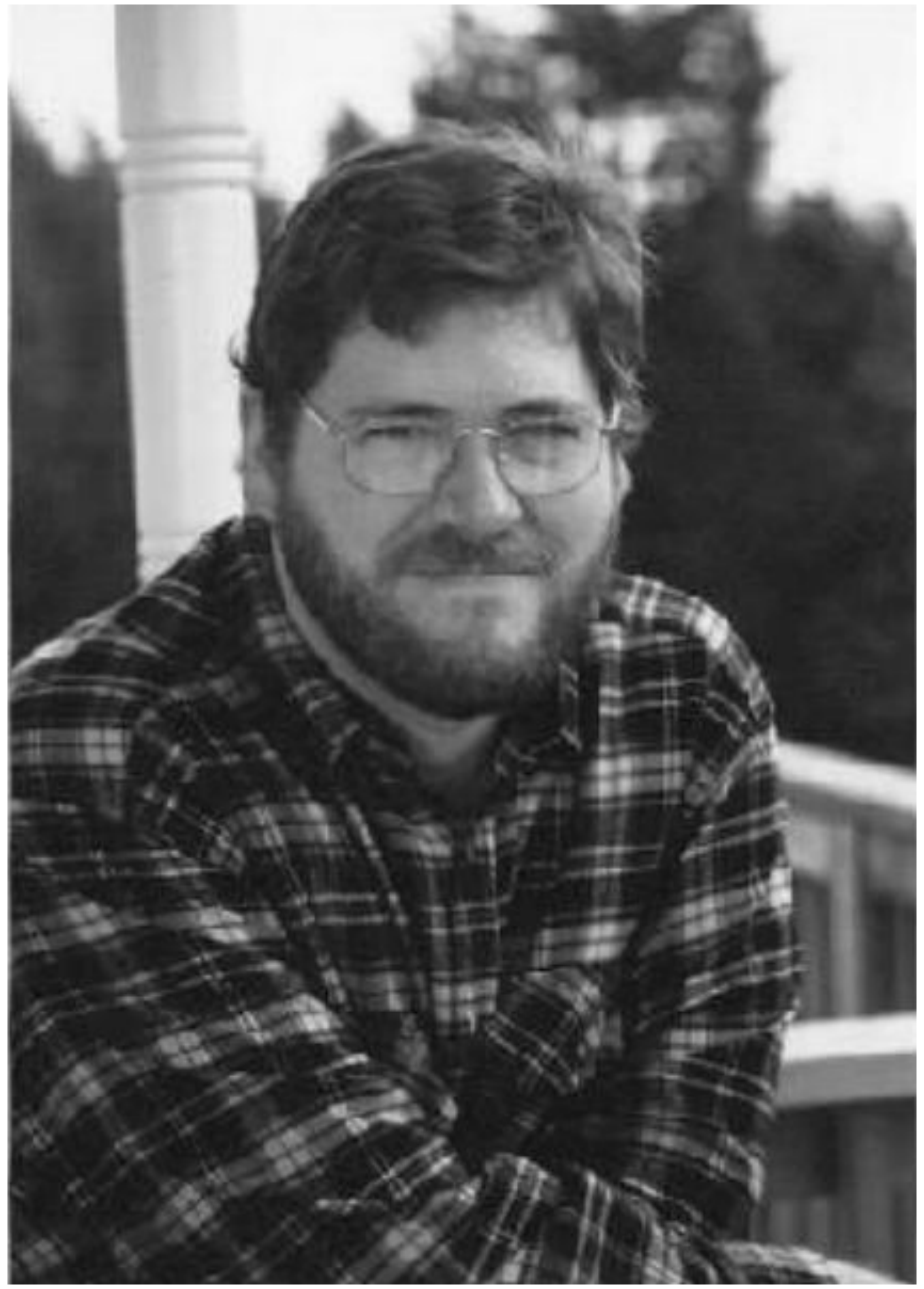
What to Do
When Software
Fails

Cem Kaner
David Pels

About James Bach

www.satisfice.com

I started in this business as a programmer. I like programming. But I find the problems of software quality analysis and improvement more interesting than those of software production. For me, there's something very compelling about the question "How do I know my work is good?" Indeed, how do I know anything is good? What does good mean? That's why I got into SQA, in 1987. Today, I work with project teams and individual engineers to help them plan SQA, change control, and testing processes that allow them to understand and control the risks of product failure. I also assist in product risk analysis, test design, and in the design and implementation of computer-supported testing. Most of my experience is with market-driven Silicon Valley software companies like Apple Computer and Borland, so the techniques I've gathered and developed are designed for use under conditions of compressed schedules, high rates of change, component- based technology, and poor specification.



About Rebecca L. Fiedler

I've been teaching students of all ages – from Kindergarten to University – for the past 25 years. My primary interests are how people learn and how technology can make educational efforts more effective and more accessible to more people. Until recently, I served as an Assistant Professor of Education at Indiana State University and St. Mary-of-the- Woods College, but to really get to the roots of effective design of online education, especially for working professionals, it made more sense for me to go independent and focus my own time as an independent consultant. I consult primarily through Acclaro Research Solutions. Cem Kaner and I are co-Principal Investigators on the National Science Foundation grant that subsidizes development of these courses. My Ph.D. (University of Central Florida) concentrations were in Instructional Technology and Curriculum. My dissertation research applied qualitative research methods to the use of electronic portfolios. I also hold an M.B.A. in Management and a Bachelor of Music (Education).

Many Thanks...

The BBST lectures evolved out of courses co-authored by Kaner & Hung Quoc Nguyen and by Kaner & Doug Hoffman, which we merged with James Bach's and Michael Bolton's Rapid Software Testing (RST) courses. The online adaptation of BBST was designed primarily by Rebecca L. Fiedler.

After being developed by practitioners, the course evolved through academic teaching and research largely funded by the National Science Foundation. The Association for Software Testing served as our learning lab for practitioner courses. We also evolved the 4-week structure with AST. We could not have created this series without AST's collaboration. Since 2014, Altom has been offering the course commercially. Starting with 2019, Altom has been maintaining and updating the course materials.

Many Thanks...

We also thank Jon Bach, Scott Barber, Bernie Berger, Ajay Bhagwat, Rex Black, Jack Falk, Elizabeth Hendrickson, Kathy Iberle, Bob Johnson, Karen Johnson, Brian Lawrence, Brian Marick, John McConda, Melora Svoboda, dozens of participants in the Los Altos Workshops on Software Testing, the Software Test Managers’ Roundtable, the Workshops on Heuristic & Exploratory Techniques, the Workshops on Teaching Software Testing, the Austin Workshops on Test Automation and the Toronto Workshops on Software Testing and students in AST and Altom courses for critically reviewing materials from the perspective of experienced practitioners.

We also thank the many students and co-instructors at Florida Tech who helped us evolve the academic versions of this course, especially Pushpa Bhallamudi, Walter P. Bond, Tim Coulter, Sabrina Fay, Ajay Jha, Alan Jorgenson, Kishore Kattamuri, Pat McGee, Sowmya Padmanabhan, Andy Tinkham, and Giri Vijayaraghavan.

We also thank all instructors, practitioners and Altom employees who contribute to updating and developing new content for this course series, especially Ancuța Bodnărescu, Alexandra Casapu, Oana Casapu, Ru Cindrea, Gabriel Dobrițescu, Zoltán Molnár, Ray Oei, and Dolores Pente.

Our Approach

Testing software involves investigating a product under tight constraints. Our goal is to help you become a better investigator:

- **Knowledge and skills** important to testing practitioners
- **Context-driven**
  - Diverse contexts call for diverse practices.
  - We don't teach "best practices." Instead, we teach practices that are useful in the appropriate circumstances.

See https://kaner.com/?p=49

BBST Learning Objectives

- Understand key testing challenges that demand thoughtful tradeoffs by test designers and managers
- Develop skills with several test techniques
- Choose effective techniques for a given objective under your constraints
- Improve the critical thinking and rapid learning skills that underlie good testing
- Communicate your findings effectively
- Work effectively online with remote collaborators
- Plan investments (in documentation, tools, and process improvement) to meet your actual needs
- Create work products that you can use in job interviews to demonstrate testing skill

Foundations Course Objectives

Learning about testing	Improving academic skills
● Key challenges of testing: ○ Information objectives drive the testing mission and strategy ○ Oracles are heuristic ○ Coverage is multidimensional ○ Complete testing is impossible ○ Measurement is important, but hard ● Introduce you to: ○ Basic vocabulary of the field ○ Basic facts of data storage and manipulation in computing ○ Diversity of viewpoints ○ Viewpoints drive vocabulary	● Online collaboration tools: ○ Forums ○ Wikis ● Precision in reading ● Clear, well-structured communication ● Effective peer review ● Cope calmly and effectively with formative assessments (such as tests designed to help you learn) ○ Assessment can be helpfully hard without being risky

Instructional Approach

Every aspect of the course is designed to help you learn:

- Orientation exercises (readiness for learning)
- Video lectures
- Quizzes
- Labs and assignments (tasks that apply learning)
- Social interactions
- Peer reviews
- Study-guide driven exams
    - Exam prep forum
- Exam with an optional Interactive Grading session

Course Overview: Fundamental Topics

1. The Nature of Testing
   Overview and Basic Definitions ⬅

2. Why are we testing? What are we trying to learn? How should we organize our work to achieve this? ***Information objectives drive the testing mission and strategy***

3. How can we know whether a program has passed or failed a test?
   Oracles are heuristic

4. How can we determine how much testing has been done? What core knowledge about program internals do testers need to consider this question?
   Coverage is a multidimensional problem

5. Are we done yet?
   Complete testing is impossible

6. How much testing have we completed and how well have we done it?
   Measurement is important but hard

What’s a Computer Program

Textbooks often define a “computer program” like this:

- A program is a set of instructions for a computer

What’s a Computer Program

That’s like defining a house like this:

- A house is a set of construction materials assembled according to house-design patterns.

We’d rather define it as:

- A house is something built for people to live in.

This second definition focuses on the purpose and stakeholders of the house, rather than on its materials.

What’s a Computer Program

A house is something built for people to live in.

The focus is on:

- Stakeholders (for people)
- Purpose (to live in)

Stakeholder

Any person affected by:

- **success or failure of a project,**
- **actions or inactions of a product,**
- **effects of a service.**

What’s a Computer Program

The narrow focus on the machine prepares Computer Science students to make the worst errors in software engineering — in their first two weeks of school.

What's a Computer Program

A different definition for a computer program is:

- a communication
- among several humans and computers
- who are distributed over space and time,
- that contains instructions that can be executed by a computer.

The point of the program is to provide value to the stakeholders.

What Are We Really Testing For?

Quality is value to some person - Jerry Weinberg

- Quality is inherently subjective.
- Different stakeholders will perceive the same product as having different levels of quality.
- Testers look for different things ... for different stakeholders.

Software Error (AKA Bug)

An attribute of a software product:

- **that reduces its value to a favored stakeholder**
- **or increases its value to a disfavored stakeholder**
- **without a sufficiently large countervailing benefit.**

An error:

- May or may not be a coding error, or a functional error

Design errors are bugs too.

Software Testing

- is an empirical
- technical
- investigation
- conducted to provide stakeholders
- with information
- about the quality
- of the product or service under test.

We design and run tests in order to gain useful information about the product's quality.

Testing Is Always a Search for Information

- Find important bugs
- Assess the quality of the product
- Help managers assess the progress of the project
- Help managers make release decisions
- Block premature product releases
- Help predict and control product support costs
- Check interoperability with other products
- Find safe scenarios for use of the product
- Assess conformance to specifications
- Certify the product meets a particular standard
- Ensure the testing process meets accountability standards
- Minimize the risk of safety-related lawsuits
- Help clients assess their product's quality and testability
- Help clients evaluate their processes and suggest/assess ways to improve them
- Evaluate the product for a third party

Different objectives require different testing tools and strategies and will yield different tests, test documentation and test results.

There Are No “Correct” Definitions

- Some people don’t like our definition of testing
  - They would rather call testing a hunt for bugs
  - Or a process for verifying that a program meets its specification

Try to reconcile THOSE two definitions!

- The different definitions reflect different visions of testing.
- Meaning is not absolute. Words mean what the people who say them intend them to mean and what the people who hear them interpret them as meaning.
- Clear communication requires people to share definitions of the terms they use. If you're not certain that you know what someone else means, **ask them**.

We would rather embrace the genuine diversity of our field than try to use standards committees to hide it or legislate it away.

We Use “Working Definitions”

- We provide definitions for key concepts:
  - To limit ambiguity, and
  - To express clearly the ideas in our courses
- And we expect you to learn them.
- And we will test you on them. And give you bad test grades if you get them “wrong.”
- We DON’T require you to accept our definitions as correct
  OR as the most desirable definitions.
- We only require you to demonstrate that you understand what we are saying.
- In the “real world,” when someone uses one of these words,
  ask them what THEY mean instead of assuming that they mean what WE mean.

We welcome critical discussion in our forums.

Black Box Testing

Testing and test design without knowledge of the code (or without use of knowledge of the code).

The tester designs tests from his (research-based) knowledge of the product's user characteristics and needs, the subject area (e.g. "insurance"), the product's market, risks, and environment (hardware/ software).

Some authors narrow this concept to testing exclusively against an authoritative specification. (We don't.)

The black box tester becomes an expert in the relationships between the program and the world in which it runs.

Glass Box Testing

Testing and test design using knowledge of the details of the internals of the program (code and data).

Glass box testers typically ask:

- “Does this code do what the programmer expects or intends?”

In contrast to the black box question:

- “Does this do what the users (human and software) expect?”

Glass box is often called “white box” to contrast with “black box.”

The glass box tester becomes an expert in the implementation of the product under test.

Grey Box Testing?

People often ask us, “If there is black box testing and white box testing, what is grey box testing?”

We don’t think there is a standard definition. “A blend of black box and white box approaches” is not very informative. Examples of grey box:

- Studying variables that are not visible to the end user (e.g. log file analysis or performance of subsystems)
- Designing tests to stress relationships between variables that are not visible to the end user

Search the web for more examples of “grey box testing” descriptions. There are thousands.

Are These "Techniques"?

Are "black box" or "glass box" test techniques?

We prefer to call them "approaches."

Dictionary.com defines "technique" as:

- "The body of specialized procedures and methods used in any specific field, esp. in an area of applied science.
- Method of performance; way of accomplishing."

When someone says they'll do "black box testing," you don't know what they'll actually **do,** what tools they'll use, what bugs they'll look for, how they'll look for them, or how they'll decide whether they've found a bug. Some techniques are more likely to be used in a black box way, so we might call these "black box techniques." But it is the technique ("usability testing") that is black box, not "black box" that is the technique.

Behavioral Testing

Behavioral testing is focused on the observable behavior of the product.

Behavioral testing is useful when our purpose is to verify that the program does what the programmer intended.

Structural Testing

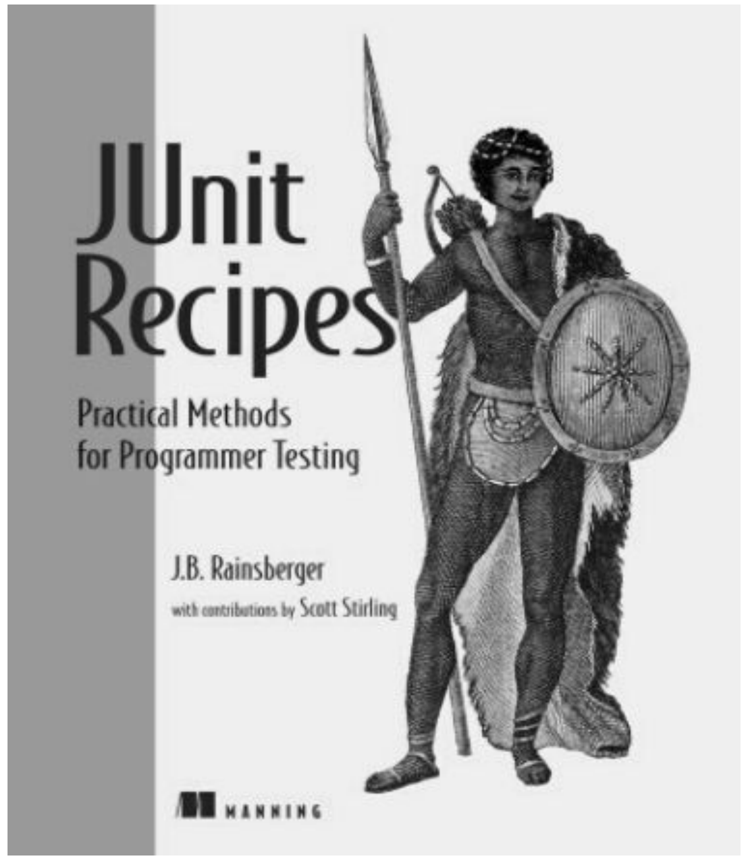
As far as we can tell,

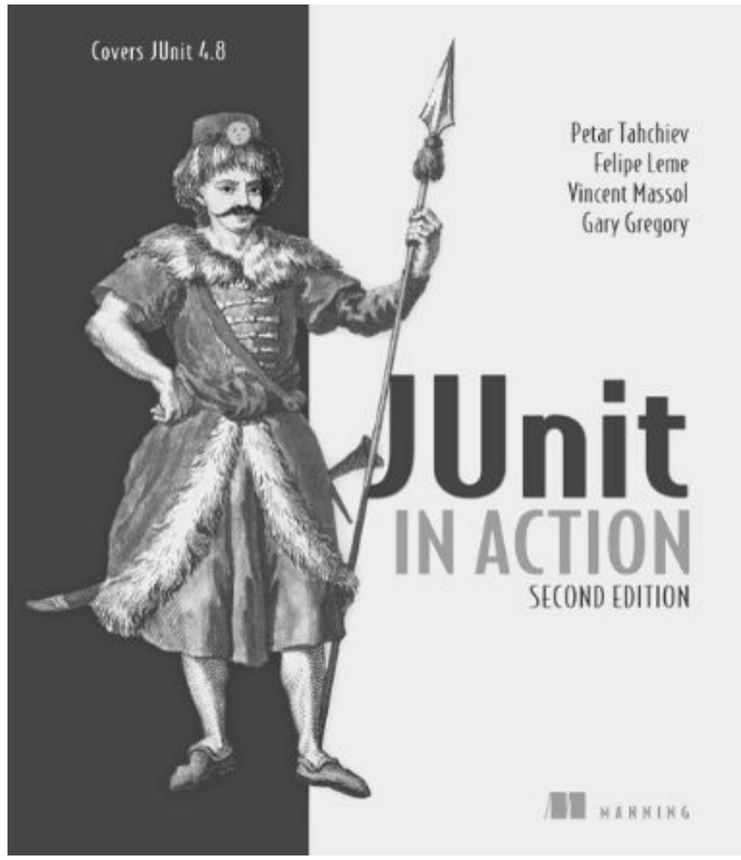
structural testing is the same as glass box testing.

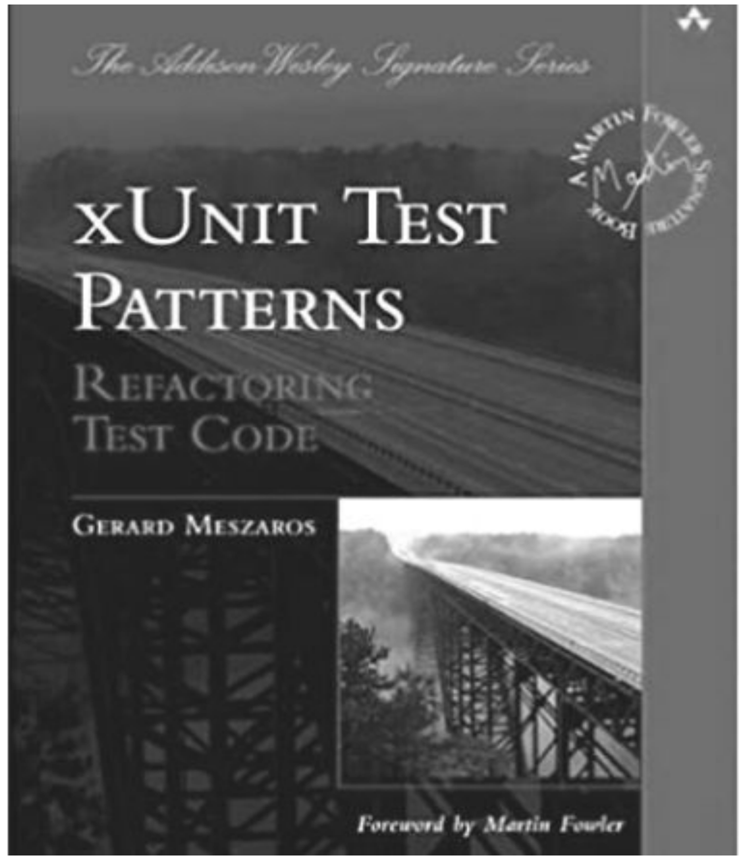
Unit, Integration & System Testing

Unit tests focus on individual units of the product.

Programmers typically describe unit testing as glass box testing, focused on individual methods (functions) or individual classes.


JUnit Recipes
Practical Methods for Programmer Testing
J.B. Rainsberger
with contributions by Scott Stirling
MANNING




The Addison-Wesley Signature Series
A Martin Fowler Signature Book
xUnit Test Patterns
Refactoring Test Code
Gerard Meszaros
Foreword by Martin Fowler

Unit, Integration & System Testing

IEEE standard 1008 on software unit testing clarifies that a unit

- "may occur at any level of the design hierarchy from a single module to a complete program."

If you think of it as one thing, and test it as one thing, it's a unit.

Black box unit tests?

Imagine a code library that specifies the interfaces of its functions but provides no source code.

How does the programmer try out a function?

Unit, Integration & System Testing

Integration tests study how two (or more) units work together.

You can have:

- low-level integration (2 or 3 units) and
- high-level integration (many units, all the way up to tests of the complete, running system).

Integration testing might be black box or glass box. Integration testers often use knowledge of the code to predict and evaluate how data flows among the units.

Unit, Integration & System Testing

System testing focuses on the value of the running system.

"System testing is the process of attempting to demonstrate how the program does not meet its objectives"

See Glen Myers (1979), *The Art of Software Testing,* p. 110

Implementation-Level vs. System-Level

Implementation-level testing is focused on the details of the implementation.

- Typically it is glass box testing.
- Examples include unit tests, integration tests, tests of dataflows, and tests of performance of specific parts of the program. These are all implementation-level tests.
- Typically, implementation-level tests ask whether the program works as the programmer intended or whether the program can be optimized in some way.

Functional & Parafunctional

Functional testing is system-level testing that looks at the program as a collection of functions. A “function” might be an individual feature or a broader capability that relies on several underlying features.

We analyze a function in terms of the inputs we can provide it and the outputs we would expect, given those inputs.

See W.E. Howden, *Functional Program Testing & Analysis*, 1987

Functional & Parafunctional

In contrast to "functional testing", people often refer to **parafunctional** or **nonfunctional** testing. (Why parafunctional instead of nonfunctional? Calling tests "nonfunctional" forces absurd statements, like "all the nonfunctional tests are now working…")

The concept of "functional testing" is fairly well defined, but parafunctional includes anything "other than" ("para") functional. Same for non-functional.

This includes testing attributes of the software that are general to the program rather than tied to any particular function, such as usability, scalability, maintainability, security, speed, localizability, supportability, etc.

The concept of parafunctional (or non-functional) testing is so vague as to be dysfunctional.

We won't often use it in the course.

Acceptance Testing #1

In early times, most software development was done under contract. A customer (e.g. the government) hired a contractor (e.g. IBM) to write a program. The customer and contractor would negotiate the contract. Eventually the contractor would say that the software is done and the customer or her agent (such as an independent test lab) would perform acceptance testing.

Acceptance testing determines whether the software developed under a contract should be accepted by the customer.

If software failed the tests, it was unacceptable and the customer would refuse to pay for it until the software was made to conform to the promises in the contract (which were what was checked by the acceptance tests).

This is the meaning we will adopt in this course.

Acceptance Testing #2

There really is no place for acceptance testing if there are no contract-based requirements. (At least, not in the traditional sense of the word.) But many people use the word anyway.

To them, “acceptance testing” refers to tests that might help someone decide whether a product is ready for sale, installation on a production server, or delivery to a customer. To us, this describes a developer’s decision (whether to deliver) rather than a customer’s decision (whether to accept), so we won’t use this term this way. However, it is a common usage, with many local variations. Therefore, far be it from us to call it “wrong.” But when you hear or read about “acceptance testing”, don’t assume you know what meaning is intended. Check your local definition.

Bolton, http://www.developsense.com/presentations/2007-10-PNSQC-UserAcceptanceTesting.pdf

Independent Testing

Testing done by a third party, often an external test lab.

Some companies have an independent in-house test group.

The key notion is that the independent testers aren’t influenced or pressured to analyze and test the software in ways preferred by the developers.

Independent labs might be retained to do any type of testing, such as functional testing, performance testing, security testing, etc.

Quiz Standards, Rules & Tips

- BBST Quizzes are OPEN BOOK
- You are welcome to take the quiz while you watch the video or read the materials.
- You can take the quiz with a friend (sit side by side or skype together)
- You may not copy someone else's answers. If you use someone else's answer without figuring out yourself what the answer is, or working it out with a partner (and actively engage in reasoning about it with your partner), you are cheating.
    - If you make an honest effort on the quizzes but score poorly, don't panic. The scores are for your feedback and to tell us who is *trying* to make progress in the course. No one who has honestly attempted the quizzes has ever failed the course because of low quiz grades.

Quiz Standards, Rules & Tips

The quizzes are designed to help you determine how well you understand the lecture or the readings and to help you gain new insights from lecture/readings.

- We will make fine distinctions. (If you're not sure of the answer, go back and read again or watch the video)
- We will demand precise reading. (The ability to read carefully, make distinctions, and recognize and evaluate inferences in what is read, is essential for analyzing specifications. All testers need to build these skills.)
- We will sometimes ask you to think about a concept and work to a conclusion.

It is common for students to learn new things while they take the quiz.

Quiz Standards, Rules & Tips

- Typical question has 7 alternatives:

  a. (a)

  b. (b)

  c. (c)

  d. (a) and (b)

  e. (a) and (c)

  f. (b) and (c)

  g. (a) and (b) and (c)

- Score is 25% if you select one correct of two (e.g. answer (a) instead of (d).)
- Score is 0 if you include an error (e.g. answer (d) when right answer is only (a).) People usually remember the errors they hear from you more than they notice what you omitted to say.

Sample Quiz Question

What is the significance of the difference between black box and glass box tests?

a. Black box tests cannot be as powerful as glass box tests because the tester doesn't know what issues in the code to look for.

b. Black box tests are typically better suited to measure the software against the expectations of the user, whereas glass box tests measure the program against the expectations of the programmer who wrote it.

c. Glass box tests focus on the internals of the program whereas black box tests focus on the externally visible behavior.

d. (a) and (b)

e. (a) and (c)

f. (b) and (c)

g. (a) and (b) and (c)

Sample Quiz Question

What is the significance of the difference between black box and glass box tests?

✓ b. Black box tests are typically better suited to measure the software against the expectations of the user, whereas glass box tests measure the program against the expectations of the programmer who wrote it.

c. Glass box tests focus on the internals of the program whereas black box tests focus on the externally visible behavior.

  ○ This is factually correct, but irrelevant. The question doesn’t ask what the difference is between black box and glass box. It asks “What is the **significance** of the difference?”

These might seem unfairly hard to begin with, but you’ll get better at them with practice.

The underlying skills have value.

Sample Quiz Question

According to the lecture, independent testing...

a. must be done by an outside company.

b. is a form of black box testing that is typically done by an outside test lab.

c. is typically done by an outside company (test lab) but can be done in-house if the testers are shielded from influence by the development staff.

d. (a) and (b)

e. (a) and (c)

f. (b) and (c)

g. (a) and (b) and (c)

Sample Quiz Question

According to the lecture, independent testing...

b. is a form of black box testing that is typically done by an outside test lab.

- ○ Answer (b) might be correct under other definitions of independent testing but not in the definition “according to the lecture.”
- ○ The lecture includes any type of testing, as long it is independent.

✓ c. is typically done by an outside company (test lab) but can be done in-house if the testers are shielded from influence by the development staff.

Real-life example: electronic voting systems are subject to code review and glass box testing by independent test labs.

About the Exam

- Our exams are closed book, essay style.
- We focus students' work with essay questions in a study guide. We draw all exam questions from this guide.
- We expect well-reasoned, well-presented answers. This is the tradeoff. You have lots of time before the exam to develop answers. On the exam, we expect good answers.
- We encourage students to develop answers together.
- Please don't try to memorize other students' answers instead of working on your own. It's usually ineffective (memorization errors lead to bad grades) and you end up learning very little from the course.
- Please don't post study guide questions and suggested answers on public websites. That encourages students (in other courses) to memorize your answers instead of developing their own. Even if someone could memorize all your answers perfectly, and all your answers were perfect, this would teach them nothing about testing. It would cheat them of the educational value of the course.

Black Box Software Testing Foundations
Lecture 2
Strategy

Cem Kaner J.D., PH.D.

Professor Emeritus, Software Engineering, Florida Institute of Technology

Course Overview: Fundamental Topics

1. The Nature of Testing
   Overview and Basic Definitions

2. Why are we testing? What are we trying to learn? How should we organize our work to achieve this? ***Information objectives drive the testing mission and strategy*** ←

3. How can we know whether a program has passed or failed a test?
   Oracles are heuristic

4. How can we determine how much testing has been done? What core knowledge about program internals do testers need to consider this question?
   Coverage is a multidimensional problem

5. Are we done yet?
   Complete testing is impossible

6. How much testing have we completed and how well have we done it?
   Measurement is important but hard

Today's Readings

Required

- Cem Kaner, Elisabeth Hendrickson & Jennifer Smith-Brock (2001), "Managing the Proportion of Testers to (Other) Developers", https://kaner.com/pdfs/pnsqc_ratio_of_testers.pdf

Useful to skim

- James Bach, "The Heuristic Test Strategy Model", www.satisfice.com/tools/satisfice-tsm-4p.pdf
- Cem Kaner (2000), "Recruiting Software Testers", https://kaner.com/pdfs/JobsRev6.pdf
- Jonathan Kohl (2010), "How Do I Create Value With My Testing?", www.kohl.ca/blog/archives/000217.html
- Karl Popper (2002, 3rd Ed.), *"Conjectures and Refutations: The Growth of Scientific Knowledge (Routledge Classics)"*

What Is Software Testing?

Testing is:

- "the process of executing a program with the intent of finding errors." Glen Myers (1979, p. 5), *Art of Software Testing*
- "questioning a product in order to evaluate it." - James Bach

Some definitions are simple and straightforward.

What Is Software Testing?

"The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component." (IEEE standard 610.12-1990)

"Any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results... Testing is the measurement of software quality." Bill Hetzel (1988, 2nd ed., p. 6), *Complete Guide to Software Testing*.

Other definitions are a little more complex.

Software Testing

- is an empirical
- technical
- investigation
- conducted to provide stakeholders
- with information
- about the quality
- of the product or service under test

Defining Testing

An empirical

- We gain knowledge from the world, not from theory. (We call our experiments, “tests.”)
- We gain knowledge from many sources, including qualitative data from technical support, user experiences, etc.

technical

- We use technical means, including experimentation, logic, mathematics, models, tools (testing-support programs), and tools (measuring instruments, event generators, etc.)

Defining Testing

An empirical, technical…

… investigation

- An organized and thorough search for information
- This is an active process of inquiry. We ask hard questions (aka run hard test cases) and look carefully at the results.

conducted to provide stakeholders

- Someone who has a vested interest in the success of the testing effort
- Someone who has a vested interest in the success of the product

A law firm suing a company for having shipped defective software has no interest in the success of the product development effort but a big interest in the success of its own testing project (researching the product's defects).

Defining Testing

An empirical, technical investigation conducted to provide stakeholders…

…with information

- The information of interest is often about the presence (or absence) of bugs, but other types of information are sometimes more vital to your particular stakeholders.
- In information theory, "information" refers to reduction of uncertainty. A test that will almost certainly give an expected result is not expected to (and not designed to) yield much information.

Karl Popper argued that experiments designed to confirm an expected result are of far less scientific value than experiments designed to disprove (refute) the hypothesis that predicts the expectation.
See his enormously influential book, *Conjectures & Refutations.*

Defining Testing

An empirical, technical investigation conducted to provide stakeholders with information...

... about the quality

- Value to some person

of the product or service under test

- The product includes the data, documentation, hardware, whatever the customer gets. If it doesn't all work together, it doesn't work.
- A service (such as custom programming) often includes sub-services (such as support).
- Most software combines product & service.

Many Different Information Objectives

- Find important bugs
- Assess the quality of the product
- Help managers assess the progress of the project
- Help managers make release decisions
- Block premature product releases
- Help predict and control product support costs
- Check interoperability with other products
- Find safe scenarios for use of the product
- Assess conformance to specifications
- Certify the product meets a particular standard
- Ensure the testing process meets accountability standards
- Minimize the risk of safety-related lawsuits
- Help clients assess their product's quality and testability
- Help clients evaluate their processes and suggest/assess ways to improve them
- Evaluate the product for a third party

Different objectives require different testing tools and strategies and will yield different tests, test documentation and test results.

Many Different Information Objectives

Different Context	→	Different Information Objectives
Mass-market software, close to release date. The test group believes the product is too buggy and that better-informed stakeholders wouldn't ship it.	→	These testers are likely to do bug-hunting, looking for important bugs that will cause key whether they are willing to release the product.
Software fails in use and causes serious losses. A law firm hires testers to determine what caused the failures and when the seller found these bugs.	→	These testers won't do general bug-hunting. They'll try to determine how (and in how many ways) they can replicate specific failures and they'll study corporate quality records.

Your Testing Mission

Your “mission” is your answer to the question, “Why are you testing?”

- Typically, your mission is to achieve your primary information objective(s).
  - If there are too many objectives, you have a fragmented, and probably unachievable, mission.
  - Awareness of your mission helps you focus your work. Tasks that help you achieve your mission are obviously of higher priority (or should be) than tasks that don’t help you achieve your mission.

Your Testing Mission(s)

- The test group's mission probably changes over the course of the project. For example, imagine a 6-month development project, with first code delivery to test in month 2.
- Month 2/3/4/5 may be bug-hunting.
  - Harsh tests in areas of highest risk
  - Exploratory scans for unanticipated areas of risk
- Month 6 may be helping the project manager determine whether the product is ready to ship.
  - Status and quality assessments. Less testing.
  - Tests include coverage-oriented surveys.

Make your mission explicit. Be wary of trying to achieve several missions at the same time.

Testing Strategy

Given a testing mission, how will you achieve it?

We define

- **Strategy** as the set of ideas that guide your test design
- **Logistics** as the set of ideas that guide your application of resources, and
- **Plan** as the combination of your strategy, your logistics and your project risk management.

See Bach's "Heuristic Test Planning: Context Model" http://www.satisfice.com/tools/satisfice-cm.pdf

Testing Strategy

Given a testing mission, how will you achieve it?

- The test strategy takes into account:
  - Your resources (time, money, tools, etc.)
  - Your staff's knowledge and skills
  - What is hard/easy/cheap (etc.) in your project environment
  - What risks apply to this project
- To choose the best combination of resources and techniques
  - that you can realistically bring to bear
  - to achieve the your mission
  - as well as you can under the circumstances

See Bach's "Heuristic Test Strategy Model", https://www.satisfice.com/download/heuristic-test-strategy-model

Strategy and Design

Think of the design task as applying the strategy to the choosing of specific test techniques and generating test ideas and supporting data, code or procedures:

- Who's going to run these tests? (What are their skills/ knowledge)?
- What kinds of potential problems are they looking for?
- How will they recognize suspicious behavior or "clear" failure? (Oracles?)
- What aspects of the software are they testing? (What are they ignoring?)
- How will they recognize that they have done enough of **this type** of testing?
- How are they going to test? (What are they actually going to do?)
- What tools will they use to create or run or assess these tests? (Do they have to create any of these tools?)
- What is their source of test data? (Why is this a good source? What makes these data suitable?)
- Will they create documentation or data archives to help organize their work or to guide the work of future testers?
- What are the outputs of these activities? (Reports? Logs? Archives? Code?)
- What aspects of the project context will make it hard to do this work

Two Examples of Test Techniques

Scenario Testing	Domain Testing
● Tests are complex stories that capture how the program will be used in real-life situations. ● These are combination tests, whose combinations are credible reflections of real use. ● These tests are highly credible (stakeholders will believe users will do these things) and so failures are likely to be fixed.	● For every variable or combination of variables, consider the set of possible values. ● Reduce the set by partitioning into subsets. Pick a few high-risk representatives (e.g. boundary values) of each subset. ● These tests are very powerful. They are more likely to trigger failures, but their reliance on extreme values makes some tests less credible.

Test Techniques (Bach)

A test technique is like a recipe. It tells you how it puts together some ingredients. Then you vary it to suit your needs.

A technique typically tells you how to do several (rarely all) of these:

- Analyze the situation
- Model the test space
- Select what to cover
- Determine test oracles
- Configure the test system
- Operate the test system
- Observe the test system
- Evaluate the test results

It takes several different recipes to create a complete meal.

Domain Testing

Illustrates the Components of the Recipe

Analyze the situation	We want to imagine the program as a collection of input and output variables. What are their possible values?
Model the test space	Follow a stratified sampling model, biased for higher probability of failure: For each variable, split possible values into groups that are treated equivalently. Consider valid & invalid values. Test at least one from each group, preferably the one most likely to show a failure. Next, test groups of a few variables together, applying a similar analysis.
Select what to cover	Which variables and combinations will we test?
Determine test oracles	Do we look only for input-rejection or output overflow?
Configure the test system	What equipment do we test on? Are we using tools to create tests? Execute them? Set everything up.
Operate the test system	Execute the tests (e.g. run the automation)
Observe the test system	Watch the execution of the tests. Is the system working correctly? (e.g. if human testers follow scripts, what actually happens while they test this way?)
Evaluate the test results	Did the program pass the tests?

Review

So far today:

- Testing
- Stakeholders
- Information objectives
- Mission
- Strategy
- Test techniques

Next:

- How is the testing effort organized?

A "Typical" Context

SoftCo creates tax preparation software.

- Sell 100,000 copies per year
- Two planned updates (incorporating changes in the tax code twice per year)
- Used by consumers and some paid preparers of tax returns
- Programming and testing are done within the company, at the same corporate headquarters.
- Test group (4 or more testers) reports to its own manager.

How will this project work?

This is like the projects we had in mind in *Testing Computer Software.*

A "Typical" Context: Typical Group

SoftCo's test group includes:

- Tester skilled with databases and calculations
- Bug-hunter
- Tool smith
- Tax lawyer (subject matter expert)
- Tester interested in network issues (including security & performance)
- Configuration tester
- Writer (writes test docs well)
- Test group manager

The details of this list are less important than the diversity of this group. Everyone is particularly good at something. Collectively, they will be expert at testing this class of application.

See Kaner (2000), "Recruiting Software Testers", https://kaner.com/pdfs/JobsRev6.pdf

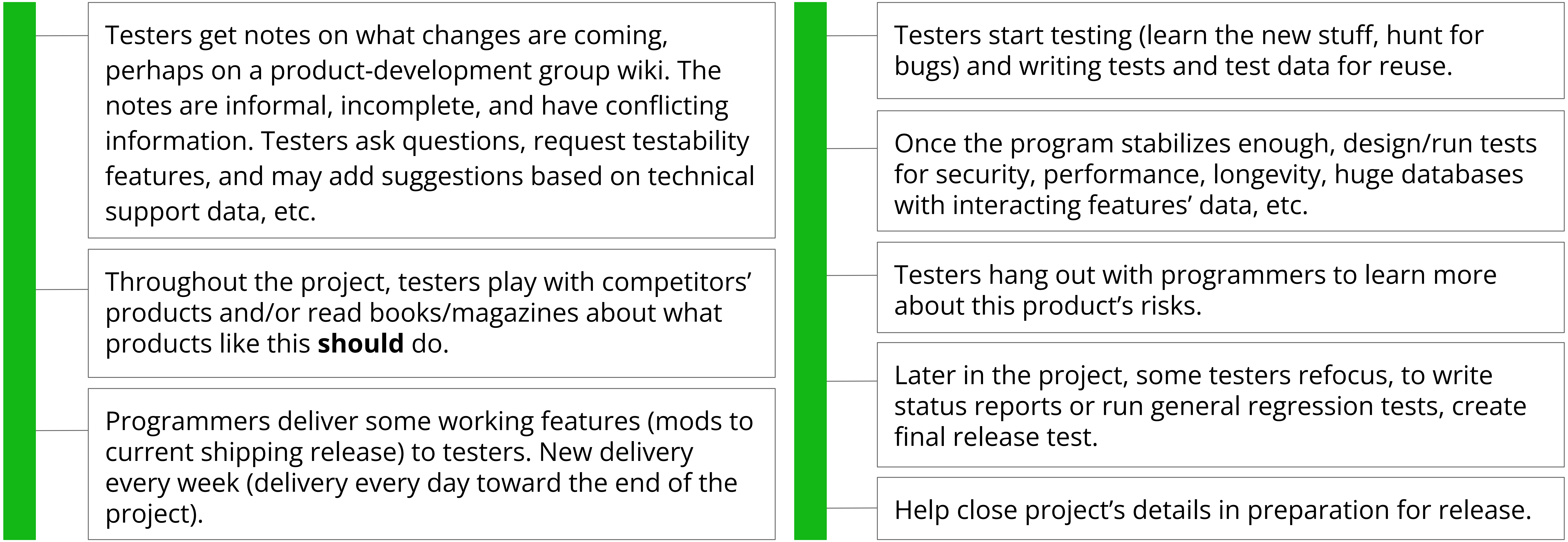
A “Typical” Context: Typical Tasks

- Research ways this product can fail or be unsatisfactory (essentially a requirements analysis from a tester’s point of view)
- Hunt bugs (exploratory risk-based testing)
- Analyze the specification and create tests that trace to spec items of interest
- Create sets of test data with well-understood attributes (to be used in several tests and archived)
- Create reusable tests (manual or automated)
- Create checklists for manual testing or to guide automation
- Research failures and write well-researched, persuasive bug reports

Most in-house test groups do most of these tasks.

A “Typical” Context: Tasks Over Time

Along with “testing”, these testers are involved in a diverse set of quality-related activities and release-support activities. These groups' scope varies over time and across test managers' and execs' attitudes.



Testers get notes on what changes are coming, perhaps on a product-development group wiki. The notes are informal, incomplete, and have conflicting information. Testers ask questions, request testability features, and may add suggestions based on technical support data, etc.

Throughout the project, testers play with competitors’ products and/or read books/magazines about what products like this **should** do.

Programmers deliver some working features (mods to current shipping release) to testers. New delivery every week (delivery every day toward the end of the project).

Testers start testing (learn the new stuff, hunt for bugs) and writing tests and test data for reuse.

Once the program stabilizes enough, design/run tests for security, performance, longevity, huge databases with interacting features’ data, etc.

Testers hang out with programmers to learn more about this product’s risks.

Later in the project, some testers refocus, to write status reports or run general regression tests, create final release test.

Help close project’s details in preparation for release.

A “Typical” Context: Less Common Tasks

- Write requirements
- Participate in inspections and walkthroughs
- Compile the software
- Conduct glass box tests
- Write installers
- Configure and maintain programming-related tools, such as the source control system
- Archive the software
- Investigate bugs, analyzing the source code to discover the underlying errors
- Evaluate the reliability of components that the company is thinking of using in its software
- Provide technical support
- Demonstrate the product at trade shows or internal company meetings

Few test groups provide all these services, but many in-house test groups provide several.
The more of these your staff provides, the more testers and the more skill-set diversity you need.
See Kaner, Hendrickson & Smith-Brock for discussion.

A “Typical” Context: Less Common Tasks

- Train new users (or tech support or training staff) in the use of the product
- Provide risk assessments
- Collect and report statistical data (software metrics) about the project
- Build and maintain internal test-related tools such as the bug tracking system
- Benchmark competing products
- Evaluate market significance of various hardware/software configurations (to inform their choices of configuration tests)
- Conduct usability tests
- Lead or audit efforts to comply with regulatory or industry standards (such as those published by SEI, ISO, IEEE, FDA, etc.)
- Provide project management services

These illustrate tradeoffs between “independence” and “collaboration.” Groups that see themselves as fundamentally independent provide a narrower range of services and have a narrower range of influence.

Missions (In-House)

- The typical missions that I’ve encountered when working with in-house test groups at mass-market software publishers have been much broader than bug-hunting.
- We would summarize some of the most common ones as follows (Note: a single testing project operates under one mission at a time):
  - Bug hunters
  - Quality advocacy
  - Development support
  - Release management
  - Support cost reduction

Many group's missions include their core goal for their own staff. For example, a group might see the services it provides as vehicles to support the education or career growth of its staff.

Change of Context: In-House IT?

- Several in-house IT organizations are reorganizing testing to try to get comparable benefits.
- I have no sense of industry statistics because the people who contact me have a serious problem and are willing to entertain my ideas on how to fix it.
- These managers complain about:
  - Ineffectiveness of their testers
  - Attention to process instead of quality
  - Lack of product/service knowledge
  - Lack of collaboration

An executive at a huge company explained to me why they were outsourcing testing. “We can’t get good testing from our own staff. If I have to get bad testing, I want it cheap.”

Change of Context: External Lab

People send their products for testing by an external lab for many reasons:

- The lab might offer specific skills that the original development company lacks.
- The customer (such as a government agency) might require a vendor to have the software tested by an independent lab because it doesn't trust the vendor.
- The company developing the software might perceive the outsourcer's services as cheaper.

Change of Context: External Lab

Difference	Consequences
They might be more skilled with some testing technology or at some specific testing tasks	● They might be more effective in some types of tests, e.g. producing better scripts (automated) faster for routine (e.g. domain) techniques
They don't understand our market (expectations, risks, needed benefits, competitors) or the idiosyncratic priorities of our key stakeholders.	● They probably won't be as good at benefit-driven testing (e.g. scenario) ● Their exploratory tests will be less well informed by knowledge of this class of product ● Usually they are focused on verification rather than validation ● Their design critiques in bug reports will be less credible and less welcome
They don't have collaborative opportunities with local developers and stakeholders	● They will need more supporting documentation ● They will generate more documentation and need it reviewed ● They will be unavailable or ineffective for collaborative bug fixing, release management, etc.

When I talk of "bug advocacy", friends whose experience is entirely "independent lab" think I am talking about pure "theory." Context constraints what tester roles are possible, and that shapes the possible missions.

Black Box Software Testing Foundations
Lecture 3
Oracles

Cem Kaner J.D., PH.D.

Professor Emeritus, Software Engineering, Florida Institute of Technology

Course Overview: Fundamental Topics

1. The Nature of Testing
   Overview and Basic Definitions

2. Why are we testing? What are we trying to learn? How should we organize our work to achieve this? ***Information objectives drive the testing mission and strategy***

3. How can we know whether a program has passed or failed a test? ←
   Oracles are heuristic

4. How can we determine how much testing has been done? What core knowledge about program internals do testers need to consider this question?
   Coverage is a multidimensional problem

5. Are we done yet?
   Complete testing is impossible

6. How much testing have we completed and how well have we done it?
   Measurement is important but hard

Today’s Readings

Required

- Michael Bolton (2005), “Testing without a map”, http://www.developsense.com/articles/2005-01-TestingWithoutAMap.pdf

Useful to skim

- Michael Kelly (2006), “Using Heuristic Test Oracles”, http://www.informit.com/articles/article.aspx?p=463947
- Billy V. Koen (1985), *Definition of the Engineering Method, American Society for Engineering Education (ASEE)*. (A later version that is more thorough but maybe less approachable is Discussion of the Method, Oxford University Press, 2003).
- Elaine Weyuker (1980), “On testing nontestable programs”, https://www.ics.uci.edu/~redmiles/ics221-FQ03/papers/Wey82.pdf
- Ru Cindrea (2020), “5 Concepts from BBST that Will Help You Write More Powerful Automated Tests”, https://bbst.courses/blog/bbst-concepts-that-will-help-you-create-powerful-automated-tests/

Once Upon A Time…

There used to be two common description of oracles:

1. An oracle is a mechanism for determining whether the program passed or failed a test.
2. An oracle is a reference program. If you give the same inputs to the software under test and the oracle, you can tell whether the software under test passed by comparing its results to the oracle's.

A Little More Terminology

SUT: Software (or system) under test. Similarly for the application under test (AUT) and the program under test (PUT).

Reference program: If we evaluate the behavior of the SUT by comparing it to another program’s behavior, the second program is the reference program or the **reference oracle.**

Comparator: the software or human that compares the behavior of the SUT to the oracle.

Oracle

Unfortunately, the ideas underlying the common oracle definitions are wrong.

Oracle

- “the **oracle assumption** [...] states that the tester is able to determine whether or not the output produced on the test data is correct. The mechanism which checks this correctness is known as an oracle. [...]

  Intuitively, it does not seem unreasonable to require that the tester be able to determine the correct answer in some ‘reasonable’ amount of time while expanding some ‘reasonable’ amount of effort. **Therefore, if either of the following two conditions occur, a program should be considered nontestable.**

  1) There does not exist an oracle.

  2) It is theoretically possible, but practically too difficult to determine the correct output.” (p. 1-2)
- **“many, if not most programs are by our definition nontestable.”** (p. 6)

See Weyuker, “On testing nontestable programs”, 1980 https://www.ics.uci.edu/~redmiles/ics221-FQ03/papers/Wey82.pdf

The Need for Judgement

When we describe **testing** as a

- **process of comparing empirical results to expected results**

we must consider that even the basic process of **comparison**

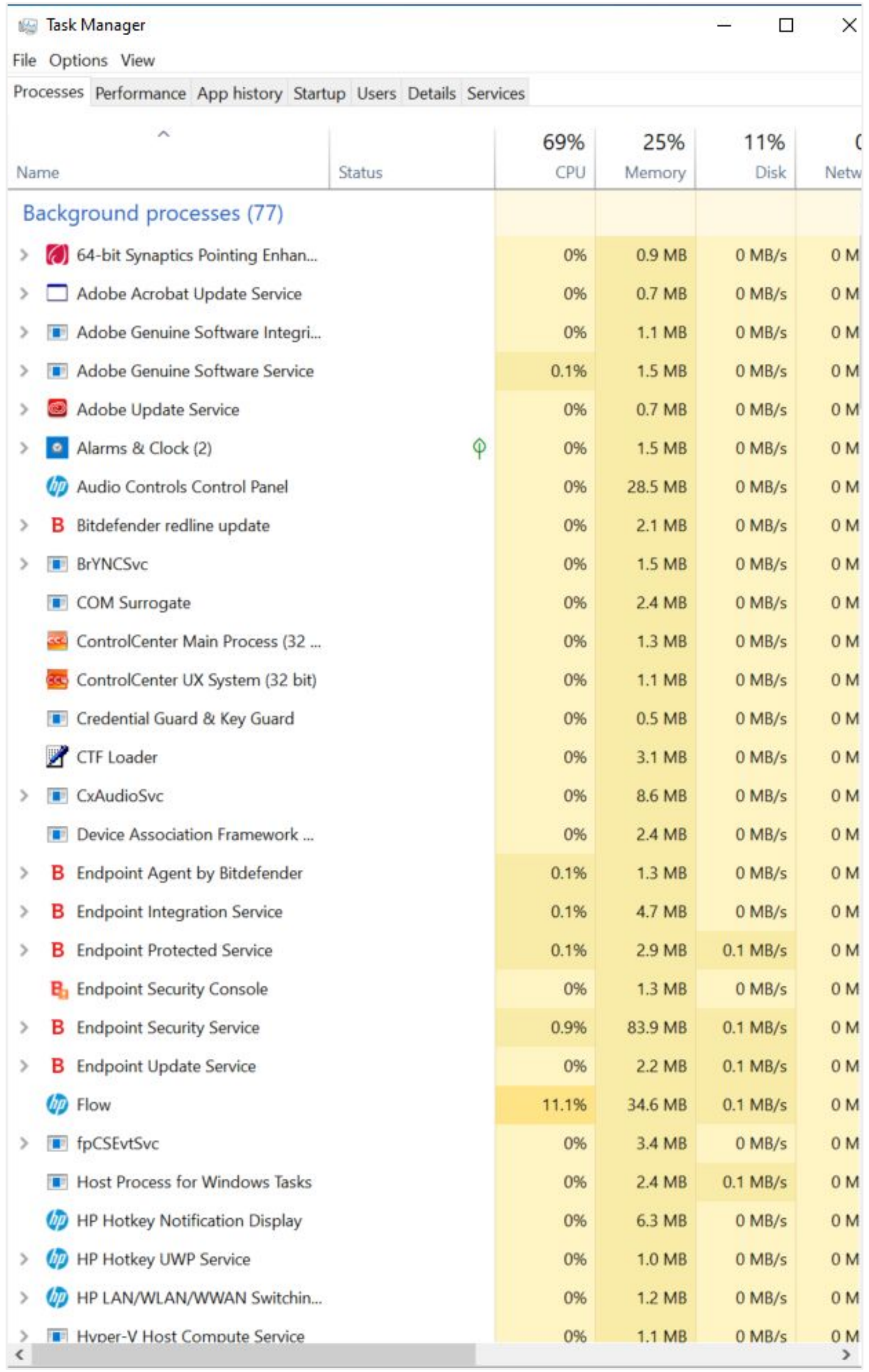
- **requires human judgment, based on an understanding of the problem domain.**

Can You Specify Your **Test Configuration?**

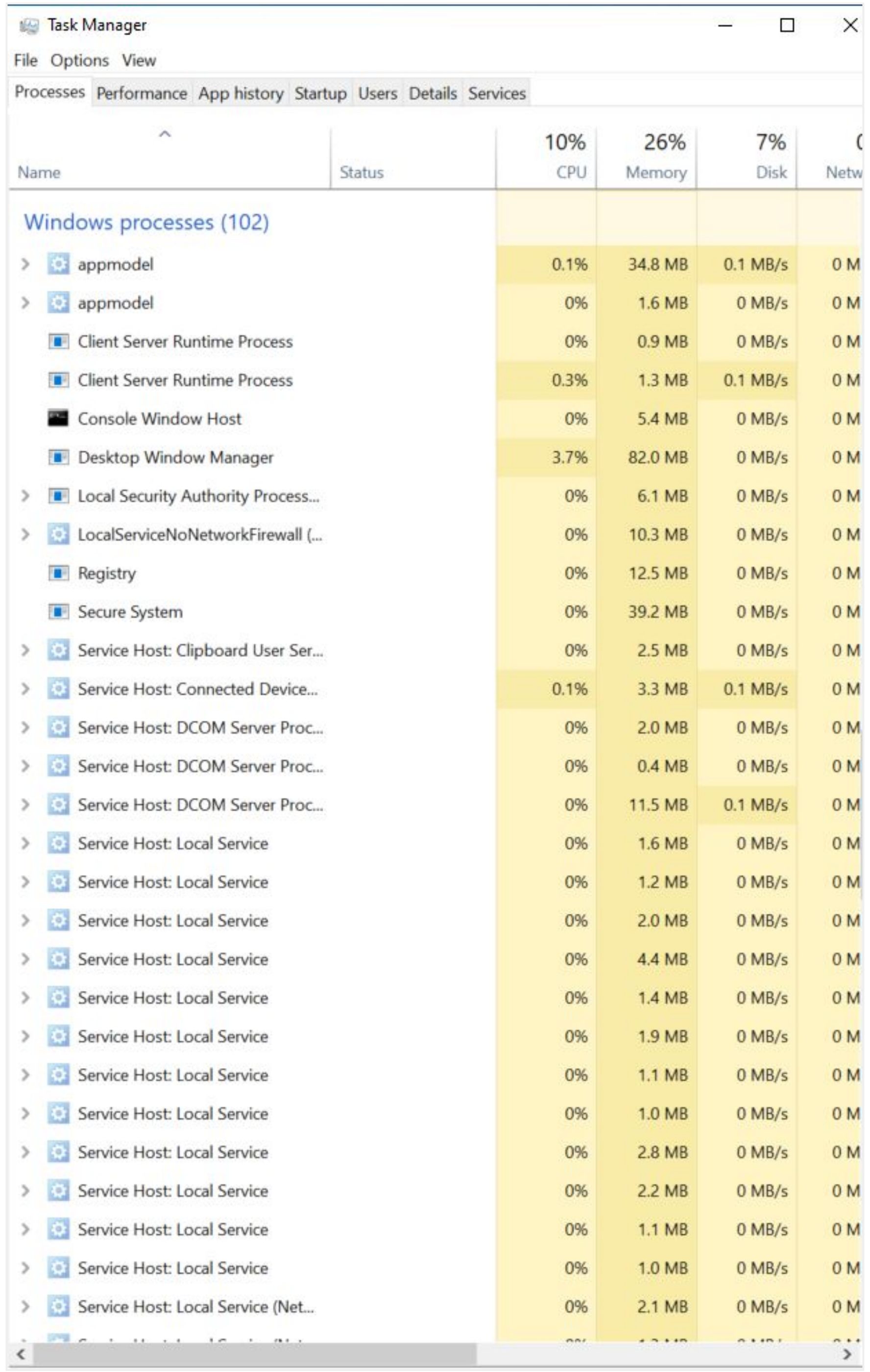
Comparison to a reference function is fallible. We only control some inputs and observe some results (outputs).

For example, do you know whether test & reference systems are equivalently configured?

- Does your test documentation specify ALL the processes running on your computer?
- Does it specify what version of each one?
- **Do you even know how to tell:**
  - What version of each of these you are running?
  - When you (or your system) last updated each one?
  - Whether there is a later update?

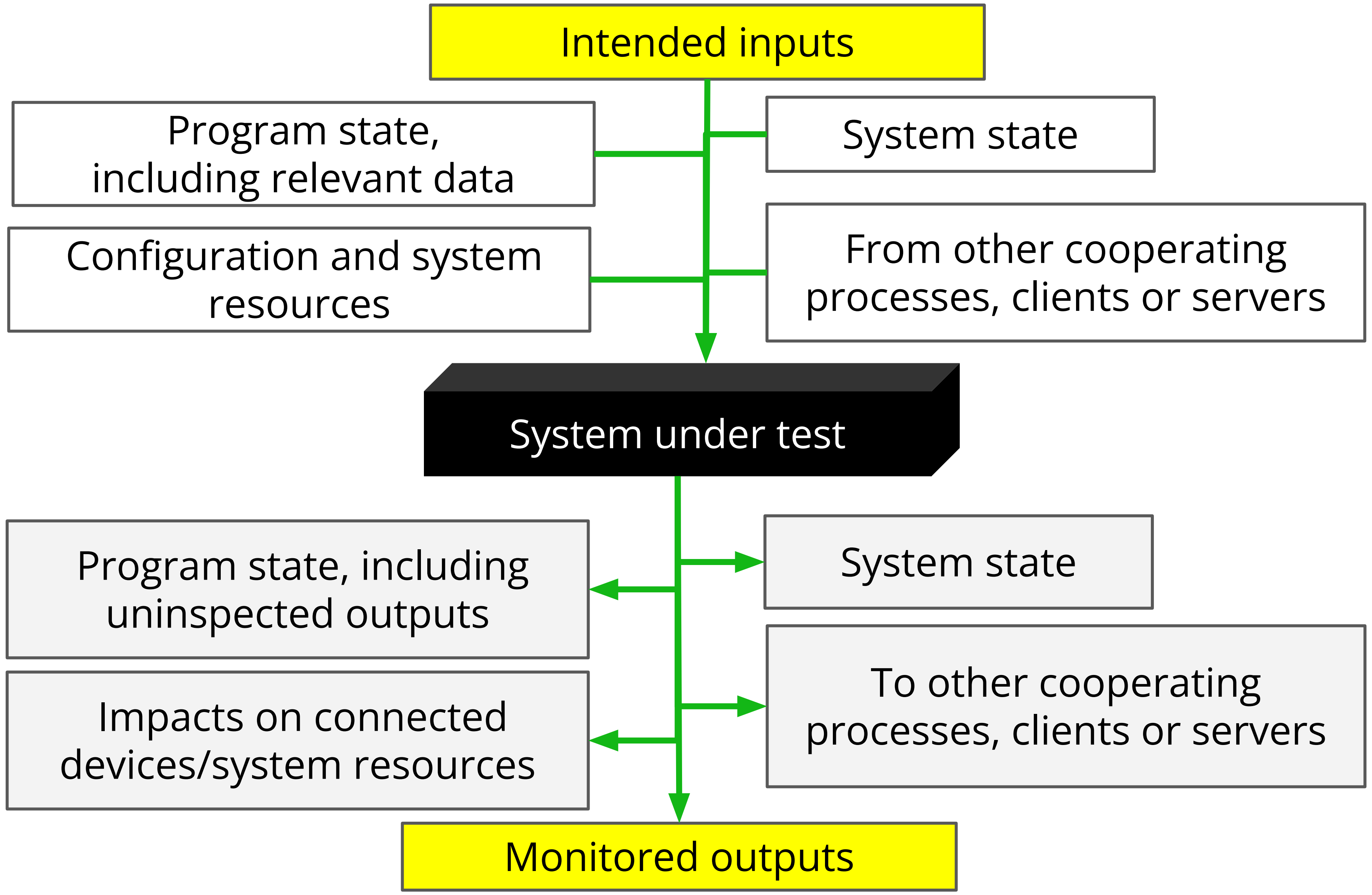

Task Manager
File Options View
Processes Performance App history Startup Users Details Services

Name	Status	69% CPU	25% Memory	11% Disk	(Netw
Background processes (77)					
64-bit Synaptics Pointing Enhan...		0%	0.9 MB	0 MB/s	0 M
Adobe Acrobat Update Service		0%	0.7 MB	0 MB/s	0 M
Adobe Genuine Software Integri...		0%	1.1 MB	0 MB/s	0 M
Adobe Genuine Software Service		0.1%	1.5 MB	0 MB/s	0 M
Adobe Update Service		0%	0.7 MB	0 MB/s	0 M
Alarms & Clock (2)		0%	1.5 MB	0 MB/s	0 M
Audio Controls Control Panel		0%	28.5 MB	0 MB/s	0 M
Bitdefender redline update		0%	2.1 MB	0 MB/s	0 M
BrYNCSvc		0%	1.5 MB	0 MB/s	0 M
COM Surrogate		0%	2.4 MB	0 MB/s	0 M
ControlCenter Main Process (32 ...		0%	1.3 MB	0 MB/s	0 M
ControlCenter UX System (32 bit)		0%	1.1 MB	0 MB/s	0 M
Credential Guard & Key Guard		0%	0.5 MB	0 MB/s	0 M
CTF Loader		0%	3.1 MB	0 MB/s	0 M
CxAudioSvc		0%	8.6 MB	0 MB/s	0 M
Device Association Framework ...		0%	2.4 MB	0 MB/s	0 M
Endpoint Agent by Bitdefender		0.1%	1.3 MB	0 MB/s	0 M
Endpoint Integration Service		0.1%	4.7 MB	0 MB/s	0 M
Endpoint Protected Service		0.1%	2.9 MB	0.1 MB/s	0 M
Endpoint Security Console		0%	1.3 MB	0 MB/s	0 M
Endpoint Security Service		0.9%	83.9 MB	0.1 MB/s	0 M
Endpoint Update Service		0%	2.2 MB	0.1 MB/s	0 M
Flow		11.1%	34.6 MB	0.1 MB/s	0 M
fpCSEvtSvc		0%	3.4 MB	0 MB/s	0 M
Host Process for Windows Tasks		0%	2.4 MB	0.1 MB/s	0 M
HP Hotkey Notification Display		0%	6.3 MB	0 MB/s	0 M
HP Hotkey UWP Service		0%	1.0 MB	0 MB/s	0 M
HP LAN/WLAN/WWAN Switchin...		0%	1.2 MB	0 MB/s	0 M
Hyper-V Host Compute Service		0%	1.1 MB	0 MB/s	0 M


Task Manager
File Options View
Processes Performance App history Startup Users Details Services

Name	Status	10% CPU	26% Memory	7% Disk	(Netw
Windows processes (102)					
appmodel		0.1%	34.8 MB	0.1 MB/s	0 M
appmodel		0%	1.6 MB	0 MB/s	0 M
Client Server Runtime Process		0%	0.9 MB	0 MB/s	0 M
Client Server Runtime Process		0.3%	1.3 MB	0.1 MB/s	0 M
Console Window Host		0%	5.4 MB	0 MB/s	0 M
Desktop Window Manager		3.7%	82.0 MB	0 MB/s	0 M
Local Security Authority Process...		0%	6.1 MB	0 MB/s	0 M
LocalServiceNoNetworkFirewall (...		0%	10.3 MB	0 MB/s	0 M
Registry		0%	12.5 MB	0 MB/s	0 M
Secure System		0%	39.2 MB	0 MB/s	0 M
Service Host: Clipboard User Ser...		0%	2.5 MB	0 MB/s	0 M
Service Host: Connected Device...		0.1%	3.3 MB	0.1 MB/s	0 M
Service Host: DCOM Server Proc...		0%	2.0 MB	0 MB/s	0 M
Service Host: DCOM Server Proc...		0%	0.4 MB	0 MB/s	0 M
Service Host: DCOM Server Proc...		0%	11.5 MB	0.1 MB/s	0 M
Service Host: Local Service		0%	1.6 MB	0 MB/s	0 M
Service Host: Local Service		0%	1.2 MB	0 MB/s	0 M
Service Host: Local Service		0%	2.0 MB	0 MB/s	0 M
Service Host: Local Service		0%	4.4 MB	0 MB/s	0 M
Service Host: Local Service		0%	1.4 MB	0 MB/s	0 M
Service Host: Local Service		0%	1.9 MB	0 MB/s	0 M
Service Host: Local Service		0%	1.1 MB	0 MB/s	0 M
Service Host: Local Service		0%	1.0 MB	0 MB/s	0 M
Service Host: Local Service		0%	2.8 MB	0 MB/s	0 M
Service Host: Local Service		0%	2.2 MB	0 MB/s	0 M
Service Host: Local Service		0%	1.1 MB	0 MB/s	0 M
Service Host: Local Service		0%	1.0 MB	0 MB/s	0 M
Service Host: Local Service (Net...		0%	2.1 MB	0 MB/s	0 M

A Model of a System Under Test



Intended inputs

Program state, including relevant data

Configuration and system resources

System state

From other cooperating processes, clients or servers

System under test

Program state, including uninspected outputs

Impacts on connected devices/system resources

System state

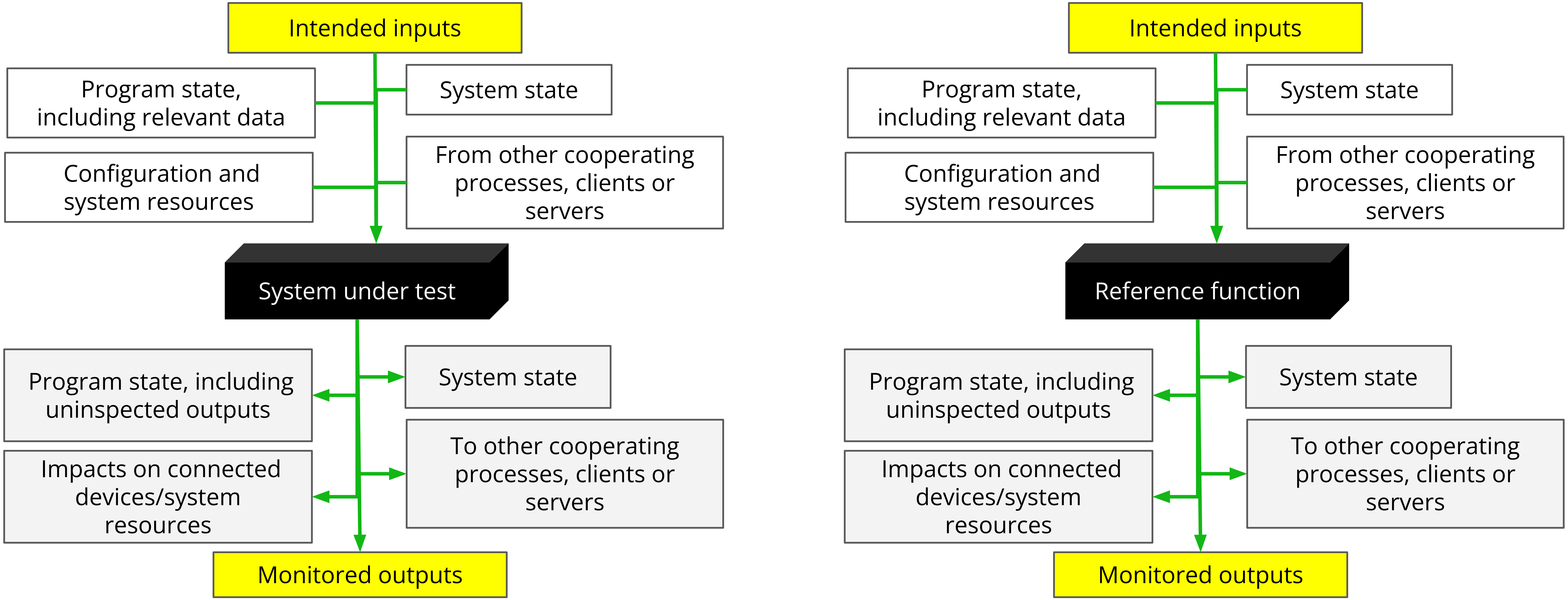
To other cooperating processes, clients or servers

Monitored outputs

See Doug Hoffman https://pdfs.semanticscholar.org/4939/9c41364c832dadf491c11376bd7ca38e4a8b.pdf

Reference Programs Have Limited Values

Based on Notes From Doug Hoffman



Intended inputs

Program state, including relevant data

System state

Configuration and system resources

From other cooperating processes, clients or servers

System under test

Program state, including uninspected outputs

System state

Impacts on connected devices/system resources

To other cooperating processes, clients or servers

Monitored outputs

Intended inputs

Program state, including relevant data

System state

Configuration and system resources

From other cooperating processes, clients or servers

Reference function

Program state, including uninspected outputs

System state

Impacts on connected devices/system resources

To other cooperating processes, clients or servers

Monitored outputs

Our Observations Can Fail in Many Ways

Selective attention and inattentional blindness

- Humans (often) don't see what they don't pay attention to
  https://www.youtube.com/watch?v=Ahg6qcgoay4
- Programs don't see what they haven't been told to pay attention to.

This is often the cause of irreproducible failures. We pay attention to the wrong conditions.

- But we can't attend to all the conditions

A Program Can Fail in Many Ways

1100 embedded diagnostics

- Even if we coded checks for each of these diagnostics
    - the side effects (data, resources, and timing)
    - would provide us a new context for Heisenberg uncertainty.

“It works” really means “It appears to meet some requirement to some degree.”

Our tests cannot practically address all of the possibilities

An Oracle Is a Heuristic

An oracle is a heuristic principle or mechanism by which you recognize a *potential* problem.

Mainstream Engineering Relies Fundamentally on Heuristics

- "A heuristic is anything that provides a plausible aid or direction in the solution of a problem but is in the final analysis unjustified, incapable of justification, and fallible. It is used to guide, to discover, and to reveal." (p. 21)
- Heuristics do not guarantee a solution. (p. 22)
- "two heuristics may contradict or give different answers to the same question and still be useful." (p. 24)
- Heuristics permit the solving of unsolvable problems or reduce the search time to a satisfactory solution. (p. 26)
- The heuristic depends on the immediate context instead of absolute truth as a standard of validity. (p. 22)

"The engineering method is the use of heuristics to cause the best change in a poorly understood situation within the available resources" (p. 70)

See Billy V. Koen, *Definition of the Engineering Method, American Society for Engineering Education,* 1985
https://files.eric.ed.gov/fulltext/ED276572.pdf

Testing Is About Ideas.
Heuristics Give You Ideas.

- A heuristic is a fallible idea or method that may you help simplify and solve a problem.
- Heuristics can hurt you when used as if they were authoritative rules.
- Heuristics may suggest wise behavior, but only in context. They do not contain wisdom.
- Your relationship to a heuristic is the key to applying it wisely

> **“Heuristic reasoning is not regarded as final and strict but as provisional and plausible only, whose purpose is to discover the solution to the present problem.”**
> **George Polya,**
> ***How to Solve It***

Fallible Decision Rules

		How the tester interprets the test	
The actual state of the program		Bug	Feature
	Bug	Hit	Miss
	Feature	False alarm	Correct acceptance

Decisions based on oracles can be erroneous in two ways:

- **Miss:** We incorrectly conclude that the program passes because we miss the incorrect behavior (or the software and the oracle are both wrong).
- **False Alarm:** We incorrectly conclude that the program failed because we interpret correct behavior as incorrect.

A fallible decision rule can be subject to either type of error (or to both).

Oracles & Test Automation

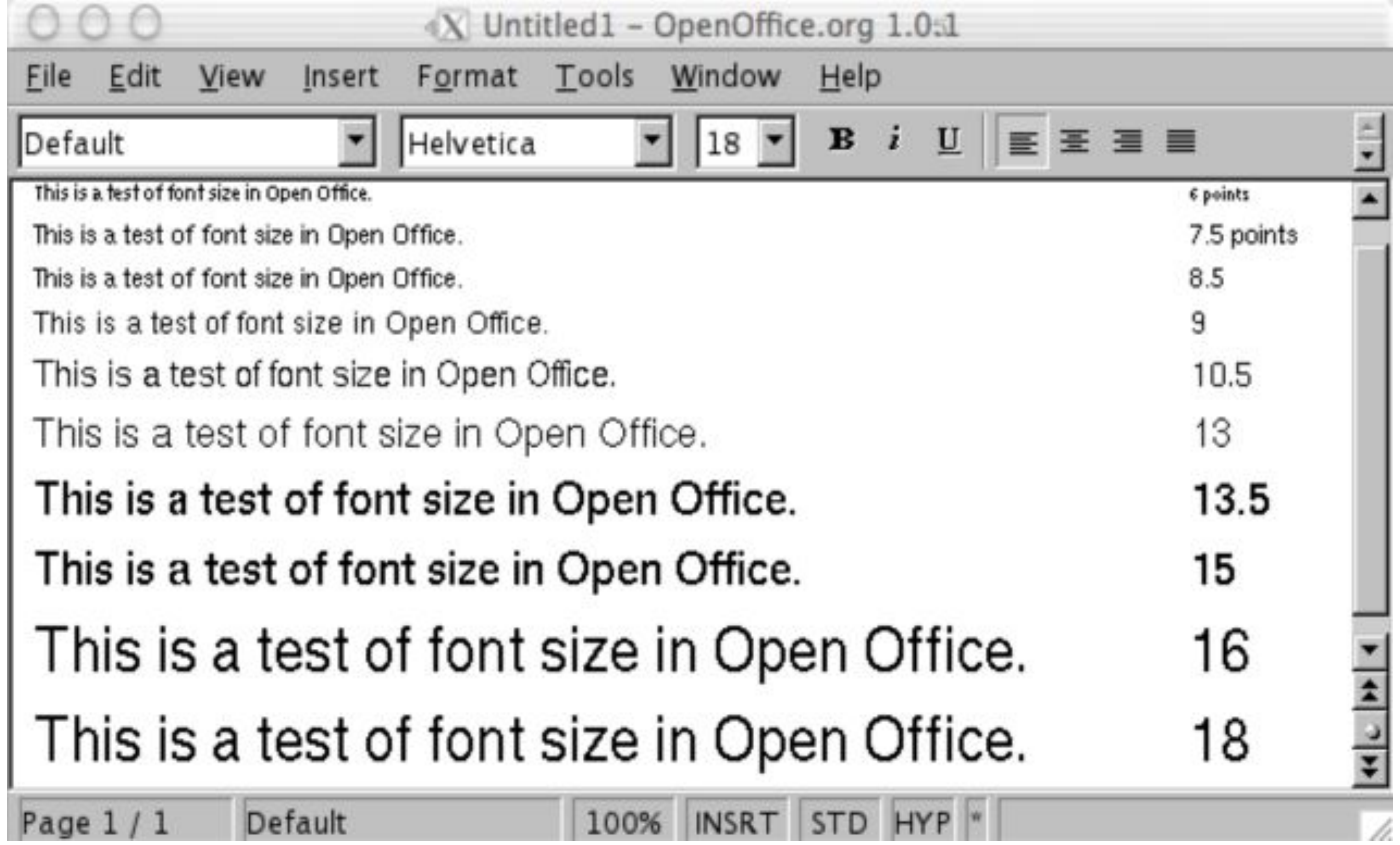
We often hear that most (or all) testing should be automated.

- Automated testing depends on our ability to programmatically detect when the software under test fails a test.
- Automate or not, you must still exercise judgment in picking risks to test against and interpreting the results.
- Automated comparison-based testing is subject to false alarms and misses.

Our ability to automate testing is fundamentally constrained by our ability to create and use oracles.

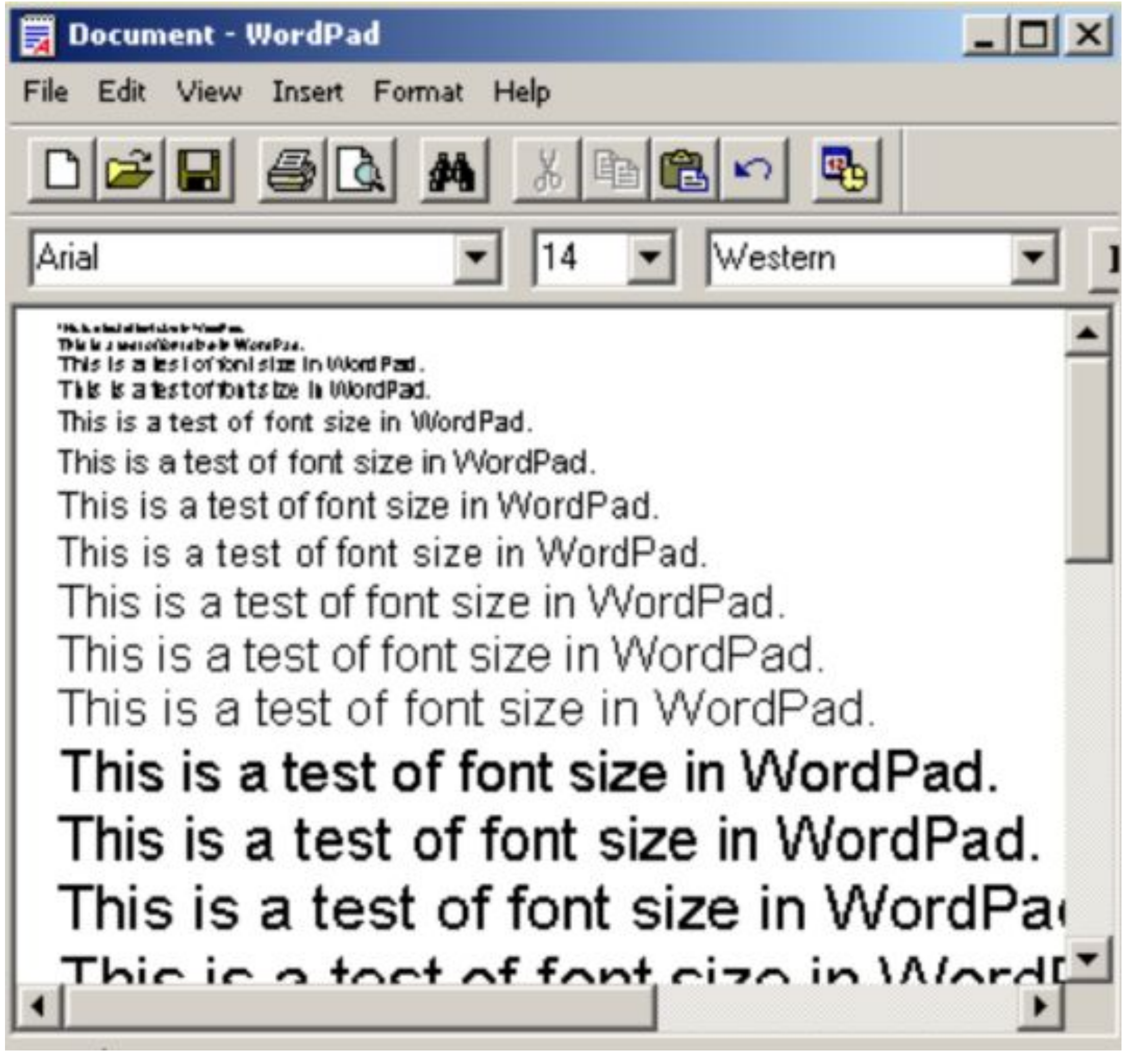
Does Font Size Work in Open Office?

What’s your oracle?



This is a test of font size in Open Office.	6 points
This is a test of font size in Open Office.	7.5 points
This is a test of font size in Open Office.	8.5
This is a test of font size in Open Office.	9
This is a test of font size in Open Office.	10.5
This is a test of font size in Open Office.	13
This is a test of font size in Open Office.	**13.5**
This is a test of font size in Open Office.	**15**
This is a test of font size in Open Office.	16
This is a test of font size in Open Office.	18

OK, So What About WordPad?



Arial

14

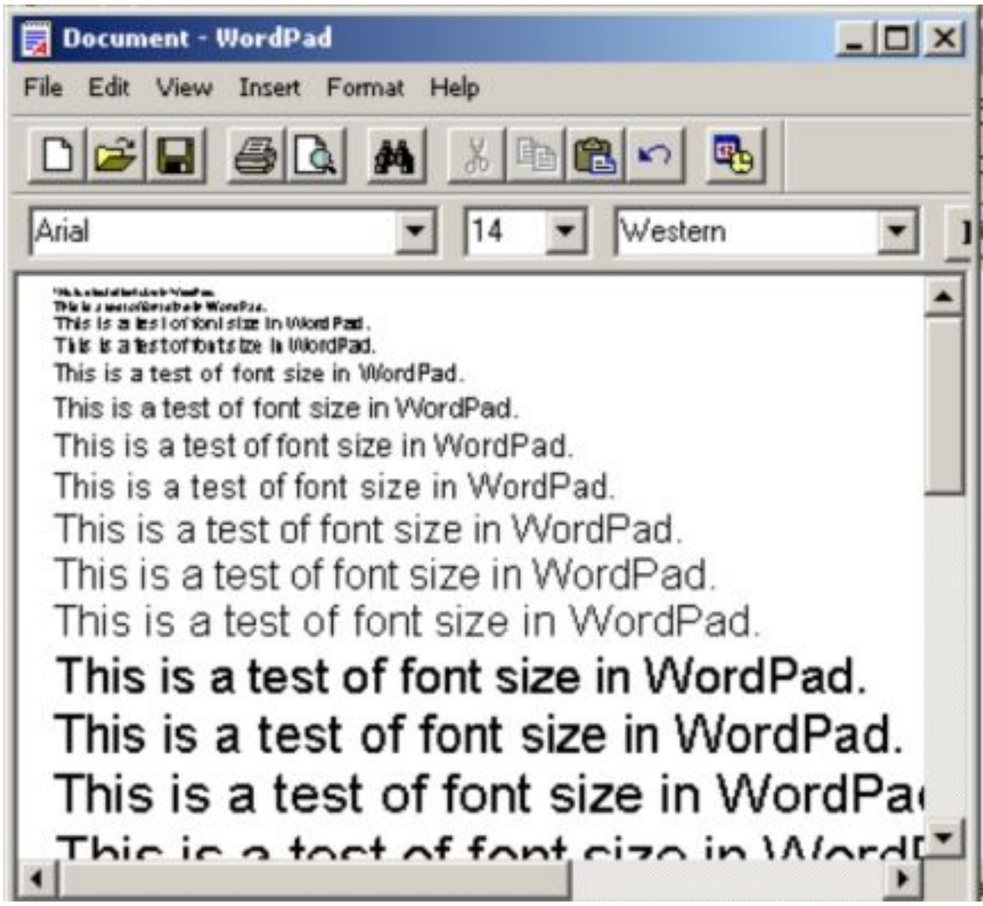
Western

[illegible]
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPa
This is a test of font size in WordP

Compare WordPad to Word

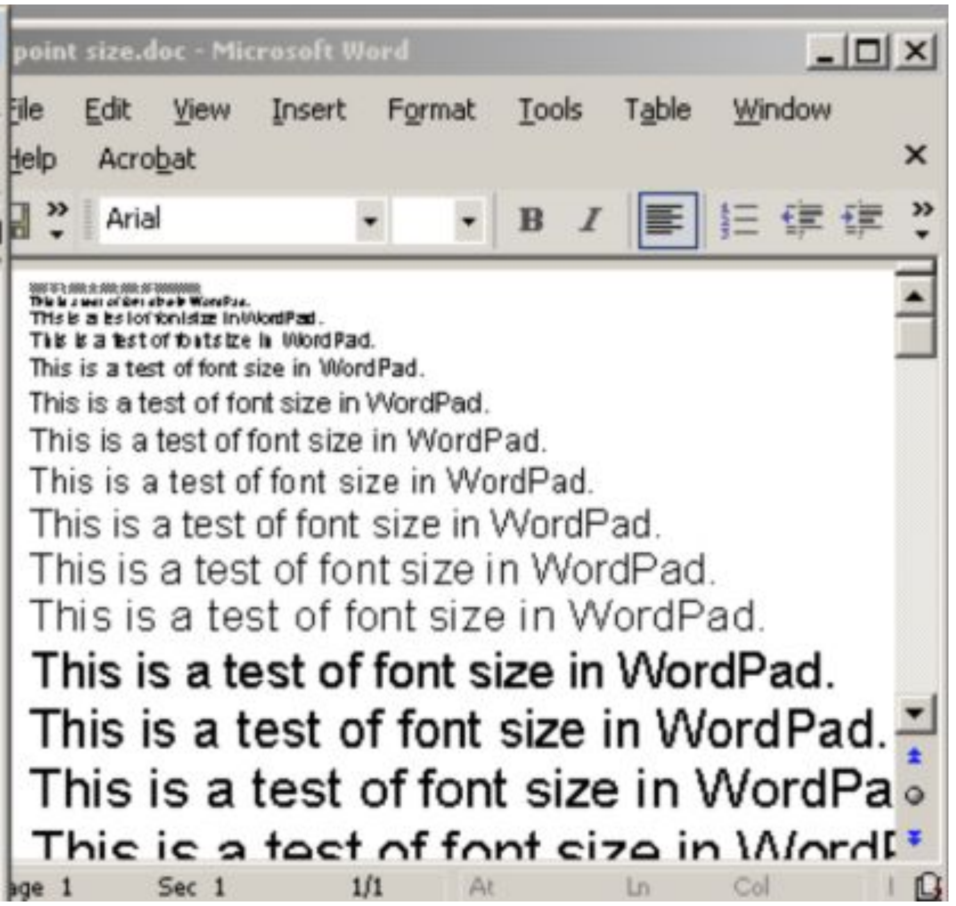
Are they as similar as they look?

WordPad



This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPa

Word

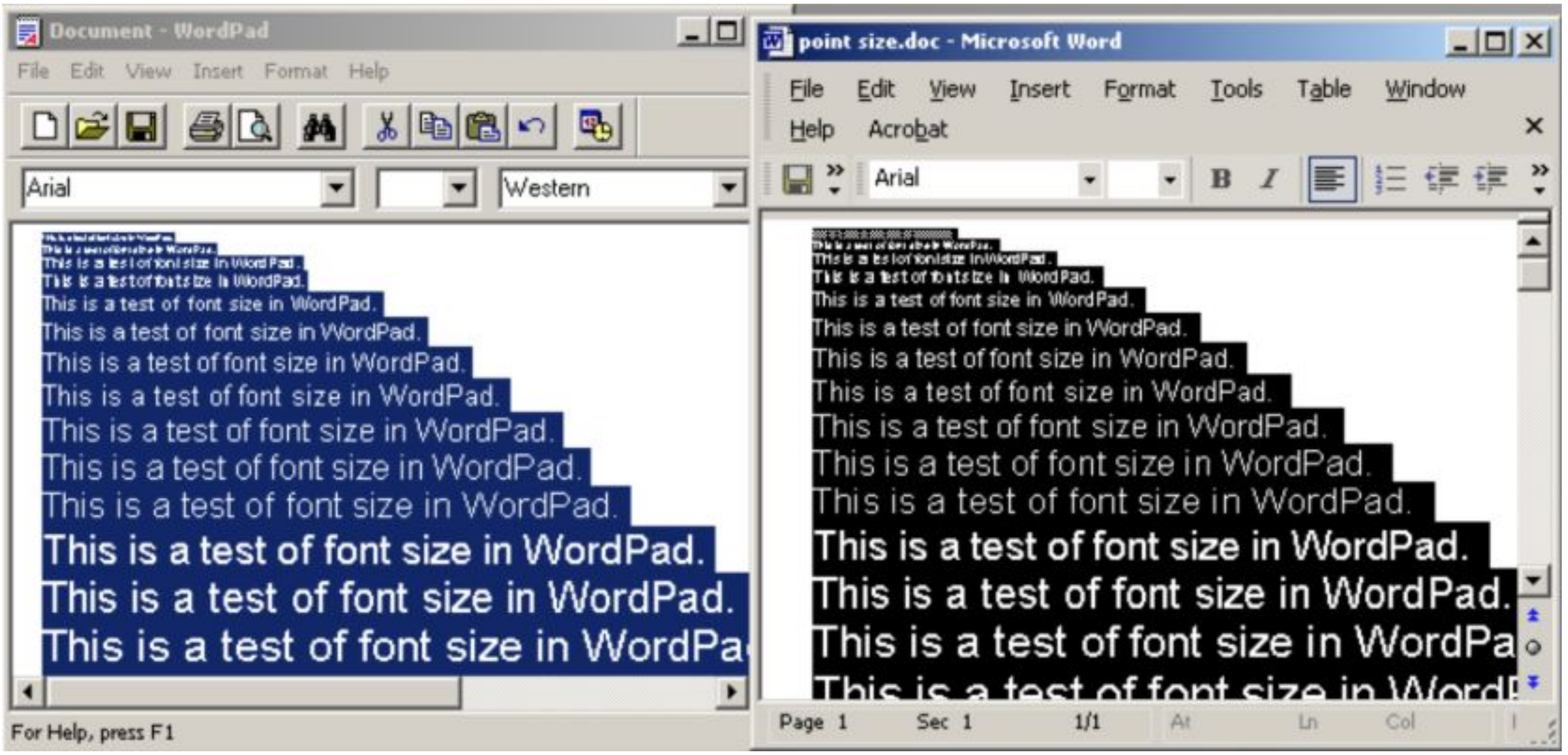


This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPa

Compare WordPad to Word

Highlighting makes relative sizes more obvious.

WordPad **Word**


Document - WordPad
File Edit View Insert Format Help
Arial
Western
For Help, press F1
point size.doc - Microsoft Word
File Edit View Insert Format Tools Table Window
Help Acrobat
Arial
B I
Page 1 Sec 1 1/1 At Ln Col

This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPa

This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPad.
This is a test of font size in WordPa
This is a test of font size in Word

Now That We See a Difference ...

... Is it a difference that makes a difference?

The oracle highlights
the fundamental role
of judgment in testing.

Testing is a cognitive activity not a mechanical activity.

Risk As a Simplifying Factor

- For Wordpad, we don’t care if font size meets precise standards of typography!
- In general it can vastly simplify testing if we focus on whether the product has a problem that matters, rather than whether the product merely satisfies all relevant standards.
- Effective testing requires that we understand standards as they relate to how our clients value the product.

Instead of thinking about pass vs. fail, consider thinking problem vs. no problem. Michael Kelly (2006), “Using Heuristic Test Oracles”

http://www.informit.com/articles/article.aspx?p=463947

Risk As a Simplifying Factor

What if we applied the same evaluation approach

- that we applied to WordPad
- to Open Office or MS Word or Adobe InDesign?

In risk-based testing,

- we choose the tests that we think are
  - the most likely to expose a serious problem,
- and skip the tests that we think are
  - unlikely to expose a problem, or
  - likely to expose problems that no one would care about.

The same evaluation criteria lead to different conclusions in different contexts

Risk As a Simplifying Factor

What would we examine if we were comparing font handling in professional word processors?

Some examples:

- Every font size (to a tenth of a point)
- Every character in every font
- Every method of changing font size
- Every user interface element associated with font size
- Interactions between font size and other contents of a document
- Interactions between font size and every other feature
- Interactions between font sizes and graphics cards & modes
- Print vs. screen display

What Evaluation Criterion?

- What do you know about typography?
  - Definition of “point” varies. There are at least six different definitions
    http://www.oberonplace.com/dtp/fonts/point.htm
  - Absolute size of characters can be measured, but not easily
    http://www.oberonplace.com/dtp/fonts/fontsize.htm
- How closely must size match to the chosen standard?
- **Heuristic approaches**, such as:
  - relative size of characters
  - comparison to MS Word
  - expectations of different kinds of users for different uses.

Testing is about ideas.

Heuristics give you ideas.

Review

We started with the traditional views:

- Testing is a process of comparing empirical results to expected results.
- An oracle is a mechanism for determining whether the program passed or failed a test.

Four problems:

- Our expectations are not necessarily correct.
- Our expectations are not complete.
- A mismatch between result and expectation might not be serious enough to report.
- Our expectations are not necessarily credible.

The traditional perspective doesn't work, but we still need, have, and use, test oracles.

Consistency Oracles

Consistent within product	Function behavior consistent with behavior of comparable functions or functional patterns within the product
Consistent with comparable products	Function behavior consistent with that of similar functions in comparable products
Consistent with history	Present behavior consistent with past behavior
Consistent with our image	Behavior consistent with an image the organization wants to project
Consistent with claims	Behavior consistent with documentation, specifications, or ads
Consistent with standards or regulations	Behavior consistent with externally-imposed requirements
Consistent with user's expectations	Behavior consistent with what we think users want
Consistent with purpose	Behavior consistent with product or function's apparent purpose

All of these are heuristics. They are useful, but they are not always correct and they are not always consistent with each other.

Use Consistency Oracles for Test Reporting

Something seems inappropriate.
How can you explain to the programmers (or other stakeholders) that this is bad?

Consider: **Consistency with purpose**

- What's the point of this product? Why do we think people should use it? What should they do with it?
- Does this error make it harder for them to achieve the benefits that they use this product to achieve?
- Research the product’s benefits (books, interview experts, course examples, specifications, marketing materials, etc.).
- Use these materials to decide what people want to gain from this product.
- Test to see if users can achieve these benefits. If not, write bug reports. Explain what benefit you expect, why (cite the reference) you expect this, and then show the test that makes it unachievable or difficult.

Tie the reports to the facts and data you collected in your research

Consistency Oracles Often Require Research

Example:

Consistency with purpose

- How do you know what the purpose of the product is?
- Even if you know (or think you know) the purpose, is your knowledge credible? Will other people agree that your perception of the purpose is correct?

Junior testers are like children, waiting for this type of information to be handed to them, and whining if they don't get it.

Competent testers ask for the information, but if they don't get it, they do their own research.

Consistency Oracles Often Require Research

Consistency with purpose

What questions should you ask in order to guide your research?

Here's an example of at least one aspect of the purpose of the product:

- **What should people want to achieve from this type of product?**
    - What is the nature of this task?
    - How do people do it in the world?
    - What do people consider “success” or “completion” in this type of task?

Consistency Oracles Often Require Research

Consistency with purpose

What sources can you consult to answer these questions?

Here are a few examples...

- **Internal documents:** such as specifications, marketing documents
- **Competing products:** what they do and how they work (work with them, read their docs and marketing statements) and published reviews of them
- **Training materials, books, courses:** for example, if you're testing a spreadsheet, where do people learn how to use them? Where do people learn about the things (e.g. balance sheets) that spreadsheets to help us create?
- **Users:** Read your company's technical support (help desk) records. Or talk with real people who have been using your product (or comparable ones) to do real tasks

Credibility doesn't come automatically to you as a tester. You have to earn it by getting to know what you are talking about.

Use Consistency Oracles

- To guide bug evaluation
  - Why do I think something is wrong with this behavior?
  - Is this a bug or not?
- To guide reporting
  - How can I credibly argue that this is a problem?
  - How can I explain why I think this is serious?
- To guide test design
  - If I know something the product *should be* consistent with, I can predict things the product should or should not do and I can design tests to check those predictions.

Another Look at Oracles

Based on Notes from Doug Hoffman

Oracles	Description	Advantages	Disadvantages
No Oracle (automated test or incompetent human)	● Doesn't explicitly check results for correctness ("Run till crash")	● Can run any amount of data (limited by the time the SUT takes) ● Useful early in testing. We generate tests randomly or from a model and see what happens	● Notices only spectacular failures ● Replication of sequence leading to failure may be difficult
No oracle (competent human testing)	● Humans often come to programs without knowing what to expect from a particular test. They figure out how to evaluate the test while they run the test.	● See Bolton (2010), "Inputs and expected results" www.developsense.com/blog/2010/05/a-transpection-session-inputs-and-expected-results ● People don't test with "no oracles." They use general expectations and product-specific information that they gather while testing.	● Testers who are too inexperienced, too insecure, or too dogmatic to rely on their wits need more structure.
Complete Oracle	● Authoritative mechanism for determining whether the program passed or failed	● Detects all types of errors ● If we have a complete oracle, we can run automated tests and check the results against it	● This is a mythological creature: software equivalent of a unicorn

More Types of Oracles

Based on Notes from Doug Hoffman

Oracles	Description	Advantages	Disadvantages
Heuristic Consistency Oracles	- Consistent with ○ within product ○ comparable products ○ history ○ our image ○ claims ○ specifications or regulations ○ user expectations ○ purpose	- We can probably force-fit most or all other types of oracles into this structure (classification system for oracles) - The structure illustrates ideas for test design and persuasive test result reporting	- The structure seems too general for some students (including some experienced practitioners)
Partial	- Verifies only some aspects of the test output - All oracles are partial oracles	- More likely to exist than a Complete Oracle - Much less expensive to create and use	- Can miss systematic errors - Can miss obvious errors

More Types of Oracles

Based on Notes from Doug Hoffman And Michael Bolton

Oracles	Description	Advantages	Disadvantages
Constraints	- Checks for ○ impossible values or ○ Impossible relationships - Examples: ○ ZIP codes must be 5 or 9 digits ○ Page size (output format) must not exceed physical page size (printer) ○ Event 1 must happen before Event 2 ○ In an order entry system, date/time correlates with order number	- The errors exposed are probably straightforward coding errors that must be fixed - This is useful even though it is insufficient	- Catches some obvious errors but if a value (or relationship between two variables' values) is incorrect but doesn't obviously conflict with the constraint, the error is not detected
Familiar failure patterns	- The application behaves in a way that reminds us of failures in other programs - This is probably not sufficient in itself to warrant a bug report, but it is enough to motivate further research	- Normally we think of oracles describing how the program should behave. (It should be consistent with X.) This works from a different mindset ("this looks like a problem," instead of "this looks like a match.")	- False analogies can be distracting or embarrassing if the tester files a report without adequate troubleshooting

More Types of Oracles

Based on Notes from Doug Hoffman

Oracles	Description	Advantages	Disadvantages
Regression Test Oracle	• Compare results of tests of this build with results from a previous build. The prior results are the oracle	• Verification is often a straightforward comparison • Can generate and verify large amounts of data • Excellent selection of tools to support this approach to testing	• Verification fails if the program's design changes (many false alarms, but some tools reduce false alarms) • Misses bugs that were in previous build or are not exposed by the comparison
Self-Verifying Data	• Embeds correct answer in the test data (such as embedding the correct response in a message comment field or the correct result of a calculation or sort in a database record) • CRC, checksum or digital signature	• Allows extensive post-test analysis • Does not require external oracles • Verification is based on contents of the message or record, not on user interface • Answers are often derived logically and vary little with changes to the user interface • Can generate and verify large amounts of complex data	• Must define answers and generate messages or records to contain them • In protocol testing (testing the creation and sending of messages and how the recipient responds), if the protocol changes we might have to change all the tests • Misses bugs that don't cause mismatching result fields

Models

The next several oracles are based on models.

- A model is a simplified, formal representation of a relationship, process or system. The simplification makes some aspects of the thing modeled clearer, more visible, and easier to work with.
- All tests are based on models, but many of those models are implicit. When the behavior of the program “feels wrong” it is clashing with your internal model of the program and how it should behave.

What Might We Model in an Oracle?

- The physical process being emulated, controlled or analyzed by the software under test
- The business process being emulated, controlled or analyzed by the software under test
- The software being emulated, controlled, communicated with or analyzed by the software under test
- The device(s) this program will interact with
- The reactions or expectations of the stakeholder community
- The uses/usage patterns of the product
- The transactions that this product participates in
- The user interface of the product
- The objects created by this product

Guides in Creating a Model

What aspects of the things we model might guide our creation of a model?

- Capabilities
- Preferences
  - Competitive analysis
  - Support records
- Focused chronology
  - Achievement of a task or life history of an object or action
- Sequences of actions
  - Such as state diagrams or other sequence diagrams
  - Flow of control
- Flow of information
  - Such as data flow diagrams or protocol diagrams or maps
- Interactions/dependencies
  - Such as combination charts or decision trees
  - Charts of data dependencies
  - Charts of connections of parts of a system
- Collections
  - Such as taxonomies or parallel lists
- Motives
  - Interest analysis: Who is affected, how, by what?

What Makes These Models, *Models*?

- The representation (the model) is simpler than what is modeled: It emphasizes some aspects of what is modeled while hiding other aspects.
- You can work with the representation to make descriptions or predictions about the underlying subject of the model.
- Using the model is easier or more convenient to work with, or more likely to lead to new insights than working with the original.

More Types of Oracles

Based on Notes from Doug Hoffman

Oracles	Description	Advantages	Disadvantages
State Model	● We can represent programs as state machines. At any time, the program is in one state and (given the right inputs) can transition to another state. The test provides input and checks whether the program switched to the correct state	● Good software exists to help test designer build the state model ● Excellent software exists to help test designer select a set of tests that drive the program through every state transition	● Maintenance of the state machine can be very expensive (e.g. the model changes when the program's UI changes.) ● Does not (usually) try to drive the program through state transitions considered impossible ● Errors that show up in some other way than bad state transition can be invisible to the comparator
Interaction Model	● We know that if the SUT does X, some other part of the system (or other system) should do Y and if the other system does Z, the SUT should do A	● To the extent that we can automate this, we can test for interactions much more thoroughly than manual tests	● We are looking at a slice of the behavior of the SUT so we will be vulnerable to misses and false alarms ● Building the model can take a lot of time. Priority decisions are important

More Types of Oracles

Based on Notes from Doug Hoffman

Oracles	Description	Advantages	Disadvantages
Calculation Oracle	- We use calculation oracles to check the calculations of a program. For example: ○ if the program adds 5 numbers, we can use some other program or library to add the 5 numbers and see what we get ○ or do the same calculations in different ways e.g. 2*3=3*2=2+2+2=3+3	- Good for ○ mathematical functions	- To obtain the predictable results, we might have to create a difficult-to-implement reference program if we can't find a library that we can use to do the calculations. - Available only for computationally predictable results
Inverse Oracle	- The inverse oracle is often a special case of a calculation oracle (the square of the square root of 2 should be 2) but not always. For example ○ imagine taking a list that is sorted low to high, sorting it high to low and then sorting it low to high. Do we get back the same list?	- Good for ○ straightforward transformations ○ invertible operations of any kind	- Available only for invertible operations

More Types of Oracles

Based on Notes from Doug Hoffman

Oracles	Description	Advantages	Disadvantages
Business Model	• We understand what is reasonable in this type of business. For example, • We might know how to calculate a tax (or at least that a tax of $1 is implausible if the taxed event or income is $1 million) • We might know inventory relationships. It might be absurd to have 1 box top and 1 million bottoms	• These oracles are probably expressed as equations or as plausibility-inequalities (“it is ridiculous for A to be more than 1000 times B”) that come from subject-matter experts. Software errors that violate these are probably important (perhaps central to the intended benefit of the application) and likely to be seen as important	• There is no completeness criterion for these models • The subject matter expert might be wrong in the scope of the model (under some conditions, the oracle should not apply and we get a false alarm) • Some models might be only temporarily true
Theoretical (e.g. Physics Or Chemical) Model	• We have theoretical knowledge of the proper functioning of some parts of the SUT. For example, we might test the program’s calculation of a trajectory against physical laws	• Theoretically sound evaluation • Comparison failures are likely to be seen as important	• Theoretical models (e.g. physics models) are sometimes only approximately correct for real-world situations

More Types of Oracles

Based on Notes from Doug Hoffman

Oracles	Description	Advantages	Disadvantages
Reference Program	• Generates the same responses to a set of inputs as expected from the SUT • The behavior of the reference program will differ from the SUT in some ways (they would be identical in all ways only if they were the same program), e.g. ○ the time it takes to add 1000 numbers might be different in the reference program versus the SUT, but if they yield the same sum, we can say that the SUT passed the test.	• Gives a straightforward mechanism for determining whether the program passed or failed a test, especially when evaluating the result of a test would be a complex task, e.g. ○ when the program uses complex algorithms to calculate the correct result.	• Testers might miss bugs if the reference program contains the same bug as the SUT
Statistical	• Checks against probabilistic predictions, such as: ○ 80% of online customers have historically been from these ZIP codes; what is today's distribution? ○ X is usually greater than Y ○ X is positively correlated with Y	• Allows checking of very large data sets • Allows checking of live systems' data • Allows checking after the fact	• False alarms and misses are both likely (Type 1 and Type 2 errors) • Can miss obvious errors

More Types of Oracles

Based on Notes from Doug Hoffman

Oracles	Description	Advantages	Disadvantages
Data Set with Known Characteristics	• Rather than testing with live data, create a data set with characteristics that you know thoroughly. Oracles may or may not be explicitly built in (they might be) but you gain predictive power from your knowledge	• The test data exercise the program in the ways you choose (e.g. limits, interdependencies, etc.) and you (if you are the data designer) expect to see outcomes associated with these built-in challenges • The characteristics can be documented for other testers • The data continue to produce interesting results despite many types of program changes	• Known data sets do not themselves provide oracles • Known data sets are often not studied or not understood by subsequent testers (especially if the creator leaves) creating Cargo Cult level testing
Hand Crafted	• Result is carefully selected by test designer	• Useful for some very complex SUTs • Expected result can be well understood	• Slow, expensive test generation • High maintenance cost • Maybe high test creation cost
Human	• A human decides whether the program is behaving acceptably	• Sometimes this is the only way. “Do you like how this looks?” “Is anything confusing?”	• Slow and subjective • Credibility varies with the credibility of the human

Summing Up

- Test oracles can only sometimes provide us with authoritative failures.
- Test oracles cannot tell us whether the program has passed the test, they can only tell us it has not obviously failed.
- Oracles subject us to two possible classes of errors:
  - Miss: The program fails but the oracle doesn’t expose it
  - False Alarm: The program did not fail but the oracle signaled a failure

Tests do not provide complete information. They provide partial information that might be useful.

About the Exam

It’s time to start working through the Exam Prep questions. You’ll learn more by working through a few questions each week than by cramming just before the exam.

The goal is to help you focus your studying and to think carefully through your answers:

- Early work helps you identify confusion or ambiguity
- Early drafting makes peer review possible

Note: The exam is closed book, and takes all of its questions from the Exam Prep set.

Black Box Software Testing Foundations
Lecture 4
Programming Fundamentals and Coverage

Cem Kaner J.D., PH.D.

Professor Emeritus, Software Engineering, Florida Institute of Technology

Course Overview: Fundamental Topics

1. The Nature of Testing
   Overview and Basic Definitions

2. Why are we testing? What are we trying to learn? How should we organize our work to achieve this? ***Information objectives drive the testing mission and strategy***

3. How can we know whether a program has passed or failed a test?
   Oracles are heuristic

4. How can we determine how much testing has been done? What core knowledge about program internals do testers need to consider this question? ←
   Coverage is a multidimensional problem

5. Are we done yet?
   Complete testing is impossible

6. How much testing have we completed and how well have we done it?
   Measurement is important but hard

Today’s Readings

Required

- Cem Kaner (1995), “Software Negligence & Testing Coverage”, https://kaner.com/pdfs/negligence_and_testing_coverage.pdf
- Brian Marick (1997), “How to Misuse Code Coverage”, http://www.exampler.com/testing-com/writings/coverage.pdf

Useful to skim

- David Goldberg (1991), “What Every Computer Scientist Should Know About Floating-Point Arithmetic”, https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
- Brian Marick (1991), “Experience With the Cost of Different Coverage Goals for Testing”, http://www.exampler.com/testing-com/writings/experience.pdf
- Charles Petzold (1993), *Code: The Hidden Language of Computer Hardware and Software.*

Computing Fundamentals

Why teach this material now?

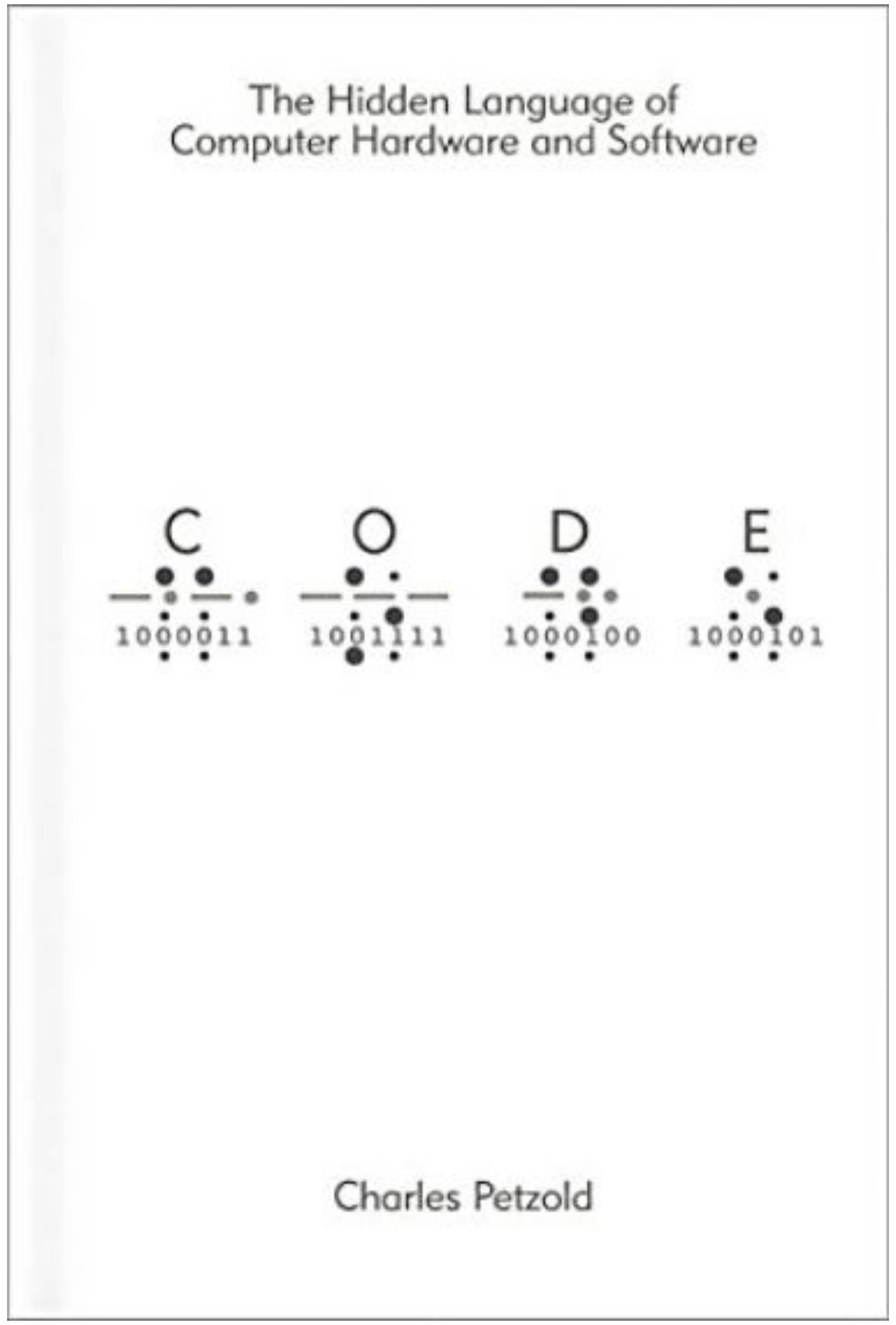
Most discussions of “coverage” in testing involve structural coverage.

To understand what people are talking about:

- what these types of coverage actually measure
- what types of tests people emphasize in order to achieve coverage
- what risks are **not** addressed by these types of coverage

you need a bit of knowledge of data representation and program structure.

Computing Fundamentals



The Hidden Language of Computer Hardware and Software

C O D E

1000011 1001111 1000100 1000101

Charles Petzold

How Do Computers Store Data?

- Basic storage and arithmetic (Decimal)
    - Decimal numbers
    - Addition
    - Overflow
    - Integers vs floating point
- Basic storage and arithmetic (Binary)
    - Representation
    - Addition
    - Overflow
    - Floating point
- Alphanumeric and other characters

Decimal Numbers

Digits: We have 10 of them.

So we can count:

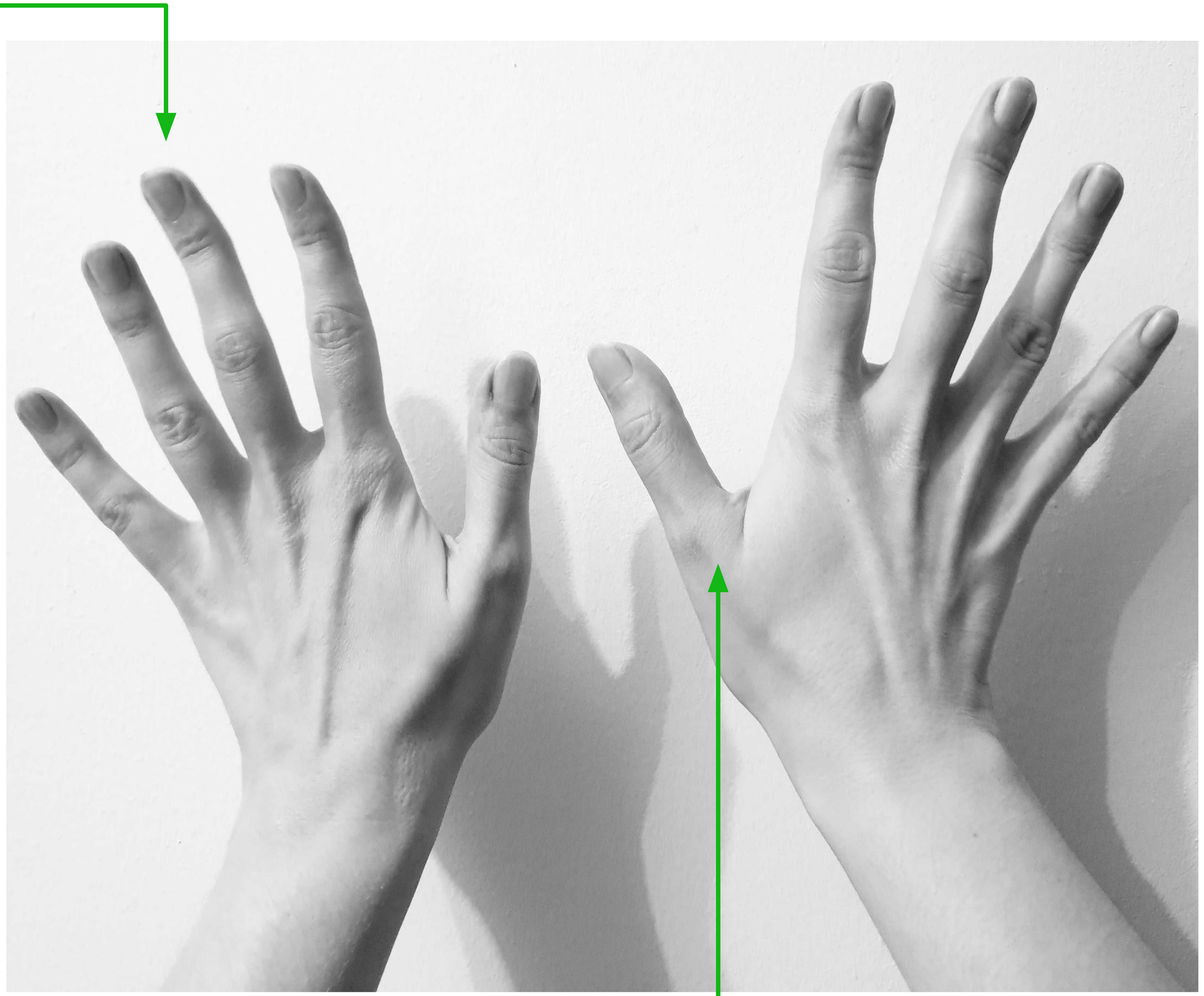
1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

But instead of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

we use the following

Decimals numerals:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9.



Decimal Numbers

Decimal arithmetic:

- “Decimal” refers to 10 (like counting on your 10 fingers).
- Base 10 arithmetic represent numbers as a sum of powers of 10:
  - $10^0 = 1$
  - $10^1 = 10$
  - $10^2 = 10 \times 10 = 100$
  - $10^3 = 10 \times 10 \times 10 = 1000$

$10 \quad = 1 \times 10^1 + 0 \times 10^0$

$954 = 9 \times 10^2 + 5 \times 10^1 + 4 \times 10^0$

(Special case: 0 = 0)

Adding Decimal Numbers

Consider

$654 = 6 \times 10^2 + 5 \times 10^1 + 4 \times 10^0$

$243 = 2 \times 10^2 + 4 \times 10^1 + 3 \times 10^0$

To add them,

- Add 4×10^0 and $3 \times 10^0 = 7 \times 10^0$
- Add 5×10^1 and $4 \times 10^1 = 9 \times 10^1$
- Add 6×10^2 and $2 \times 10^2 = 8 \times 10^2$

So, 654 + 243 = 897

Overflow

Consider the following 1-digit decimal numbers.

$6 = 6 \times 10^0$

$7 = 7 \times 10^0$

To add them,

- 6 + 7 is larger than the largest decimal numeral
- $6 + 7 = 6 + (4 + 3)$

  $= (6 + 4) + 3$

  $= 10 + 3$

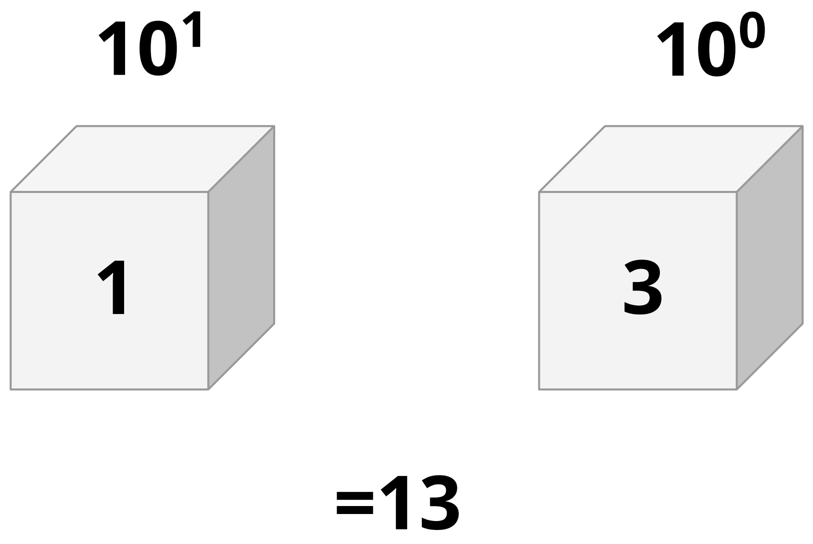
  $= 1 \times 10^1 + 3 \times 10^0$

  $= 13$

  a 2-digit decimal number

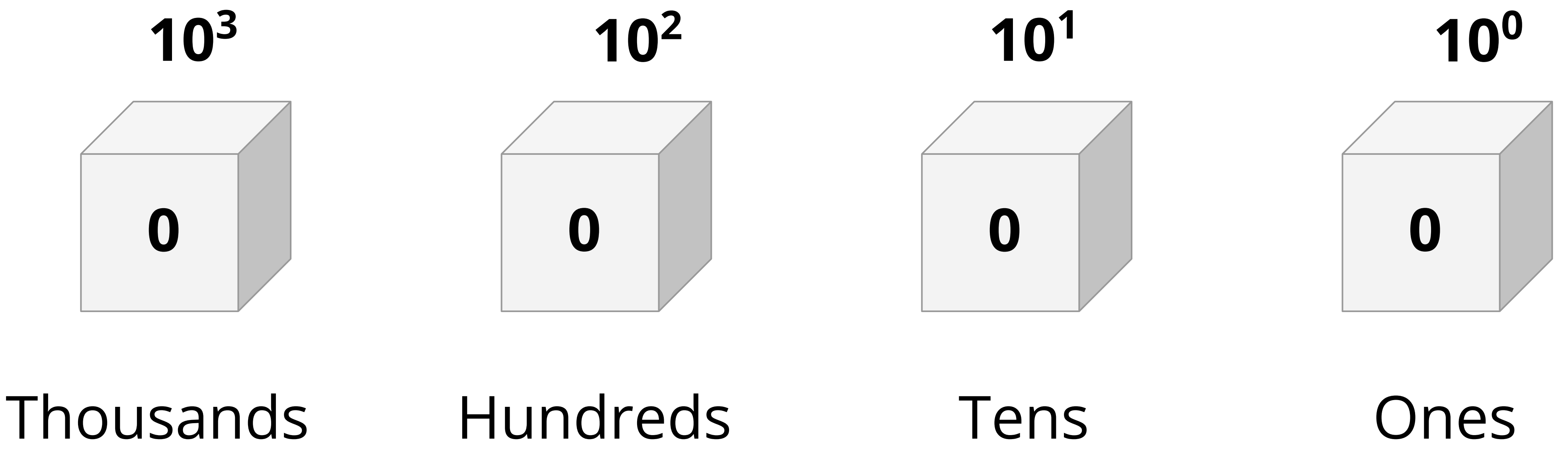
We "carry the 1". That is, we add 1 times the next higher power of 10

Overflow



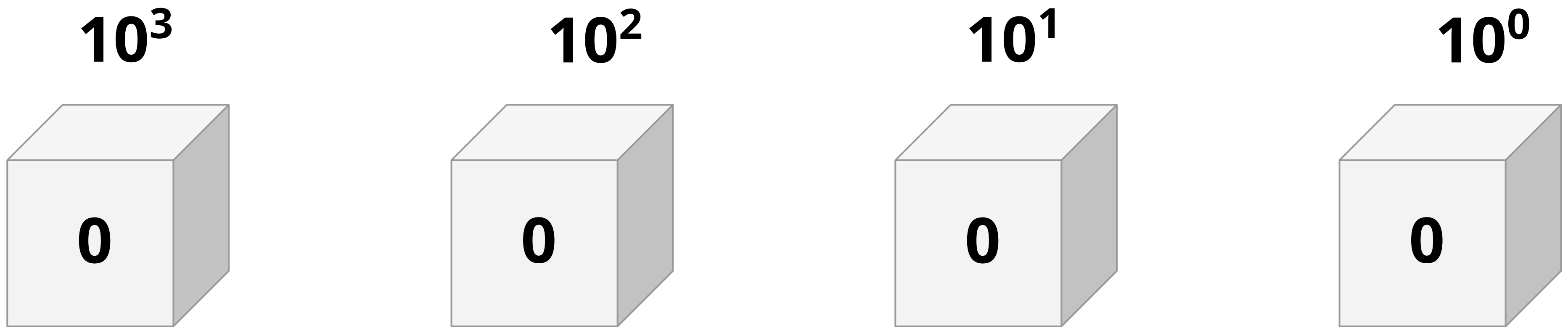
Rather than counting fingers (and toes), let's imagine boxes with 10 sides (labeled 0 through 9). We carry the 1, give us 1 in the tens' box (10^1) and 3 in the ones' box (10^0).

Even Bigger Numbers

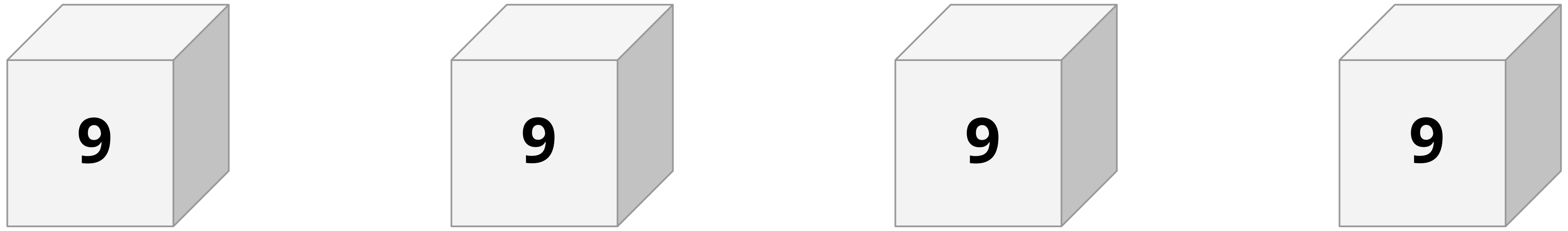


Overflow

If we store numbers in a device that can handle up to four decimal digits, we can store numbers from:

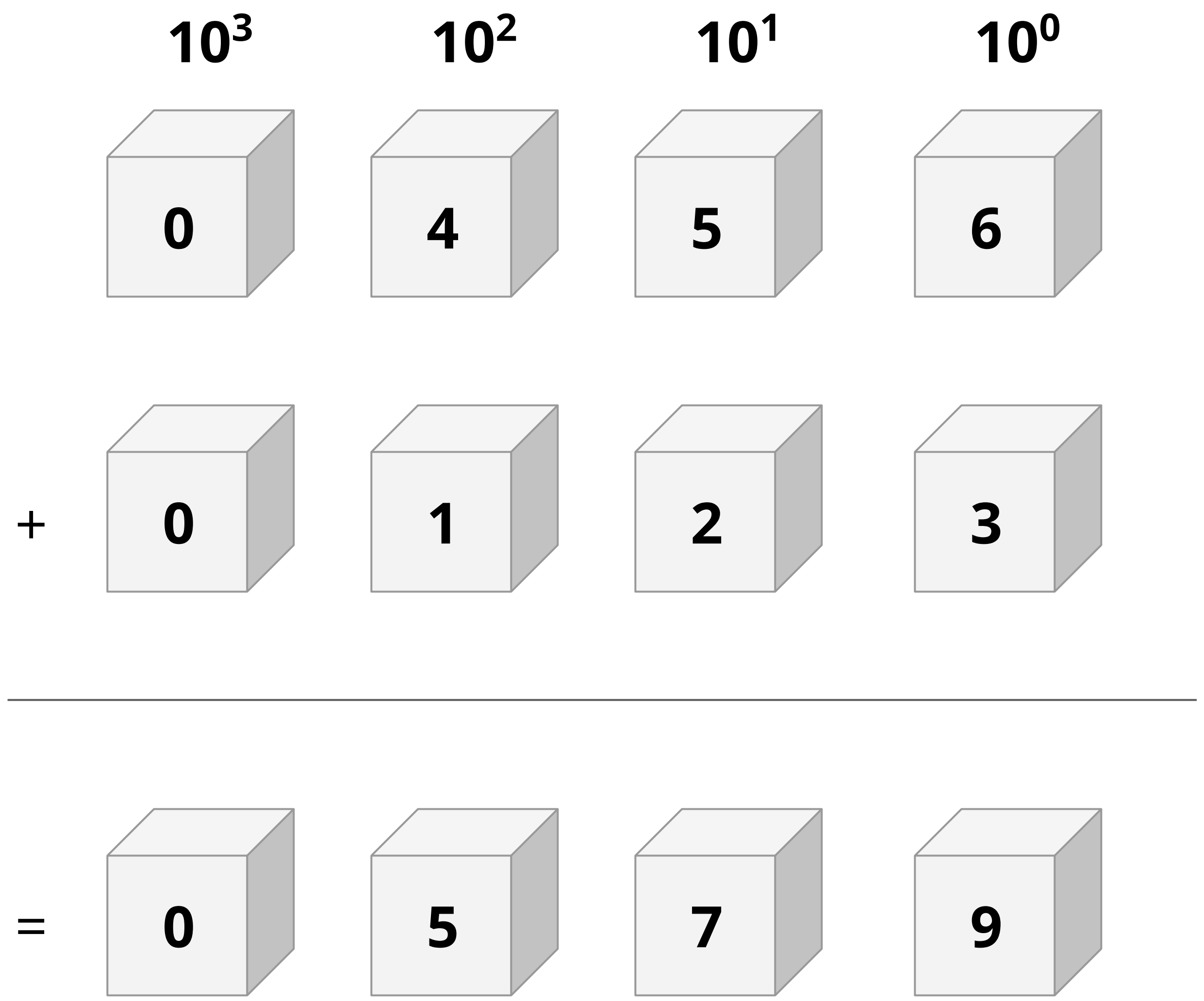


to:



Overflow

We can add numbers



	10^3	10^2	10^1	10^0
	0	4	5	6
+	0	1	2	3
=	0	5	7	9

Overflow

We can add numbers that overflow a digit. For example:

	10^3	10^2	10^1	10^0
	0	6	7	8
+	0	7	5	3
=	1	4	3	1

$8 \text{ x } 10^0 + 3 \text{ x } 10^0$ $= 1 \text{ x } 10^1 + \mathbf{1 \text{ x } 10^0}$

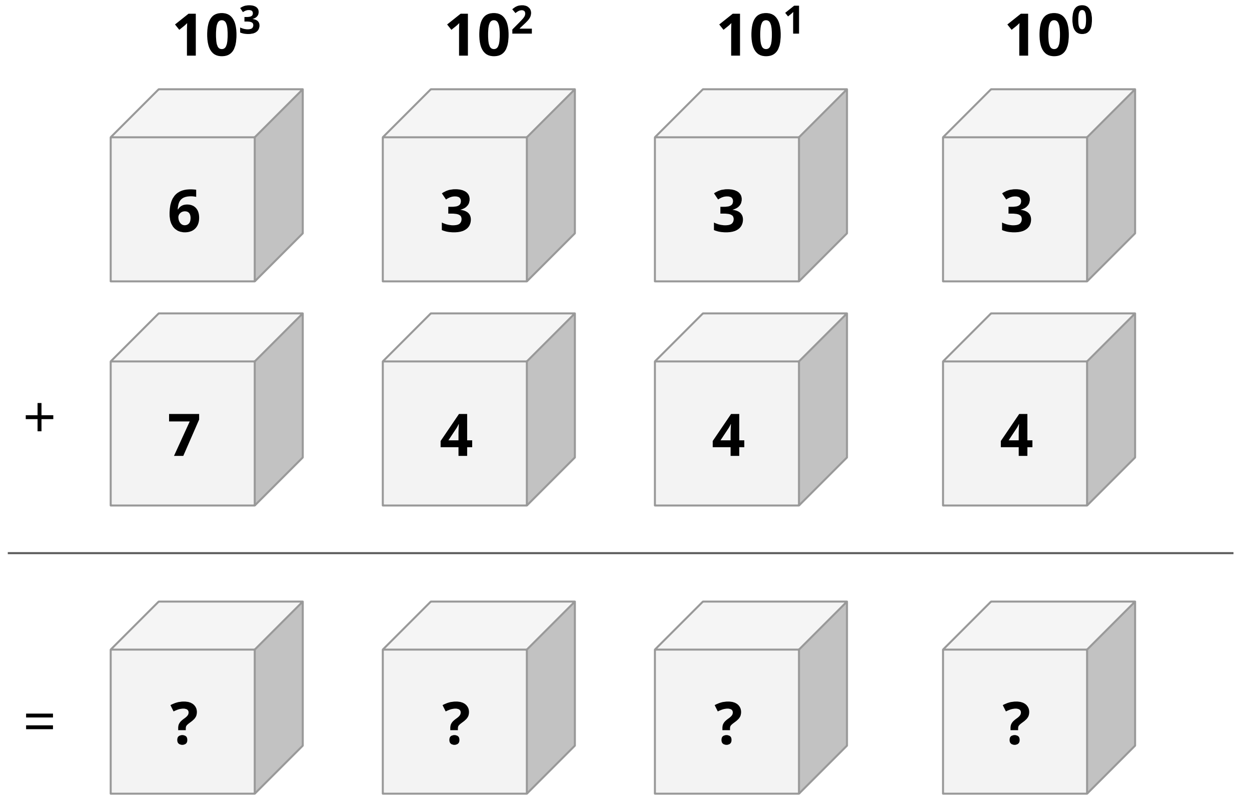
$7 \text{ x } 10^1 + 5 \text{ x } 10^1 + 1 \text{ x } 10^1$ (carried from 8+3) $= 1 \text{ x } 10^2 + \mathbf{3 \text{ x } 10^1}$

$6 \text{ x } 10^2 + 7 \text{ x } 10^2 + 1 \text{ x } 10^2$ (carried from 7+5+1) $= 1 \text{ x } 10^3 + \mathbf{4 \text{ x } 10^2}$

$0 \text{ x } 10^3 + 0 \text{ x } 10^3 + 1 \text{ x } 10^3$ (carried from 6+7+1) $= \qquad + \mathbf{1 \text{ x } 10^3}$

Overflow

But what happens if we overflow the highest digit?



	10^3	10^2	10^1	10^0
	6	3	3	3
+	7	4	4	4
=	?	?	?	?

3 x 10^0 + 4 x 10^0 = 0 x 10^1 + **7 x 10^0**

3 x 10^1 + 4 x 10^1 + 0 x 10^1 (nothing carried) = 0 x 10^2 + **7 x 10^1**

3 x 10^2 + 4 x 10^2 + 0 x 10^2 (nothing carried) = 0 x 10^3 + **7 x 10^2**

6 x 10^3 + 7 x 10^3 + 0 x 10^3 (nothing carried) = **1 x 10^4** + **3 x 10^3**

BUT WE DON'T HAVE A 10^4 DIGIT

We Can Also Represent Fractions

Base 10 arithmetic represent numbers as a sum of powers of 10:

- 10^{-3} = 1/1000
- 10^{-2} = 1/100
- 10^{-1} = 1/10
- 10^{0} = 1
- 10^{1} = 10
- 10^{2} = 100
- 10^{3} = 1000

0.02345 $= 2 \times 10^{-2} + 3 \times 10^{-3} + 4 \times 10^{-4} + 5 \times 10^{-5}$

0.2345 $= 2 \times 10^{-1} + 3 \times 10^{-2} + 4 \times 10^{-3} + 5 \times 10^{-4}$

2.345 $= 2 \times 10^{0} + 3 \times 10^{-1} + 4 \times 10^{-2} + 5 \times 10^{-3}$

23.45 $= 2 \times 10^{1} + 3 \times 10^{0} + 4 \times 10^{-1} + 5 \times 10^{-2}$

234.5 $= 2 \times 10^{2} + 3 \times 10^{1} + 4 \times 10^{0} + 5 \times 10^{-1}$

2345. $= 2 \times 10^{3} + 3 \times 10^{2} + 4 \times 10^{1} + 5 \times 10^{0}$

Fixed Point Representation

In a fixed-point representation, the decimal point stays “fixed” (same place) no matter how large or small the number.

$0.02345 = 2 \text{ x } 10^{-2} + 3 \text{ x } 10^{-3} + 4 \text{ x } 10^{-4} + 5 \text{ x } 10^{-5}$

$0.2345 = 2 \text{ x } 10^{-1} + 3 \text{ x } 10^{-2} + 4 \text{ x } 10^{-3} + 5 \text{ x } 10^{-4}$

$2.345 = 2 \text{ x } 10^{0} + 3 \text{ x } 10^{-1} + 4 \text{ x } 10^{-2} + 5 \text{ x } 10^{-3}$

$23.45 = 2 \text{ x } 10^{1} + 3 \text{ x } 10^{0} + 4 \text{ x } 10^{-1} + 5 \text{ x } 10^{-2}$

$234.5 = 2 \text{ x } 10^{2} + 3 \text{ x } 10^{1} + 4 \text{ x } 10^{0} + 5 \text{ x } 10^{-1}$

$2345.0 = 2 \text{ x } 10^{3} + 3 \text{ x } 10^{2} + 4 \text{ x } 10^{1} + 5 \text{ x } 10^{0}$

$23450.0 = 2 \text{ x } 10^{4} + 3 \text{ x } 10^{3} + 4 \text{ x } 10^{2} + 5 \text{ x } 10^{1}$

$234500.0 = 2 \text{ x } 10^{5} + 3 \text{ x } 10^{4} + 4 \text{ x } 10^{3} + 5 \text{ x } 10^{2}$

Fixed Point Representation

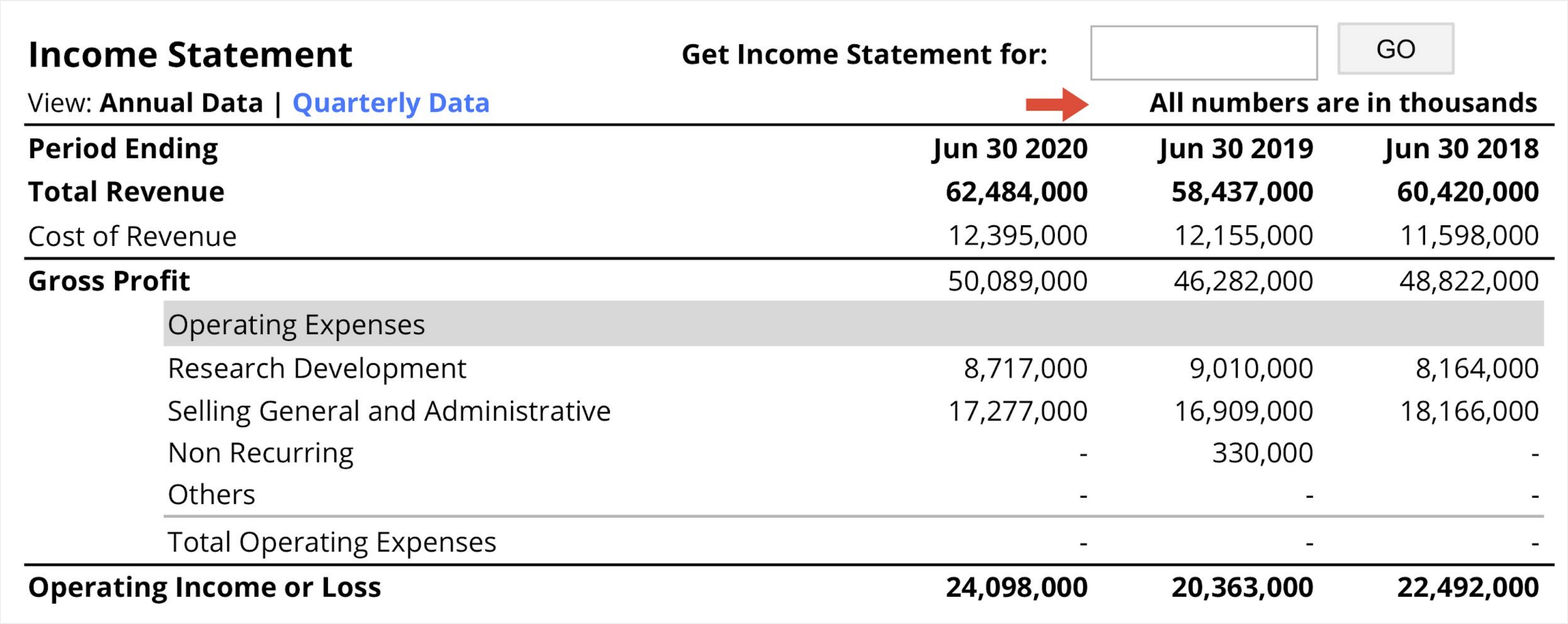
Fixed point representation in a computer is essentially the same as integer storage.

- We have a limited set of number blocks and we can't go beyond them.
- We call these our “significant digits”.
- The difference is that we get to choose (once, for all numbers) where the decimal point goes.
- For example, $1234.56 is a **six-significant-digit** fixed-point number. We cannot represent a number larger than $9999.99 or currency subdivisions finer than a penny (1/100th).

Fixed Point Representation

In fixed-point, we can choose where we place the decimal point.

Here for example: **“All numbers in thousands”**.



Income Statement

All numbers are in thousands

Period Ending	**Jun 30 2020**	**Jun 30 2019**	**Jun 30 2018**
Total Revenue	**62,484,000**	**58,437,000**	**60,420,000**
Cost of Revenue	12,395,000	12,155,000	11,598,000
Gross Profit	50,089,000	46,282,000	48,822,000
Operating Expenses			
Research Development	8,717,000	9,010,000	8,164,000
Selling General and Administrative	17,277,000	16,909,000	18,166,000
Non Recurring	-	330,000	-
Others	-	-	-
Total Operating Expenses	-	-	-
Operating Income or Loss	**24,098,000**	**20,363,000**	**22,492,000**

Floating Point

2.345

$= 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} + 5 \times 10^{-3}$

$= 10^{-3} \times 2345$

$= 10^{-3} \times (2 \times 10^3 + 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0)$

So we can represent “any” 4-digit number

- as an Integer (a number with no decimal point).
- multiplied by 10 to the appropriate power.

In 2345×10^{-3}

- 2345 is called the **mantissa** or the **significand**
- there are 4 **significant digits**
- 10 is called the **base**
- -3 is called the **exponent**

$2{,}345{,}000{,}000 = 2345 \times 10^6$

A significant digit is a digit we allow to have a value other than zero.

Floating Point

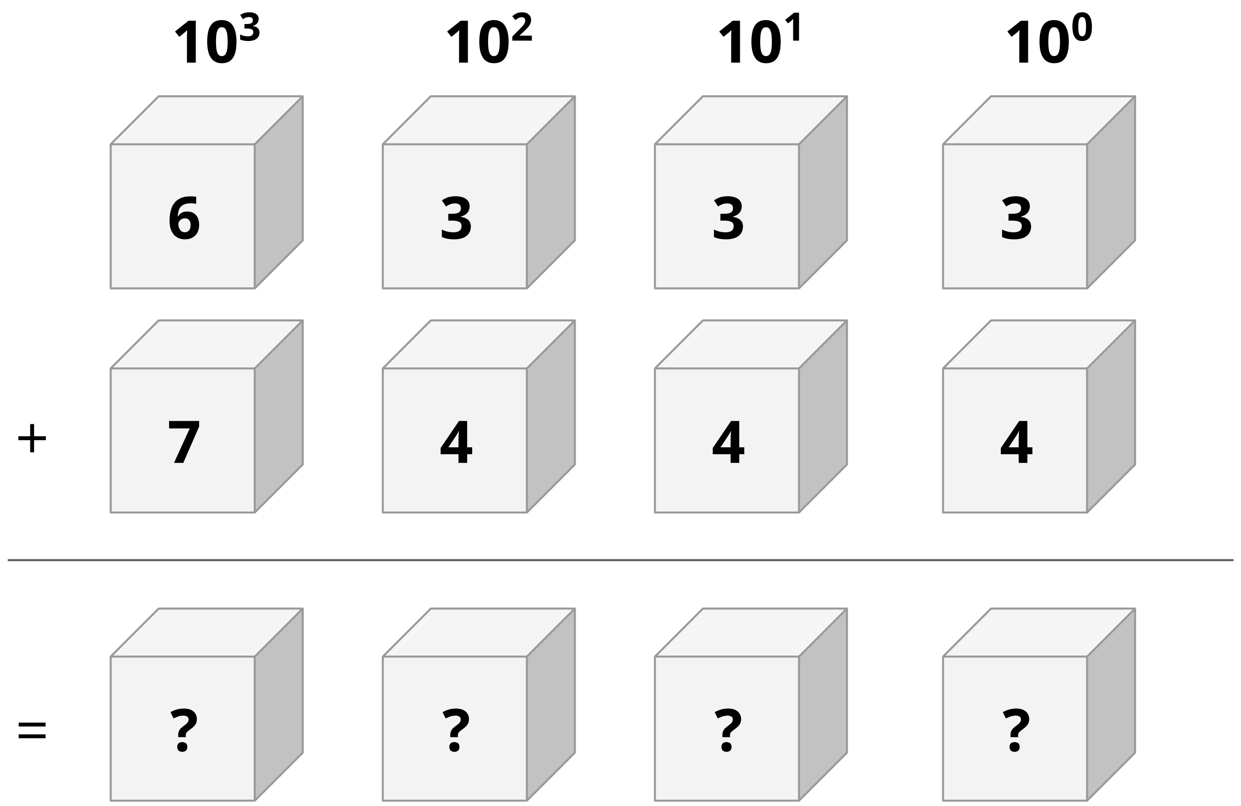
As a matter of convention, we usually show the mantissa with a decimal point after the most significant digit:

0.02345	$= 2.345 \times 10^{-2}$
2.345	$= 2.345 \times 10^{0}$
2345.	$= 2.345 \times 10^{3}$
234500000	$= 2.345 \times 10^{8}$

- Each number has 4 significant digits
- Each has the same mantissa (2.345)
- Each has the same base
- Only the exponent is varying

Overflow and Floating Point

Now consider this example again:



	10^3	10^2	10^1	10^0
	6	3	3	3
+	7	4	4	4
=	?	?	?	?

The sum is 13777, which overflows the 4 significant digits.

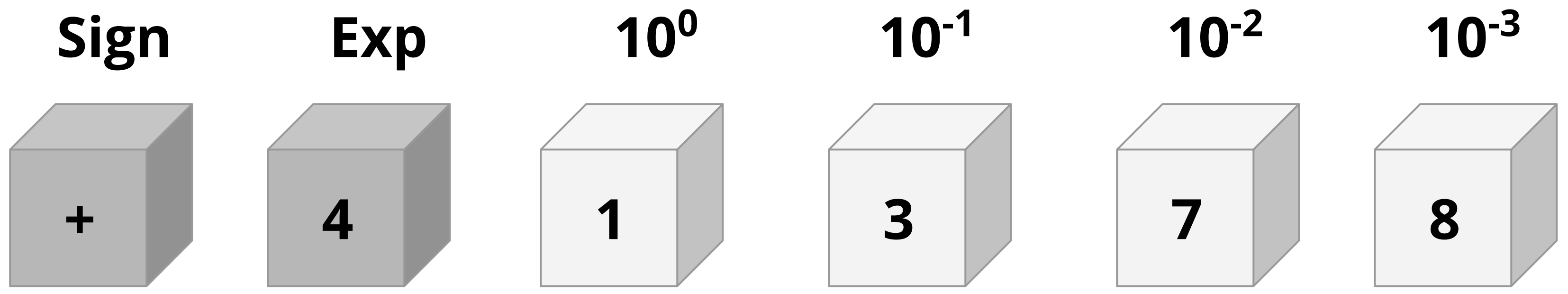
In floating point notation, it is 1.3777 x 10^4.

This is still too many digits, but we can round up: 1.378 x 10^4.

Overflow, Floating Point and Rounding

We still can’t represent 1.378 x 10^4 in four digits.

But what if we added a box for the exponent and a box for the exponent’s sign?



Sign	Exp	10^0	10^{-1}	10^{-2}	10^{-3}
+	4	1	3	7	8

= 10^4 x (1 x 10^0 + 3 x 10^{-1} + 7 x 10^{-2} + 8 x 10^{-3})

In this way, we can represent a number

- as small as 10^{-9} x 1.000 (0.000000001) and
- as large as 10^9 x 9.999 (9999000000.0).

Significant Digits and Precision

We can represent a number

- as small as 10^{-9} x 1.000 (0.000000001)
- as large as 10^{9} x 9.999 (9999000000.0)

However, we have only 4 significant digits.

Suppose we entered:

- 9999000000.0,
- 9999000001.0, and
- 9999499999.0

How will the computer store these?
In a floating point representation with 4 significant digits, all would be stored the same way, as 9.999 x 10^{9}.

Significant Digits and Precision

Which is the bigger error?

Saying that 2.000 is 1.9999?

OR

Saying that 1.99975 is 2.000?

Overflow, Floating Point and Rounding

- If we want to represent a larger range of numbers, we can change the number of digits in the Exponent.
  - With two digits, we can go from 10^{-99} x 0001 to 10^{99} x 9999
- If we want negative numbers (in the significand) as well as positive, we can add another box for the sign of the main number.

Sign	Exp	Exp	Sign	10^3	10^2	10^1	10^0
-	1	1	+	1	3	7	8

10^{99} x -9999 to

10^{-99} x -1 to

0 to

10^{-99} x 1 to

10^{99} x 9999

Floating Point and Rounding Error

With this system for representing numbers,

a) What is the sum of

$$\begin{array}{rl} & 1234 \times 10^{10} \\ + & 5678 \times 10^{-10} \quad ? \end{array}$$

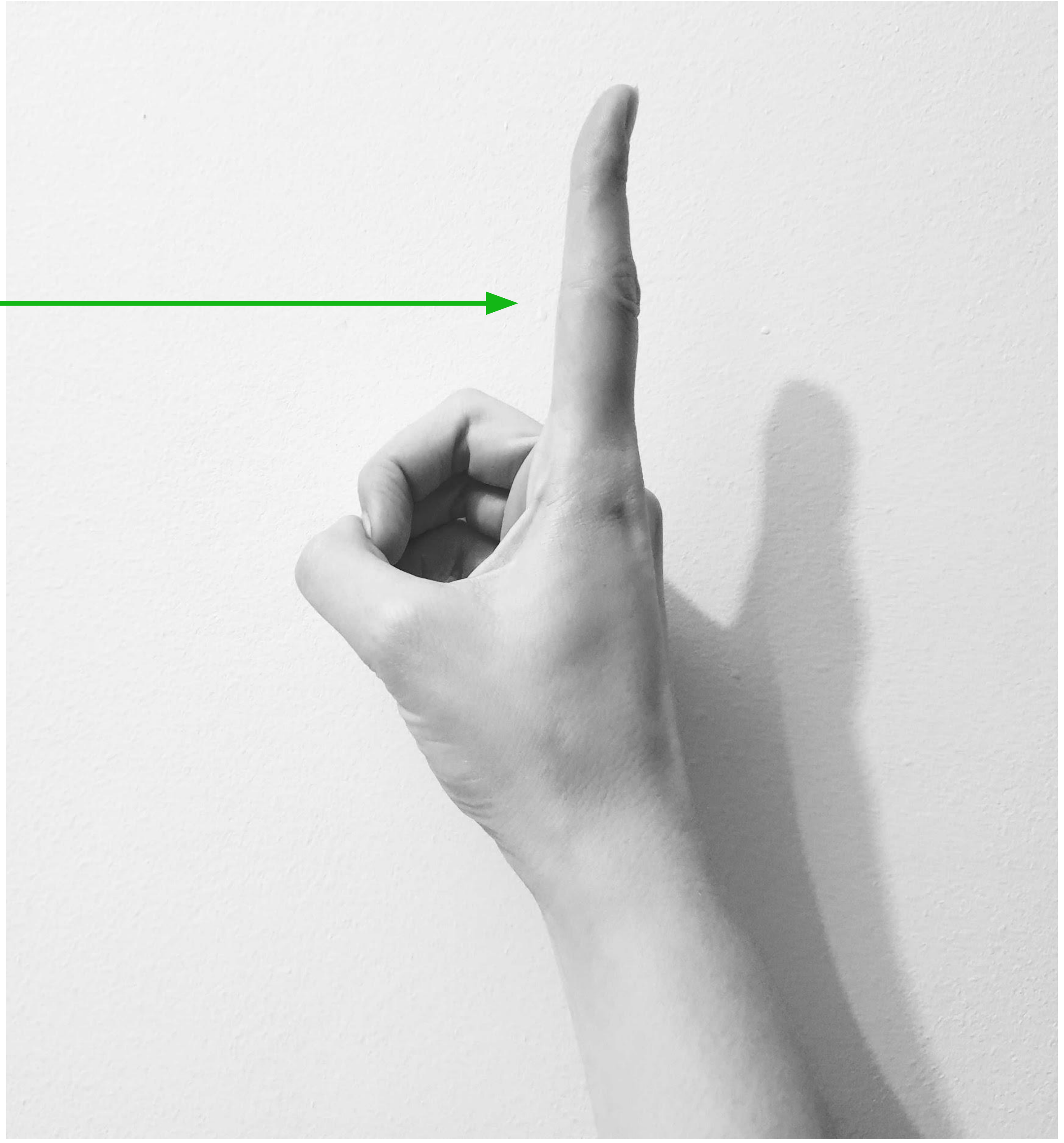
b) What is the product of

$$\begin{array}{rl} & 1234 \times 10^{10} \\ \times & 12 \quad ? \end{array}$$

In that multiplication, assume we could never store more than 4 digits at any time.

Binary Numbers

- A simpler approach to digits:
  - Instead of counting from 0 to 9 (decimal), consider counting from **0 to 1 (binary).**
- We call binary digits “bits”.
- We also call the physical portion of computer memory needed to store a 0-or-1 a “bit”.



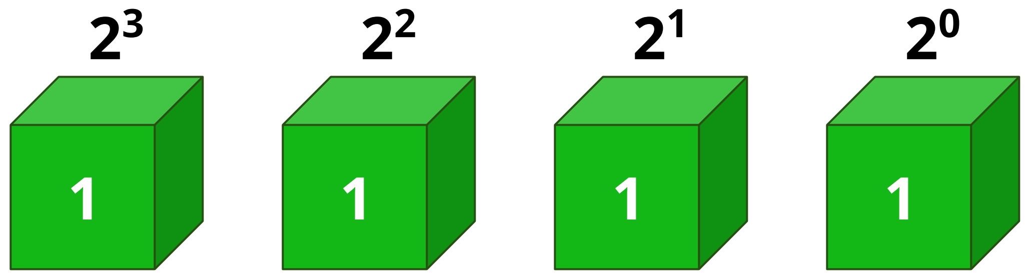
Binary Arithmetic

Binary refers to 2 (like counting with one finger that goes up or down).

Base 2 arithmetic represent numbers as a sum of powers of 2:

- $2^0 = 1$
- $2^1 = 2$
- $2^2 = 4$
- $2^3 = 8$

$15 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$



Bytes (8 Bits)

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
32	0	0	1	0	0	0	0	0
64	0	1	0	0	0	0	0	0
128	1	0	0	0	0	0	0	0
132	1	0	0	0	0	1	0	0
255	1	1	1	1	1	1	1	1

Adding Binary Numbers

Consider 8-bit binary numbers:

1 + 1 = 00000001 (1)
+ 00000001 (1)
= 00000010 (2)

Just like 5+5 = 10 (carry the 1) in decimal arithmetic (because there is no digit bigger than 9), 1+1 = 10 in binary (because there is no digit bigger than 1).

17 + 15 = 00010001 (17)
+ 00001111 (15)
= 00100000 (32)

Overflow

- The biggest number you can fit in a byte is 11111111 = 255.

  255 + 1 = 11111111 (255)

          + 00000001 (1)

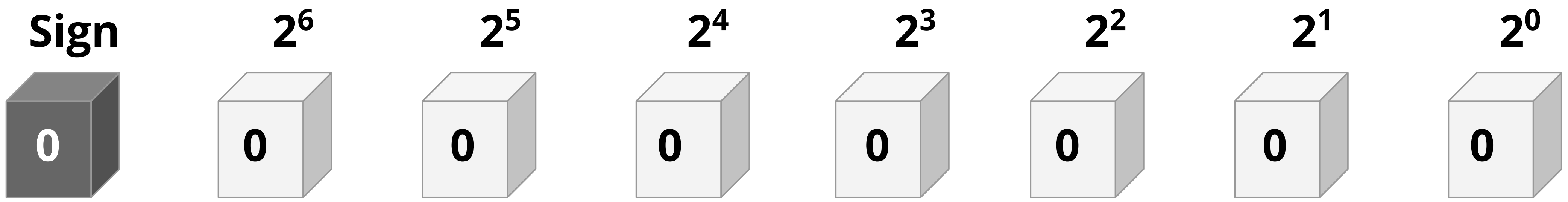
          = overflow (256)
- To deal with larger numbers, we either have to work with larger areas of memory (such as 16-bit or 32-bit words) or we have to work with floating point.
- We'll address both soon…
- But first, let's consider positives and negatives.

"In computing, *word* is a term for natural unit of data used by a particular computer design. A word is simply a fixed sized group of bits that are handled together by the system. The number of bits in word (the word size or word length) is an important characteristic of computer architecture."

https://en.wikipedia.org/wiki/Word_(computer_architecture)

Unsigned Versus Signed

Rather than interpreting the first (leftmost) bit in a binary number as a digit, we can interpret it as a sign bit.



Sign	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	0	0	0	0	0

- 00000001 is 1
- 01111111 is 127
- 10000000 is -128
- 11111111 is -1

💡 See https://en.wikipedia.org/wiki/Two%27s_complement

8, 16, 32 & 64-Bit Words

- Computers read memory several bits at a time.
- A **word** is the amount of memory typically read in one fetch operation.
  - The Apple 2 computers read memory 1 byte at a time. Its word size was 1 byte.
  - The original IBM computers read memory in blocks of 16 bits at a time. Their word size was 16 bits.
  - Most modern computers operate on 32 or 64 bits at a time, so their word size is 32 or 64 bits.

Words

A computer with a 32-bit word size can operate on smaller blocks of memory.

- It might work with 8 or 16 bits.
- Unless you are testing new chips in development, the program is **unlikely** to
  - read or work with the wrong number of bits.
  - read or write one too many or one too few bytes (it will read 8, not 7 or 9).

Integers

Textbook examples for integers typically use 8-bit or 16-bit words.

- With 16 bits, MaxInt is
    - 32767 with signed integers
    - 65535 with unsigned integers
- MinInt is
    - -32768 if integers are signed
    - 0 if integers are unsigned

It is usually a mistake to assume you know the value of MaxInt. Even if you know it today, the system will change and MaxInt will change with it. Design your tests (and code) using MinInt and MaxInt, not numeric constants.

Integers: Java

	Size		Range	
	Bytes	Bits	MinInt	MaxInt
byte	1	8	-128	127
short	2	16	-32,768	32,767
int	4	32	-2,147,483,648	2,147,483,647
long	8	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Floating Point (Single Precision)

32-bit number		
Leftmost bit: sign bit	**Next 8 bits: exponent**	**Next 23 bits: mantissa (aka significand)**
0 is positive 1 is negative	-126 to 127 (see note)	1.175494351 x 10^{-38} to 3.402823466 x 10^{38}

Note: Exponent values of 0 and 255 have special meanings. For details, see discussions of the IEEE specification for floating point numbers.

💡 See https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html and Petzold, *Code*, Chapter 23.

Floating Point (Double Precision)

64-bit number		
Leftmost bit: **sign bit**	**Next 11 bits:** **exponent**	**Next 52 bits:** **mantissa**
0 is positive 1 is negative	-1022 to 1023 (see note)	2.2250738585072014 x 10^{-308} to 1.7976931348623158 x 10^{308}

Note: Exponent values of 0 and 2047 have special meanings. For details, see discussions of the IEEE specification for floating point numbers.

See https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html and Petzold, *Code*, Chapter 23.

Hexadecimal Numbers

Decimal	Binary	Hexadecimal	Decimal	Binary	Hexadecimal
0	0000	0	16	00010000	10
1	0001	1	17	00010001	11
2	0010	2	31	00011111	1F
3	0011	3	32	00100000	20
4	0100	4	63	00111111	2F
5	0101	5	64	01000000	40
6	0110	6	100	01100100	64
7	0111	7	112	01110000	70
8	1000	8	128	10000000	80
9	1001	9	144	10010000	90
10	1010	A	160	10100000	A0
11	1011	B	176	10110000	B0
12	1100	C	192	11000000	C0
13	1101	D	208	11010000	D0
14	1110	E	224	11100000	E0
15	1111	F	255	11111111	FF

Alphanumeric Representation: ASCII

- American Standard code for Information Interchange
- Encoding for teletypes
  - Code 7 says to ring the TTY bell
  - Code 11 calls for a vertical tab
- Codes 0 to 31 are commands (non-printing characters)
- Code 32 is for Space character
- Code 33 is for !
- Code 47 is for /
- Codes 48 – 57 are for digits 0 to 9
- Codes 65 – 90 for A to Z
- Codes 97 – 122 for a to z

Desaturated and cropped. Original photo: https://www.flickr.com/photos/ajmexico/4669611994/

See http://www.asciitable.com and http://en.wikipedia.org/wiki/Unicode

Same Data, Different Meanings

- What a bit pattern in memory means depends on how the program reading it interprets it.
- The same bit pattern might be:
  - An integer
  - A floating point number
  - A character or sequence of characters
  - A command
  - An address (identifies a location in memory)
- The same pattern in the same location might be read differently by different functions.

Data Structures

- A data structure is a way of organizing data. We select a data structure to optimize some aspect of how software will work with it.
- So far, we've seen primitive data types:
  - Integers, floating point numbers, characters, bits
- We can group primitives together in meaningful ways, such as:
  - Strings
  - Records
  - Arrays
  - Lists

This scratches the surface. The goal is merely to familiarize you with some of the variety of ideas.

Data Structures: String

- A sequence of characters
- Each character comes from the same alphabet (set of acceptable symbols)
- Commonplace operations:
    - Search for a substring
    - Replace one substring with another
    - Concatenate (add one string to another. For example, One ⊕ string = Onestring)
    - Calculate length
    - Truncate

Common errors

- **Overflow**
- **Match or mismatch**

Data Structures: Record

- Related data, stored together
  - First name
  - Last name
  - Street address
  - City
  - State
  - Country
  - Postal code
  - Identification number
- One record refers to one person
- Each part is called a **field**
- We might show (or input) all the fields of a record in a dialog

Common Operations

- Search among many records (e.g. an array of records)
- Retrieve a record on basis of values of some fields
- Replace values of some fields
  - Sort records

Common errors

- Write to or retrieve wrong record or wrong fields
- Store wrong data
- Overflow or underflow

Data Structures: Array

- Linear sequence of variables of the same type. Each variable in the array is an **element**.
- Examples
  - `a[ ]` is an array of integers, so
    - `a[0]` and `a[1]` etc. are all integers
  - `b[ ]` is an array of records.
    - `b[3].lastName` yields the `lastName` field associated with record number 3
- Common operations
  - Read, write, sort, change

Common errors

- **Read/write past end of the array**
- **Read uninitialized data**
- **Read/write the wrong element**

Data Structures: List

Like arrays,

- A collection of variables of the same type.
- We can read/write an individual variable, such as a single record in a list of records.

Unlike arrays,

- The individual variables might be different sizes. For example, lists of different-length lists.
- Retrieval is not necessarily by element number.
  - Elements in the list are linked to previous/next elements via pointers.
  - Search for match to a field or combination of fields.
- To reach a given element, might have to move through the list until you reach it.

Common errors

- **Search forward when relevant element is behind current place**
- **Read/write past end of list**
- **Incorrectly specify or update pointer to next or previous element**

Control Structures

We'll consider only a few:

- Sequence
- Branch
- Loop
- Function (method) call
- Exception
- Interrupt

Control Structures: Sequence

- A program includes a list of statements.
- If you start executing a sequence, you execute all of its statements.
- Example

```
SET A = 5
SET B = 2
INPUT C FROM KEYBOARD
PRINT A + B + C
```

Control Structures: Branch

- The program decides to execute
  - one statement (or sequence)
  - rather than another
  - based on the value of a logical expression (logical expressions can evaluate to True or False)
- Example:

```
INPUT C FROM KEYBOARD

IF (C < 5)
   PRINT C
ELSE
   PRINT “C IS TOO BIG”
```

Note: “logical expressions” are also often called “Boolean expressions.”

Common errors

- **In a complex branch (CASE or a sequence of IF's), falling through the branches to an inappropriate default because a special case was missed.**
- **Incorrect branching because the logical expression is complex and was misprogrammed**

Control Structures: Loop

- The program repeats the same set of instructions until an exit criterion is met.
- Example:

```
SET A = 5

WHILE (A < 5) {
    PRINT A
    INPUT A FROM KEYBOARD
}
```

- The exit criterion is (`A ⩾ 5`). The loop continues until the user enters a value ⩾ `5` at the keyboard.

Common errors

- **Infinite loop**
- **Loop exercises one time too many or too few**
- **Out of memory**
- **Huge data files, printouts, emails, because loop runs unexpectedly many times**
- **Too slow because it executes a poorly-optimized block of code thousands of times.**

Control Structures: Function Call

A function (or method or procedure or module) is self-contained.

- Can be called from other parts of the program
- Takes an action and/or returns a value
- Examples:
  - `PRINT` is the name of a method that sends its input data to the printer.
  - `R = SQUAREROOT (X)`

    shows a function (`SQUAREROOT`) that accepts value `X` as input and returns the square root of `X` as a new value for `R`.

Common errors

- **Memory leak**
- **Unexpectedly changes global data (or data on disk or data in memory referenced by address)**
- **Fails without notifying caller or caller ignores a failure exit-code**

Control Structures: Exception

While executing a command, there is a failure. For example, while attempting to print, the printer shuts off or runs out of paper.

The Exception returns from the failed task with information about the failure.

- Example:

```
TRY {
   PRINT X
} CATCH (OUT OF PAPER) {
   ALERT USER AND WAIT
   THEN RESUME PRINTING
} CATCH (PRINTER OFF) {
   ABANDON THE JOB
   ALERT USER
}
```

Other examples

- **Divide by zero (invalid calculation)**
- **Access restricted memory area.**

Common error

- **Exceptions often leave variables or stored data in an unexpected state, files open, and other resources in mid-use resulting in a failure later, when the program next tries to access the data or resource**

Control Structures: Interrupt

- **A hardware interrupt** causes the processor to save its state of execution and begin execution of an interrupt handler. These can occur at any time, with the program in any state.
- **Software interrupts** are usually implemented as instructions that cause a context switch to an interrupt handler similar to a hardware interrupt. These occur at a time/place specified by the programmer.

Interrupts are commonly used for computer multitasking, especially in real-time computing. Such a system is said to be interrupt-driven.

Interrupt handlers are code. They can change data, write to disk, etc.

Examples of hardware interrupts

- **Key pressed on keyboard**
- **Disk I/O error message coming back through the driver**
- **Clock signals end of a timed delay**

Common errors

- **Race condition (unexpected processing delay caused by diversion of resources to interrupt)**
- **Stack overflow (interrupt handler stores program state on the stack—too many nested interrupts might blow the stack)**
- **Deadly embrace: You can't do anything with B until A is done, but you can't finish A until you finish servicing B's interrupts**

Coverage

Extent of testing of certain attributes or pieces of the software under test. Example:

- How many statements have we tested?
- Generally, we report a percentage:
    - Number tested, compared to
    - Number that could have been tested.

Extent (or proportion) of testing of a given type that has been completed, compared to the population of possible tests of this type.

Structural Code Coverage

Coverage you can measure by focusing on the control structures of the program. Examples:

- Statement coverage
  - Execute every statement in the program
- Branch coverage
  - Every statement and every branch
- Multi-condition coverage
  - All combinations of the logical expressions

Useful to skim:

- Ammann & Offutt (2008), *Introduction to Software Testing*
- Jorgensen (2008, 3rd ed), *Software Testing: A Craftsman's Approach*
- http://www.exampler.com/testing-com/writings/iview1.htm
- http://sqa.fyicenter.com/art/experience_with_the_cost_of_different_coverage_goals_for_testing.html
- http://www.bullseye.com/coverage.html

Structural Coverage

IEEE Unit Testing Standard is 100% Statement Coverage and 100% Branch Execution (IEEE Std. 982.1-1988, § 4.17, “Minimal Unit Test Case Determination”).

Most companies don’t achieve this (though they might achieve 100% of the code they actually write).

Several people seem to believe that complete statement and branch coverage means complete testing. (Or, at least, sufficient testing.)

Structural Coverage: Examples

```
INPUT A FROM KEYBOARD
INPUT B FROM KEYBOARD

IF (A < 5) {
   PRINT A
}
IF (B == “HELLO”) {
   PRINT B
}
```

Statement coverage

- Execute every statement in the program
- Tests
  - Enter `4` for `A` and `“HELLO”` for `B`

Structural Coverage: Examples

```
INPUT A FROM KEYBOARD
INPUT B FROM KEYBOARD

IF (A < 5) {
   PRINT A
}
IF (B == "HELLO") {
   PRINT B
}
```

Branch coverage

- Every statement and every branch
- Tests
  - Enter 4 for A and "HELLO" for B
  - Enter 6 for A and "GOODBYE" for B

An interrupt forces a branch to the interrupt handler. Can we seriously claim 100% branch coverage if we don't test branches for every interrupt from every instruction?

(Problem: we probably can't do all these tests…)

Structural Coverage: Examples

```
INPUT A FROM KEYBOARD
INPUT B FROM KEYBOARD

IF (A < 5) {
   PRINT A
}
IF (B == “HELLO”) {
   PRINT B
}
```

Multi-condition coverage

- All combinations of the logical expressions:

A	4	4	6	6
B	HELLO	GOODBYE	HELLO	GOODBYE

Complete Coverage?

Consider the following program:

```
Input A // the program accepts any
Input B // integer into A and B
Print A/B
```

A test with `A=2` and `B=1` will cover:

- every statement
- every branch

However, this testing misses a serious error:

- What test is missing?
- What bug is missed?

In a study by Brian Marick, 43% of failures were traced to faults of omission (missing code rather than wrong code)

http://www.exampler.com/testing-com/writings/omissions.html

Complete Coverage?

The last example shows that even if we obtain “complete coverage” (100% statement or branch or multi-condition coverage), we can still miss obvious, critical bugs. This is because these measures are blind to many aspects of the software, such as (to name just a few):

- Unexpected values (e.g. divide by zero)
- Stability of a variable at its boundary values
- Data combinations
- Data flow
- Tables that determine control flow in table-driven code
- Missing code
- Timing
- Compatibility with devices or other software or systems
- Volume or load
- Interactions with background tasks
- Side effects of interrupts
- Handling of hardware faults
- User interface errors
- Compliance with contracts or regulations
- Whether the software actually delivers the benefits or solves the problems it was intended to deliver/solve

Good Tools for Structural Coverage

- There are fine tools for this that are free and easy to use, such as Emma's Jacoco:
  - https://www.eclemma.org/jacoco/
- Programmers can easily check coverage when they test their code.
- Black box testers find it hard to check structural coverage.

Other Coverages

Structural coverage looks at the code from only one viewpoint.

Structural coverage might be the only family of coverage measures you see in programmers’ textbooks or university research papers, but we’ve seen many other types of coverage in real use.

Coverage assesses the extent (or proportion) of testing of a given type that has been completed, compared to the population of possible tests of this type.

Anything you can list, you can assess coverage against.

Track coverage of the things that are most important to your project, whether these are the “standard” coverage measures or not.

For 101 examples, see Kaner, “Software Negligence & Testing Coverage”, https://kaner.com/pdfs/negligence_and_testing_coverage.pdf

Coverage as a Measurement

People optimize what we measure them against, at the expense of what we don’t measure.

- Driving testing to achieve “high” coverage is likely to yield a mass of low-power tests.
- Brian Marick discusses this in “*How to Misuse Code Coverage*,” http://www.exampler.com/testing-com/writings/coverage.pdf

For more on measurement distortion and dysfunction, read Bob Austin’s book, *Measurement and Management of Performance in Organizations.*

Let’s Summarize

Today we took a high-level tour of some of the basic programming concepts:

- How computers store different types of data
- Binary arithmetic and the challenge of rounding errors in floating point arithmetic
- Flow of control in the program (control structures)
- Evaluation of the breadth of testing (“coverage”).

Black Box Software Testing Foundations
Lecture 5
The Impossibility of Complete Testing

Cem Kaner J.D., PH.D.

Professor Emeritus, Software Engineering, Florida Institute of Technology

Course Overview: Fundamental Topics

1. The Nature of Testing
   Overview and Basic Definitions

2. Why are we testing? What are we trying to learn? How should we organize our work to achieve this? ***Information objectives drive the testing mission and strategy***

3. How can we know whether a program has passed or failed a test?
   Oracles are heuristic

4. How can we determine how much testing has been done? What core knowledge about program internals do testers need to consider this question?
   Coverage is a multidimensional problem

5. Are we done yet?
   Complete testing is impossible ⬅

6. How much testing have we completed and how well have we done it?
   Measurement is important but hard

Today’s Readings

Required

- Doug Hoffman (2003). “Exhausting Your Test Options”, https://bbst.courses/wp-content/uploads/2022/08/Hoffman_Exhaust_Options.pdf
- Cem Kaner (1997), “The Impossibility of Complete Testing”, https://kaner.com/pdfs/imposs.pdf

Useful to skim

- Rex Black (2002), “Factors that Influence Test Estimation”, www.stickyminds.com/sitewide.asp?ObjectId=5992&Function=edetail&ObjectType=ART
- Cem Kaner (1996), “Negotiating Testing Resources: A Collaborative Approach.” https://kaner.com/pdfs/qweek1.pdf
- Mike Kelly, “Estimating testing using spreadsheets”, https://www.michaeldkelly.com/blog/2007/11/17/estimating-testing-using-spreadsheets.html

Coverage

- Last time, we considered some structural coverage measures and realized that
  - **complete coverage doesn’t mean complete testing.**

Question:

What do we have to do, to achieve complete testing?

Answer:

We can’t achieve complete testing. We might be able to achieve adequate testing…

Complete Testing

- Two tests are **distinct** if one test would expose a bug that the other test would miss.
- As we see it, for testing to be truly **complete**, you would have to:
  1. Run all distinct tests.
  2. Test so thoroughly that you know there are no bugs left in the software.
- It should be obvious (but it is not always obvious to every person) that the first and second criteria for complete testing are equivalent, and that testing that does not meet this criterion is incomplete.
- If this is not obvious to you, ask your instructor (or your colleagues) for help.

Incomplete Testing

We almost always stop testing before we know that there are no remaining bugs. At this point, testing might be "finished" (we ran out of time), but if there are still bugs to be found, how can testing be considered "complete"?

Complete Testing

To test everything, you would have to:

- Test every possible input to every variable (including output variables and intermediate results variables)
- Test every possible combination of inputs to every combination of variables
- Test every possible sequence through the program
- Test every possible timing of inputs (check for timeouts and races)
- Test every interrupt at every point it can occur
- Test every hardware/software configuration, including configurations of servers not under your control
- Test for interference with other programs operating at the same time
- Test every way in which any user might try to use the program

See Cem Kaner (1997), The Impossibility of Complete Testing, https://kaner.com/pdfs/imposs.pdf

Complete Testing

To test everything, you would have to:

- **Test every possible input to every variable (including output variables and intermediate results variables)** ←
- Test every possible combination of inputs to every combination of variables
- Test every possible sequence through the program
- Test every possible timing of inputs (check for timeouts and races)
- Test every interrupt at every point it can occur
- Test every hardware/software configuration, including configurations of servers not under your control
- Test for interference with other programs operating at the same time
- Test every way in which any user might try to use the program

Normally, we would sample the smallest and largest “valid” values (and the nearest “invalid” values). Or, if the values naturally subdivide into smaller groups, we’d sample one from each group (plus a few almost-valid values to check error handling).

Test Every Input

- All the “valid” inputs
    - How many valid inputs are there to a function that reads 32 bits from memory as an unsigned integer and takes the square root?
    - How many valid inputs to a function that reads 64 bits as an unsigned integer?
- Yes, of course we can sample. (We will often have to.)
    - But optimizations, some calculation errors, and other special-case handling can go undetected if we don’t check every possible input.

The MASPAR Example:

Testing the “Valid” Inputs

Doug Hoffman worked on the MASPAR (the Massively Parallel computer, 64K parallel processors). The MASPAR has several built-in mathematical functions. The Integer square root function takes a 32-bit word as an input, interpreting it as an integer (value is between 0 and 2^{32}-1). There are 4,294,967,296 possible inputs to this function.

How many should we test? What if you knew this machine was to be used for mission-critical and life-critical applications?

See Doug Hoffman (2003). “Exhausting your test options”
https://bbst.courses/wp-content/uploads/2022/08/Hoffman_Exhaust_Options.pdf

MASPAR

Common 32-bit test patterns include:

0	00000000000000000000000000000000
2^{32}-1	11111111111111111111111111111111
2^0	00000000000000000000000000000001
2^1	00000000000000000000000000000010
	…
2^{31}	10000000000000000000000000000000
(2^{32}-1)-1	11111111111111111111111111111110
(2^{32}-1)-2	11111111111111111111111111111101
	…
(2^{32}-1)-2^{31}	01111111111111111111111111111111

See Doug Hoffman (2003). “Exhausting your test options”
https://bbst.courses/wp-content/uploads/2022/08/Hoffman_Exhaust_Options.pdf

MASPAR

- To test the 32-bit integer square root function, Hoffman checked all values (all 4,294,967,296 of them). This took the computer about 6 minutes to run the tests and compare the results to an oracle.
- There were 2 (two) errors, neither of them near any boundary. (The underlying error was that a bit was sometimes missed, but in most error cases, there was no effect on the final calculated result.) Without an exhaustive test, these errors probably wouldn’t have shown up.
- What about the 64-bit integer square root? How could we find the time to run all of these? If we don't run them all, don't we risk missing some bugs?

- **To test all combinations of 32 bits, there are 2^{32} tests**
- **These 2^{32} tests required 6 minutes of testing.**
- **To test all combinations of 64 bits, there are 2^{64} tests: $2^{64} = 2^{32} \times 2^{32}$**
- **For this, we’d need 2^{32} x 6 minutes, i.e. (24 billion) minutes.**
- **This is clearly impossible, so we MUST sample, even though this might cause us to miss some bugs.**

Testing Every Input

Along with the simple cases, there are other “valid” inputs

- Edited inputs
    - The editing of an input can be quite complex. How much testing of editing is enough to convince us that no additional editing would trigger a new failure?
- Variations on input timing
    - Try entering data very quickly, or very slowly. Enter data before, during and after the processing of some other event, or just as the time-out interval for this data item is about to expire.
    - In a client-server world (or any situation that involves multiple processors) consideration of input timing is essential.

Invalid Inputs to Individual Variables

Normally, we look for boundaries, values at the edge of validity (almost invalid, or almost valid):

- If an input field accepts 1 to 100, we test with -1 and 0 and 101.
- If a program will multiply two numbers together using integer arithmetic, we try inputs that, together, will drive the multiplication just barely above MaxInt, to force an overflow.
- If a program can display a 9-character output field, we look for inputs that will force the output to be 10 characters.

Invalid Inputs to Individual Variables

However, there are additional possibilities. For example:

- Extreme values can cause overflows or underflows.
  - An enormous input string might overflow the area reserved for input, overwriting other data.
  - An empty input string might cause a there's-no-input failure such as a null pointer exception.
- These types of errors do happen accidently, but buffer overflows are also the most commonly exploited vulnerability by malicious code (or coders).

And there are OTHER possibilities, like Easter Eggs.

Extreme Values Expose Error-Handling Weaknesses

"No user would do that" really means "No user I can think of, who I like, would do that on purpose".

- Who aren't you thinking of?
- Who don't you like who might really use this product?
- What might good users do by accident?

Obviously, we can't test every possible invalid value (there are infinitely many). We have to sample...

Complete Testing

To test everything, you would have to:

- Test every possible input to every variable (including output variables and intermediate results variables)
- **Test every possible combination of inputs to every combination of variables** ←
- Test every possible sequence through the program
- Test every possible timing of inputs (check for timeouts and races)
- Test every interrupt at every point it can occur
- Test every hardware/software configuration, including configurations of servers not under your control
- Test for interference with other programs operating at the same time
- Test every way in which any user might try to use the program

Even if we ignore invalid variable values, the number of input combinations we can test gets impossibly large quickly.
Several techniques are available to guide our sampling of these inputs.

An Example

- Program printed user-designed calendars
  - Printing to high-resolution printers worked well
  - Displaying to a high-resolution monitor worked well
  - “Print preview” of a high-resolution printout to a high-resolution monitor crashed Windows
- The variables here are configuration variables: what printer, what video card, how much memory, etc.

The program worked well with each variable, when we tested them one at a time. But when we tested them together, the system crashed.

The Basic Combination Rule

Suppose there are K independent variables, V1, V2, ..., VK.

Label the number of choices for the variables as

N1, N2 through NK.

The total number of possible combinations is

N1 x N2 x . . . x NK.

The Basic Combination Rule

Apply the basic rule to our configuration example

- V1 is the type of printer (we're ignoring printer driver versions). N1 is the number of printers we want to test. (40 has been realistic on many projects. We've worked on projects with over 500.)
- V2 is the type of video card. N2 is the number of types of video cards we want to test. (20 or more is realistic.)
- Number of distinct tests = N1 x N2.

Number of printers	Number of video cards	Number of tests
2	2	4
3	3	9
5	5	25
40	20	800

Suppose we add a third variable (V3): how much free memory is available In the computer. Now we have N1 x N2 x N3 tests

The Basic Combination Rule

Suppose we test

- N1 printers, with
- N2 versions of their printer driver
- N3 video cards, with
- N4 versions of their driver
- N5 amount of free memory
- N6 versions of the operating system
- N7 audio drivers
- N8 mouse drivers
- N9 keyboard drivers
- N10 types of connections to the Net

= N1 x N2 x N3 x N4 x N5 x N6 x N7 x N8 x N9 x N10 distinct tests

It’s Not Just Configuration Testing

- Booked a several-segment (several country) trip on American Airlines on a special deal that yielded a relatively low first-class fare.
- AA prints a string on the ticket that lists all segments and their fares.
- Ticket agents at a busy airport couldn’t print the ticket because the string was too long. The usual easy workaround was to split up the trip (issue a few tickets) but in this combination of flights, splitting caused a huge fare change.
- It took nearly an hour of agent time to figure out a ticketing combination that worked.

- **How many variables are in play here?**
- **How many combinations would you have to test to discover this problem and determine whether it happens often enough to be considered serious?**

Combinations

- The *normal* case when testing combinations of several independent variables is to adopt a sampling scheme. (After all, we can't run **all** these tests.)
- For example, with 40 printers and 20 video cards, you might cut back to 50 tests:
    - One test for every printer (40 tests).
    - Test each video card at least once (test printer and video together, you still have only 40 tests).
    - Add a few more tests to check specific combinations that have caused technical support problems for other products.
- Variants on this sampling scheme are common. Some (the combinatorial tests, such as "all-pairs") are widely discussed.
    - In our example of 40 printers x 20 video cards x 2 levels of memory, all-pairs would reduce the 1600 tests to a sample of 800.

As with all other tests, though, any combination you don't test is a combination that might trigger a failure.

Complete Testing

To test everything, you would have to

- Test every possible input to every variable (including output variables and intermediate results variables)
- Test every possible combination of inputs to every combination of variables
- **Test every possible sequence through the program** ←
- Test every possible timing of inputs (check for timeouts and races)
- Test every interrupt at every point it can occur
- Test every hardware/software configuration, including configurations of servers not under your control
- Test for interference with other programs operating at the same time
- Test every way in which any user might try to use the program

Paths and Subpaths

A path through a program

- starts at an entry point (you start the program)
- ends at an exit point (the program stops)

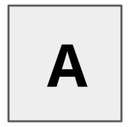
A sub-path

- starts and ends anywhere

A sub-path of length N

- starts, continues through N statements, and then stops

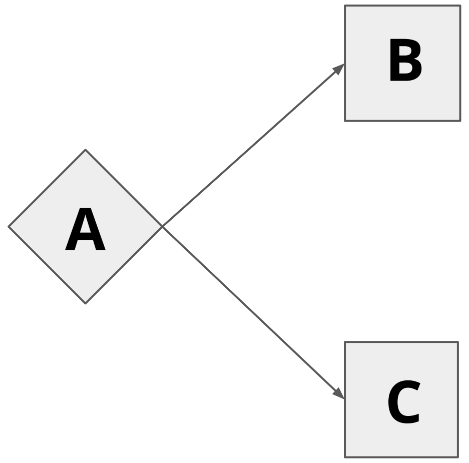
Some Notation



Do task “A”.

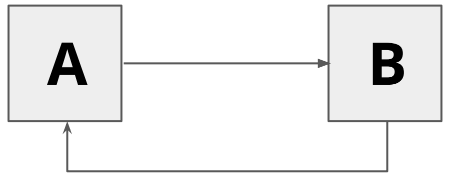
“A” might be a single statement

or a block of code or an observable event.



Do task “A” and then do “B” or “C”.

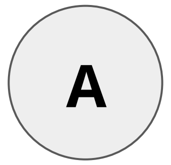
This is a basic branch.



Do task “A” and then do “B”

and then loop back to A.

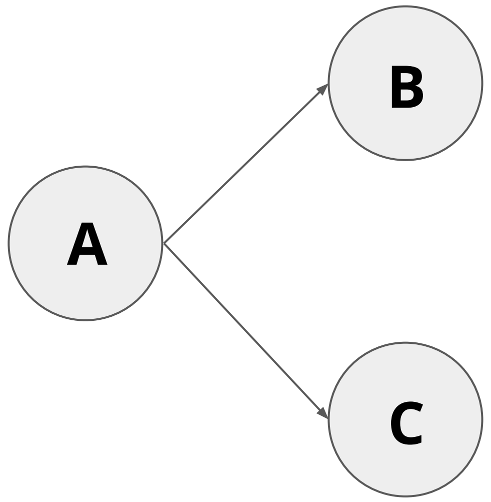
Some Notation



Do task “A”.

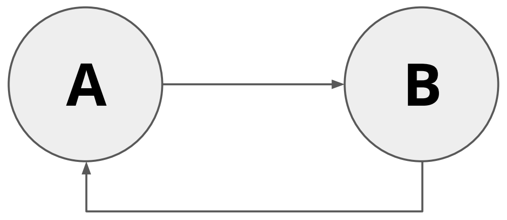
“A” might be a single statement

or a block of code or an observable event.



Do task “A” and then do “B” or “C”.

This is a basic branch.



Do task “A” and then do “B”

and then loop back to A.

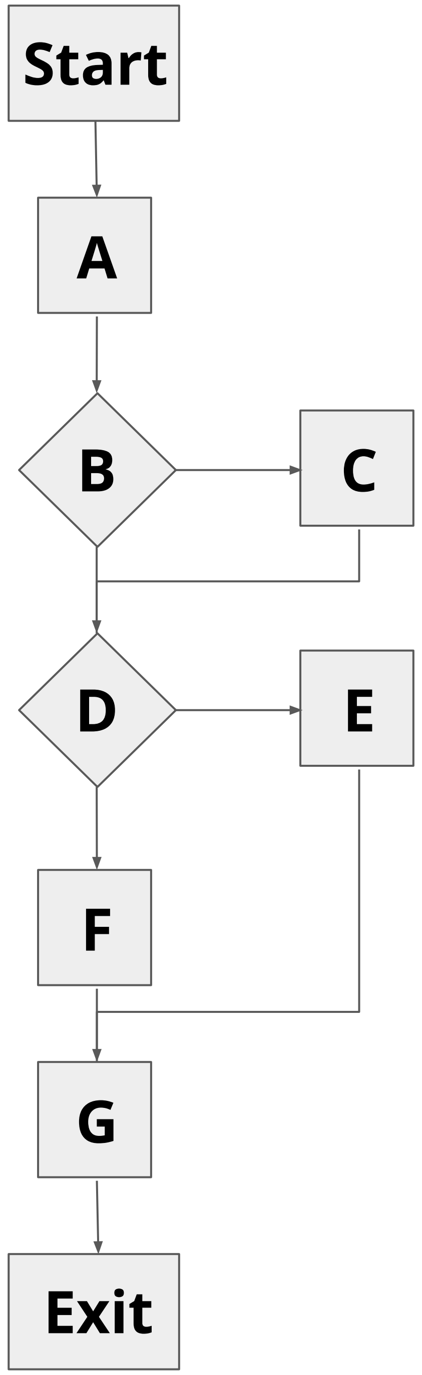
Vocabulary Alert

If we replace the boxes and diamonds with circles and call them “nodes” and call the lines “edges,” we have a “directed graph.” Directed graphs have useful mathematical properties for creating test-supporting models (e.g. state models).

For a detailed introduction written for testers, read Paul Jorgensen’s (2008) *Software Testing: A Craftsman’s Approach* (3rd Ed.)

Let's Analyze a Graph

This is based on an example from Richard Bender.

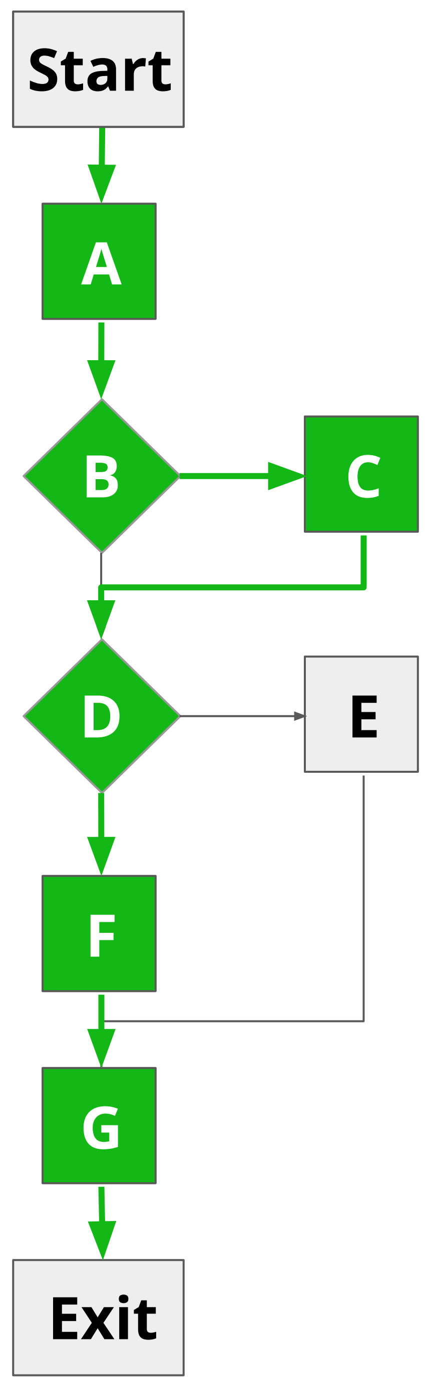


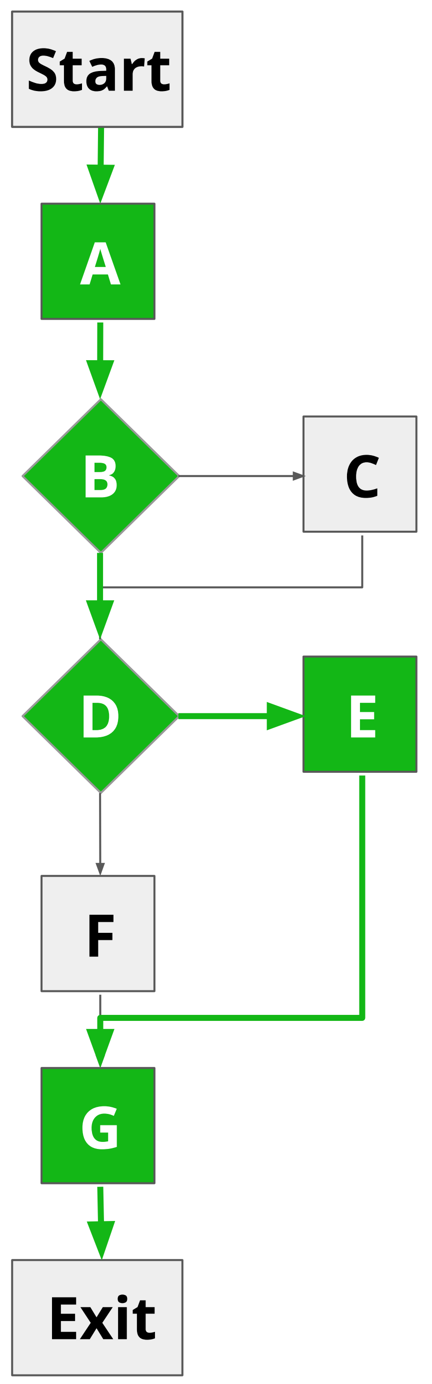
Let’s Analyze a Graph

We can achieve 100% branch coverage (all statements, all branches) by testing two paths:

- A, B, C, D, F, G
- A, B, D, E, G

Are we missing anything?





Data Flows

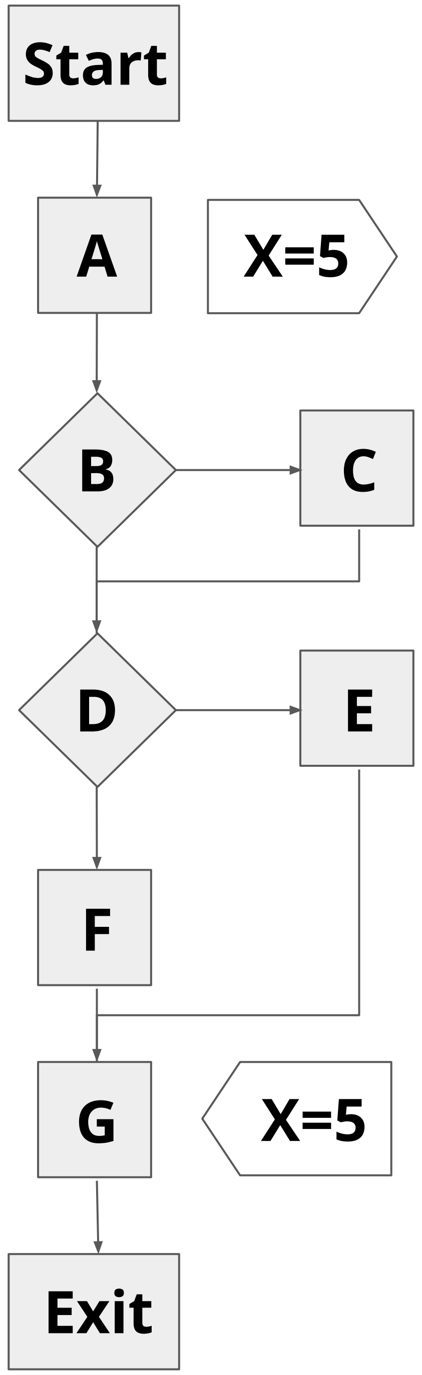
A **data flow** captures two events.

1. We set a variable to a value
2. We use the value

We call this a “Set-Use Pair”

X=5 This means “Set X = 5”

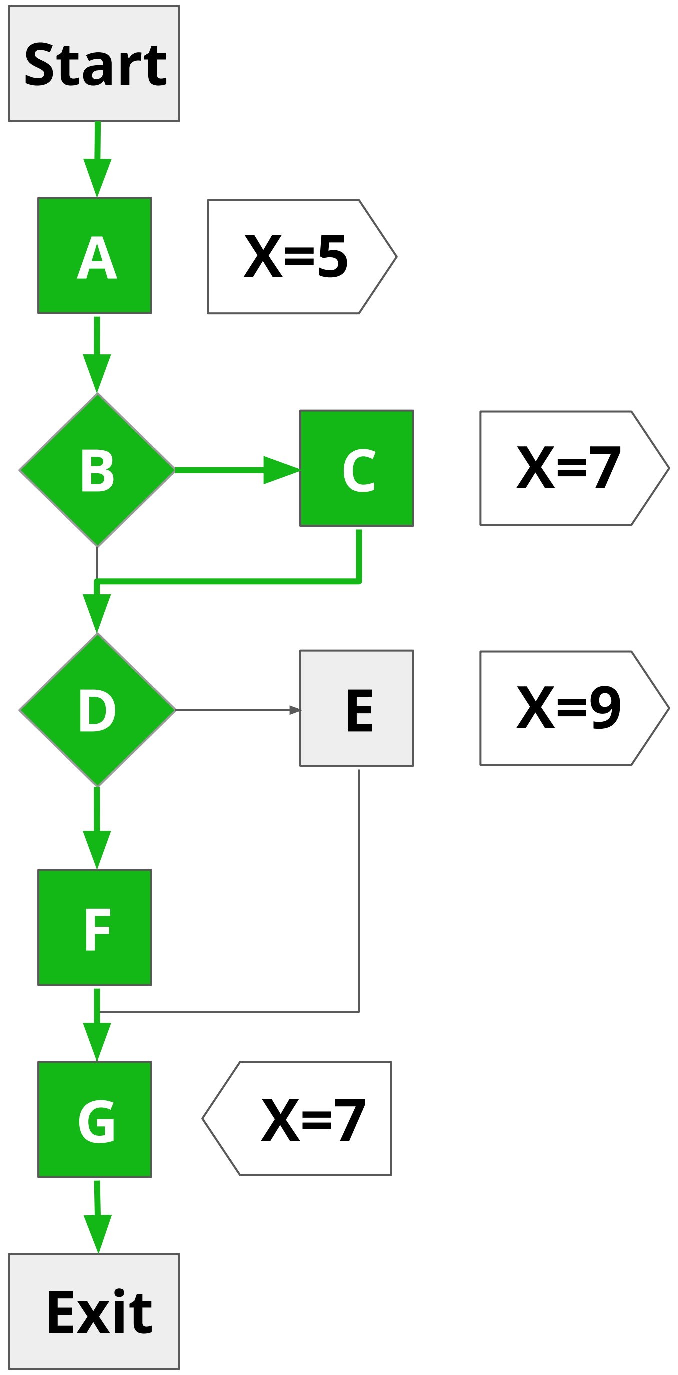
X=5 This means the program will read/use the value of X (which is 5)



Data Flows

Consider the possible data flows. We'll SET the value of X at various points and then print X at G.

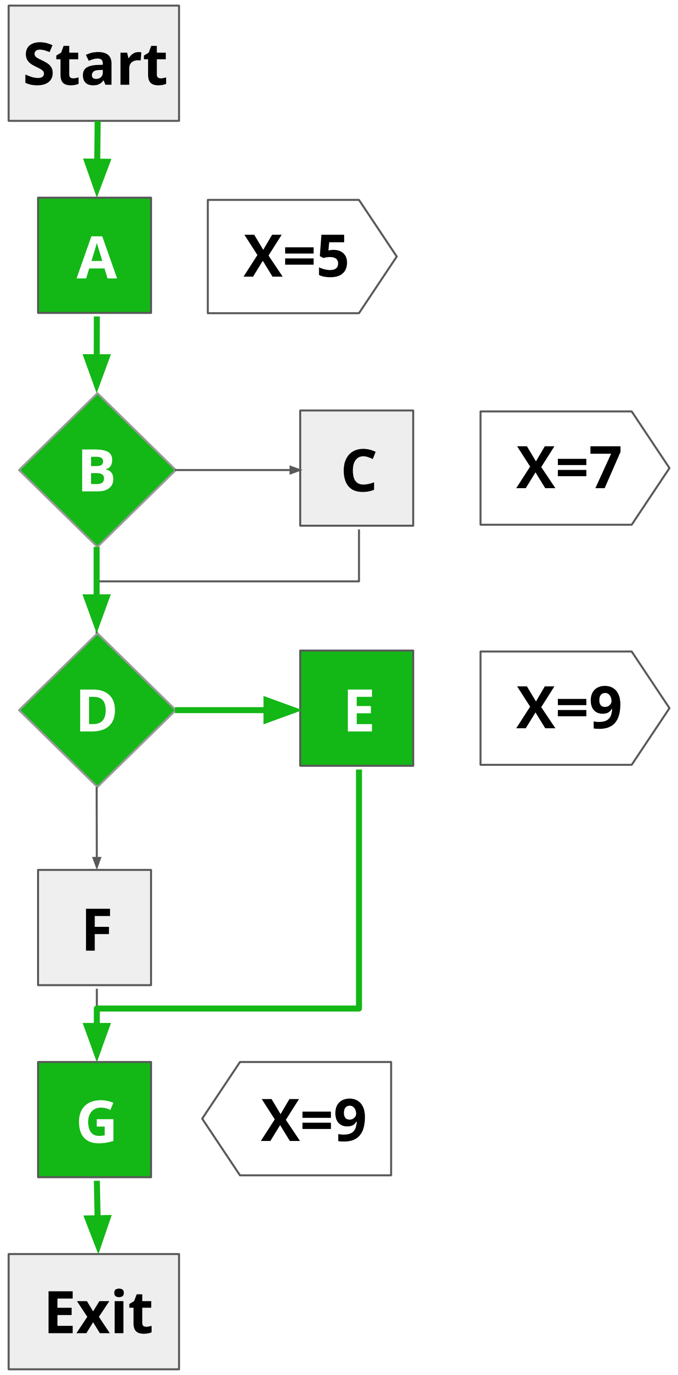
Here's our first of the two "complete-coverage" tests. The last place where X is set is at C, so at G, X=7.



Data Flows

Here’s our second of the complete-coverage tests:

- This time, the last place to set a value for X is E, so G gets a 9.



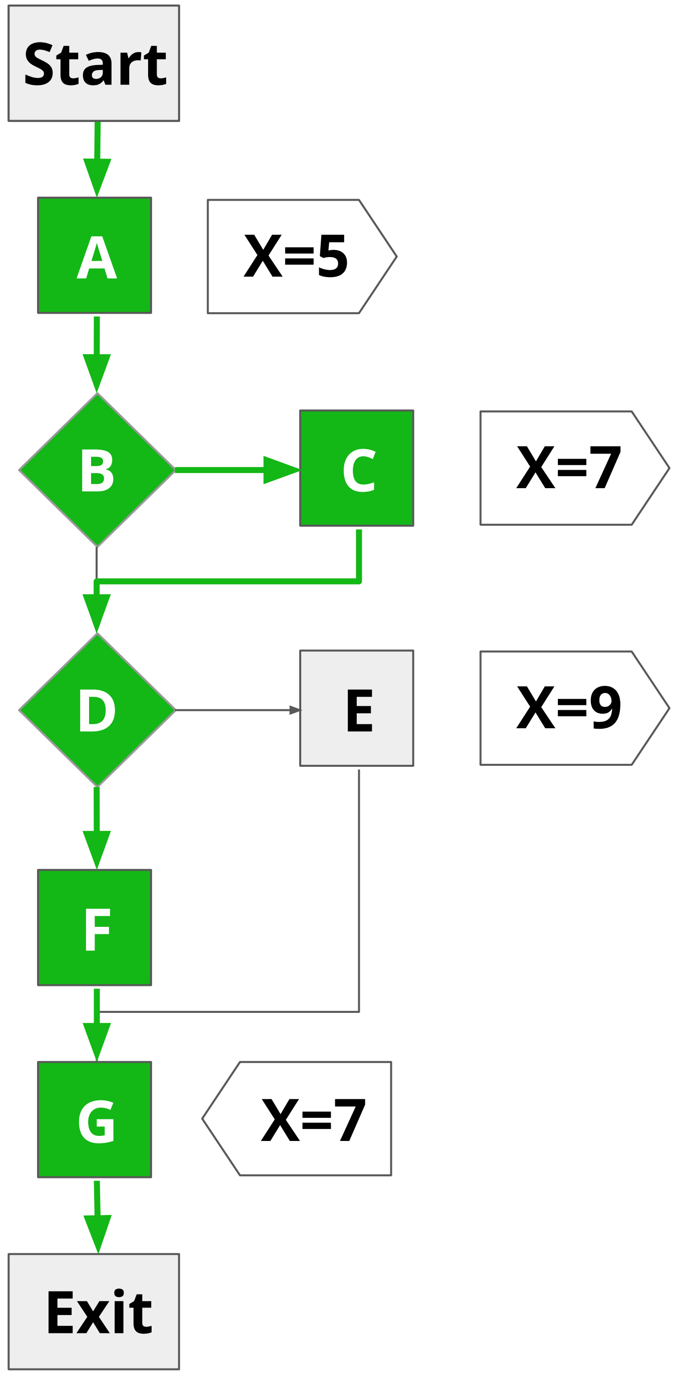
Data Flows

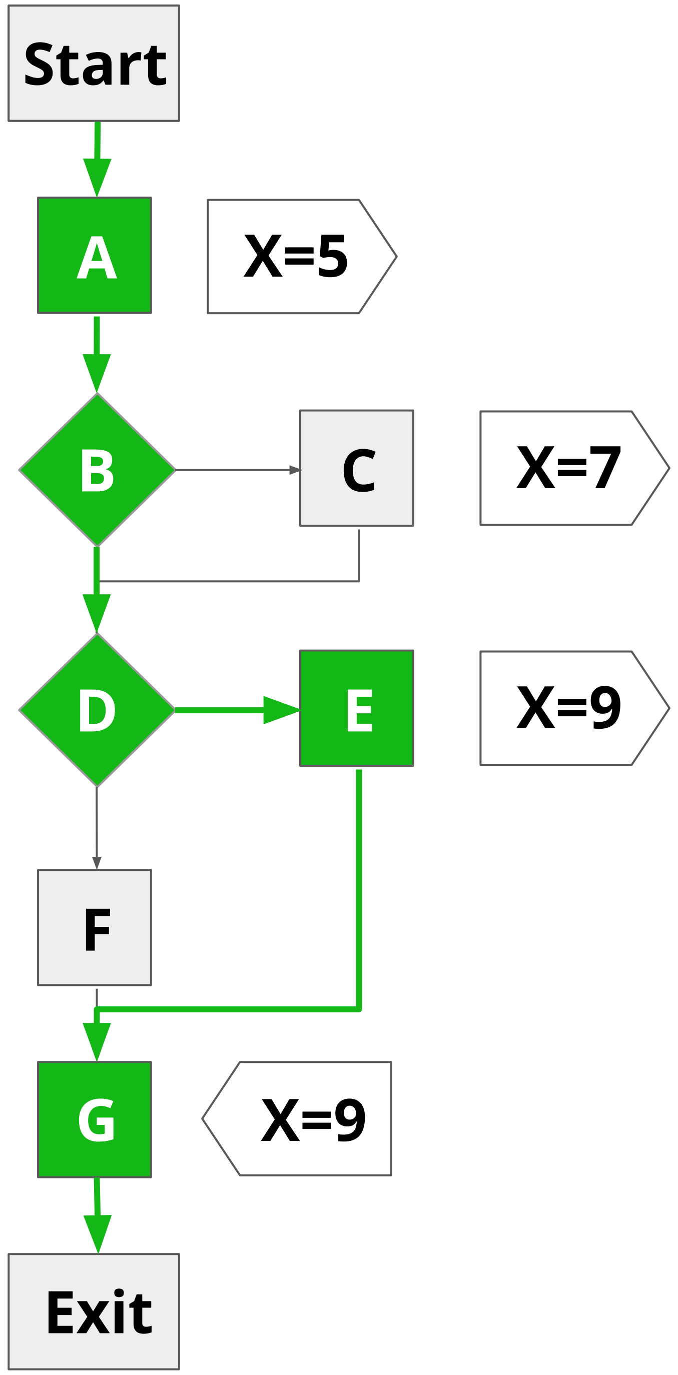
We've now tested all statements and all branches.

In doing so, we've tested the Set-Use pair (C, G)

and the Set-Use pair (E, G).

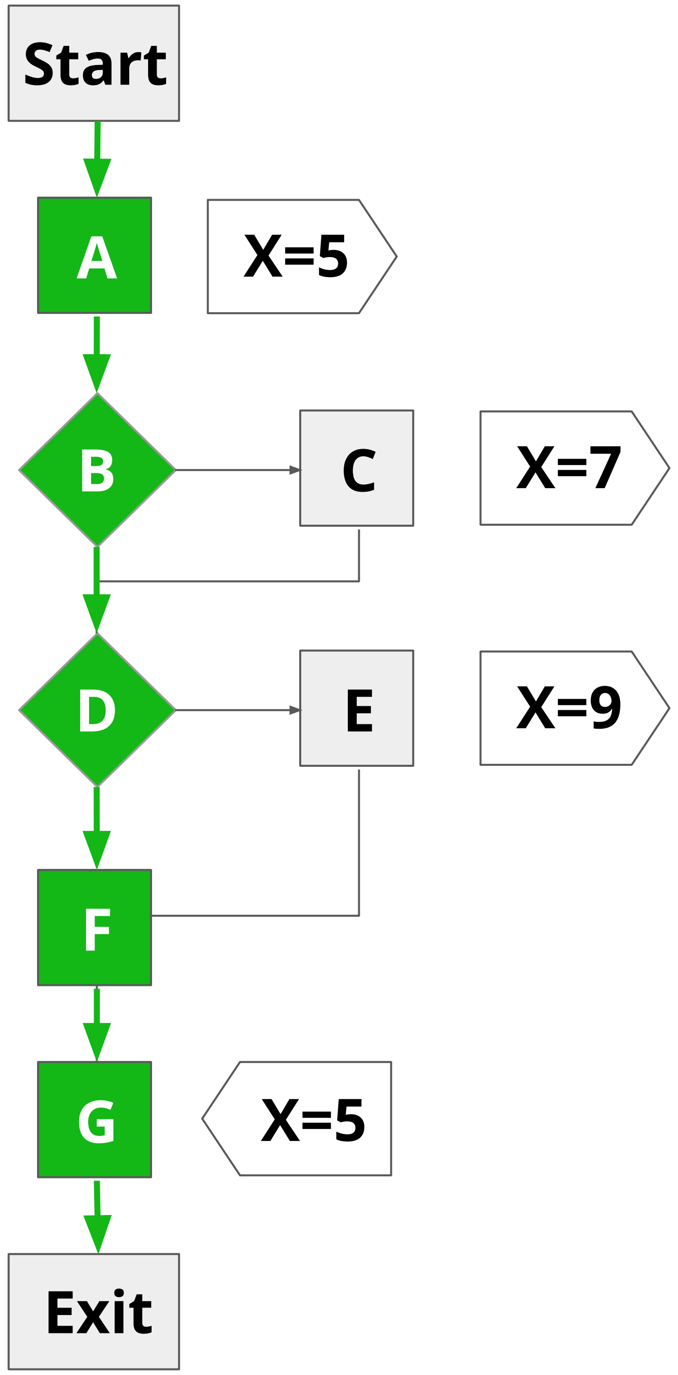
But where is the data flow from A to G?





Data Flows

To test that third data flow, we need to test a third path. This one will do it.



Data Flows: Caution

When you test data flows, it's not enough to set X and use it.

You must consider how X is used:

- What does the program do with X?
- What values of X might be troublesome for that use?
- Does the program use X in combination with other variables?
- What values of X would be troublesome with those variables?
- Does the program based another variable on X or on a calculation **that** uses X? What trouble can that variable cause?

Test the **consequences** of the use.

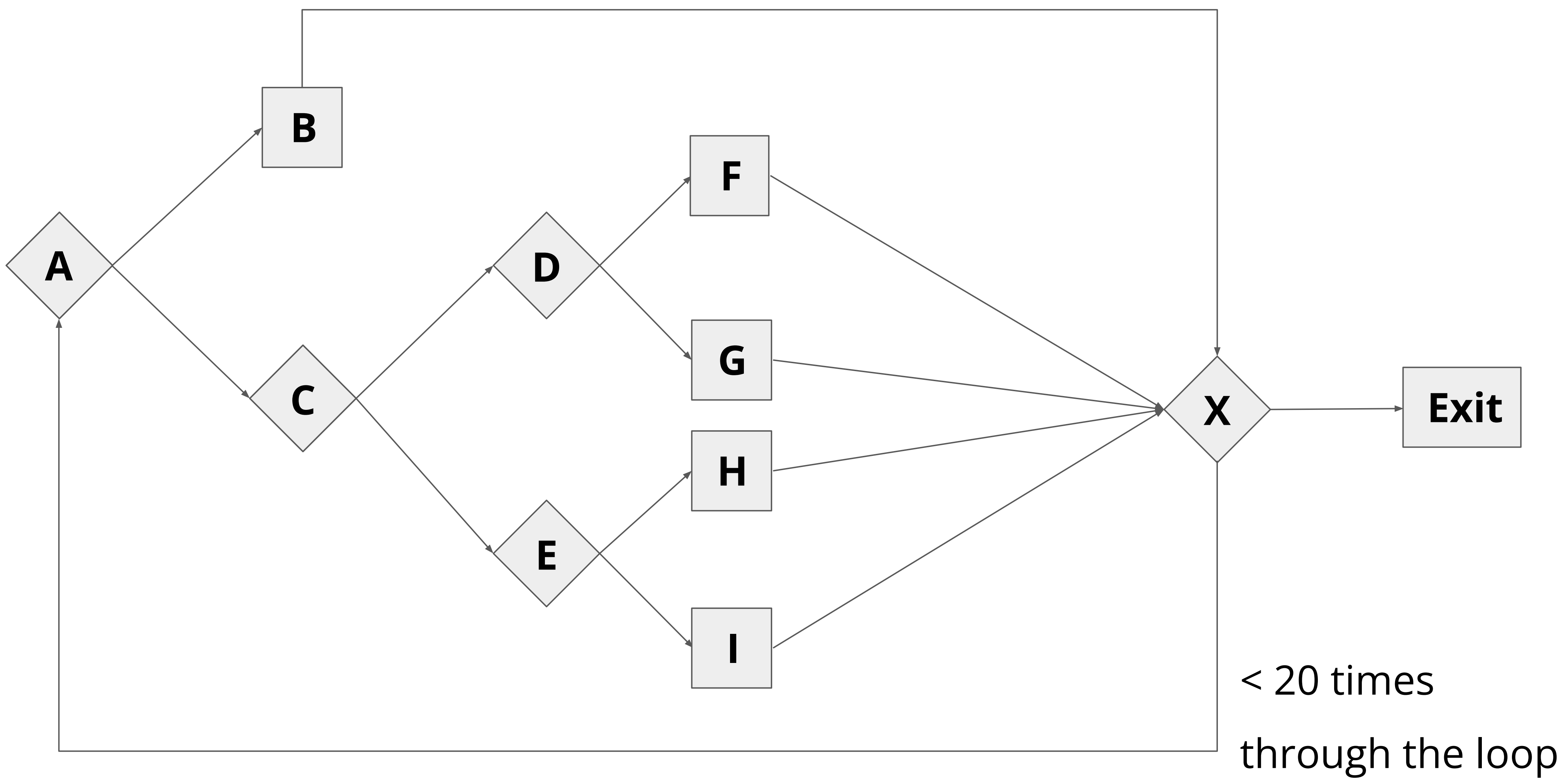
Complete Testing

To test everything, you would have to

- Test every possible input to every variable (including output variables and intermediate results variables)
- Test every possible combination of inputs to every combination of variables
- **Test every possible sequence through the program** ←
- Test every possible timing of inputs (check for timeouts and races)
- Test every interrupt at every point it can occur
- Test every hardware/software configuration, including configurations of servers not under your control
- Test for interference with other programs operating at the same time
- Test every way in which any user might try to use the program

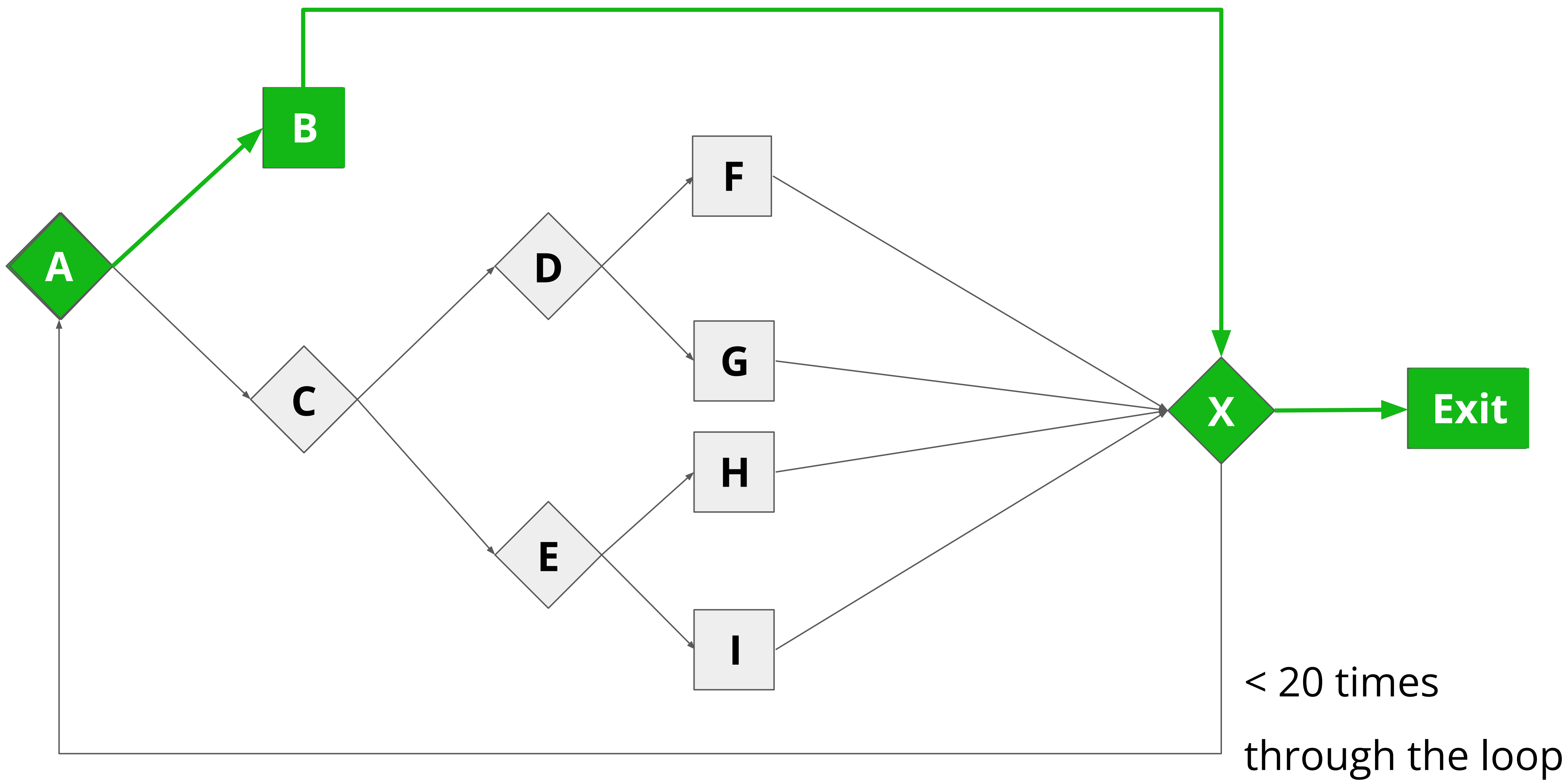
We're still in the middle of this every-possible-sequence analysis.

Next Example



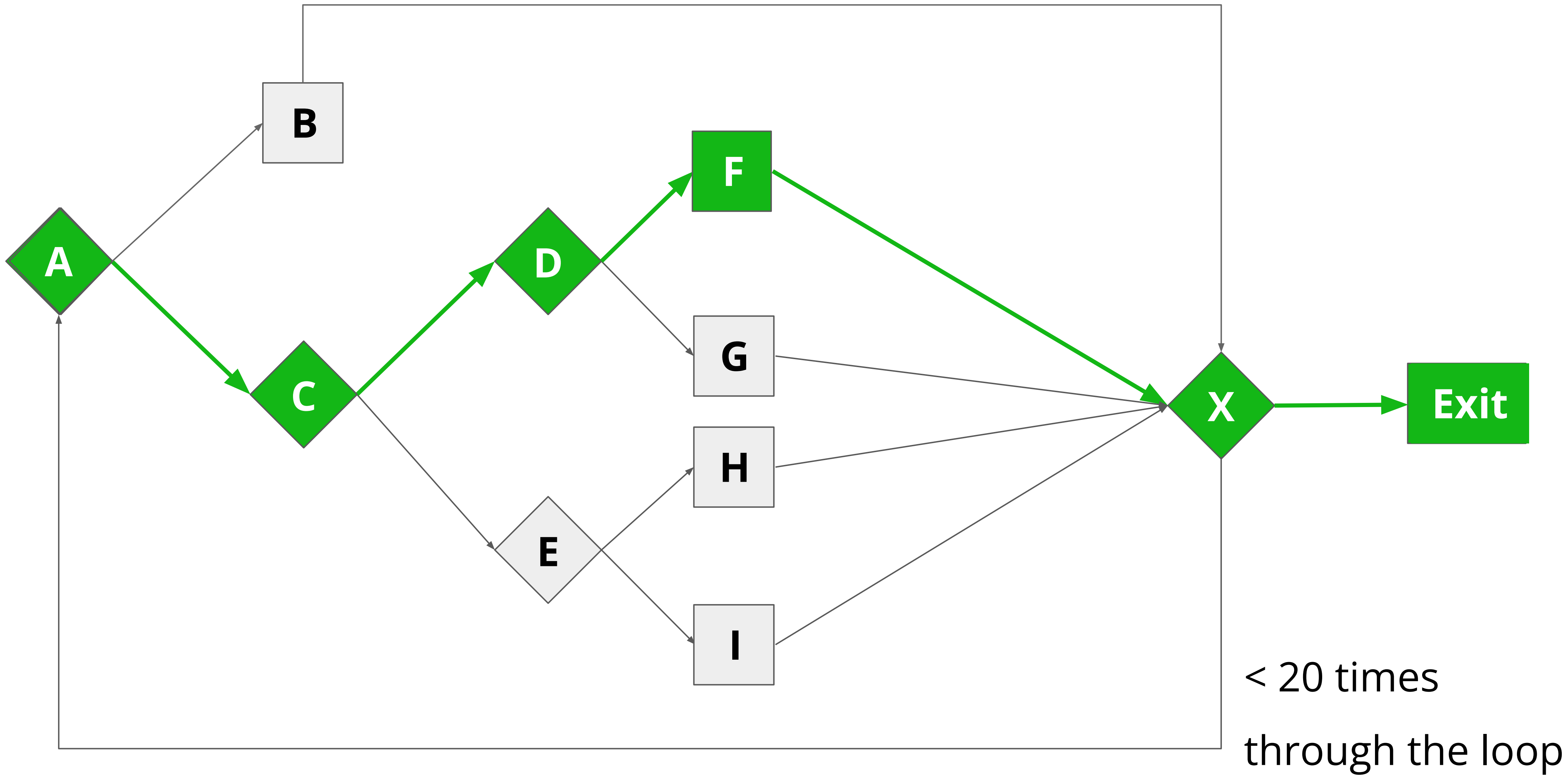
This example shows there are too many paths to test in even a fairly simple program. This is from Myers, ***The Art of Software Testing***.

Sequences



There are 5 ways to get from A to X. One of them is **A→B→X→EXIT**.

Sequences

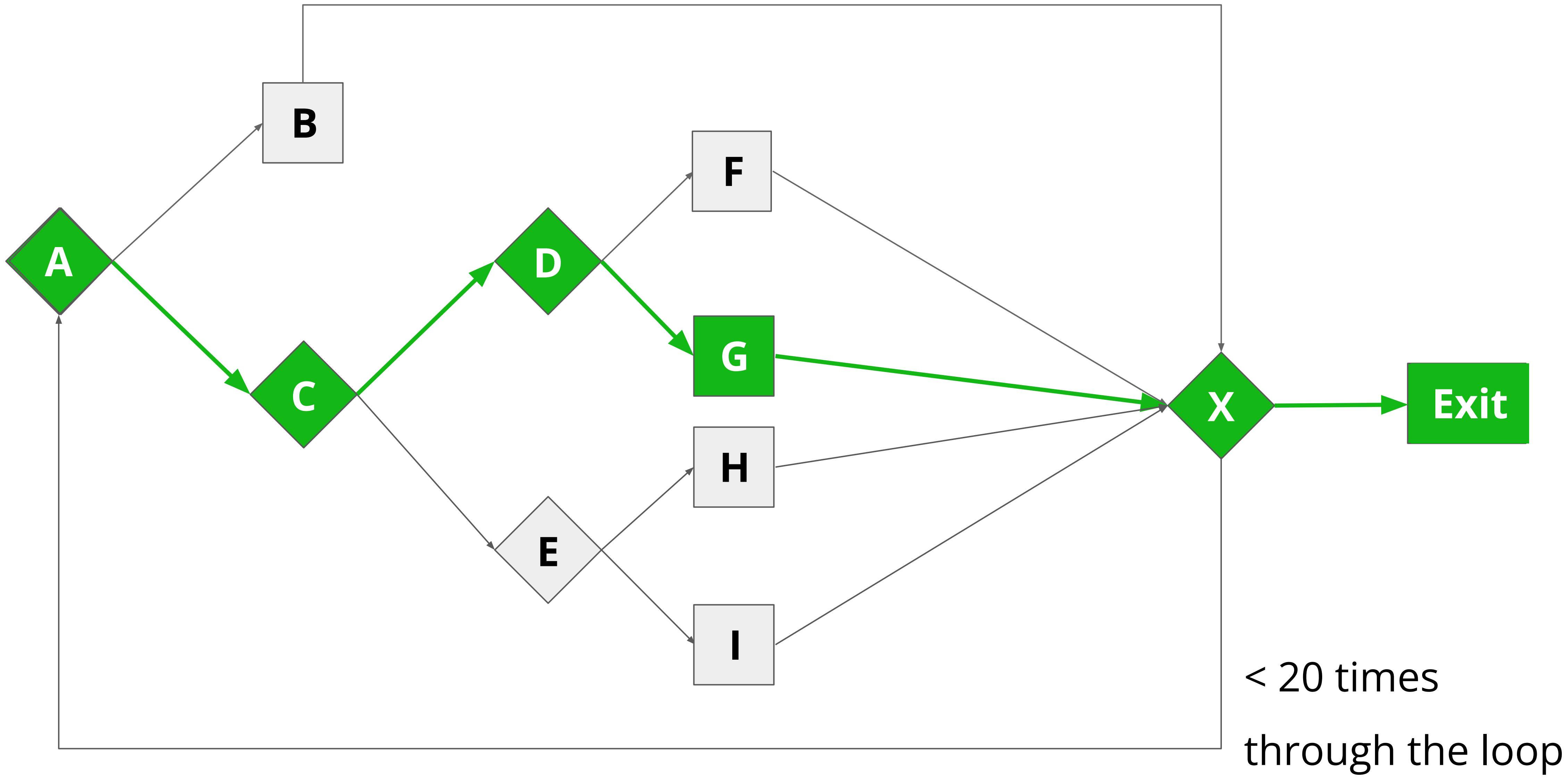


< 20 times

through the loop

A second path is **A→C→D→F→X→EXIT**.

Sequences

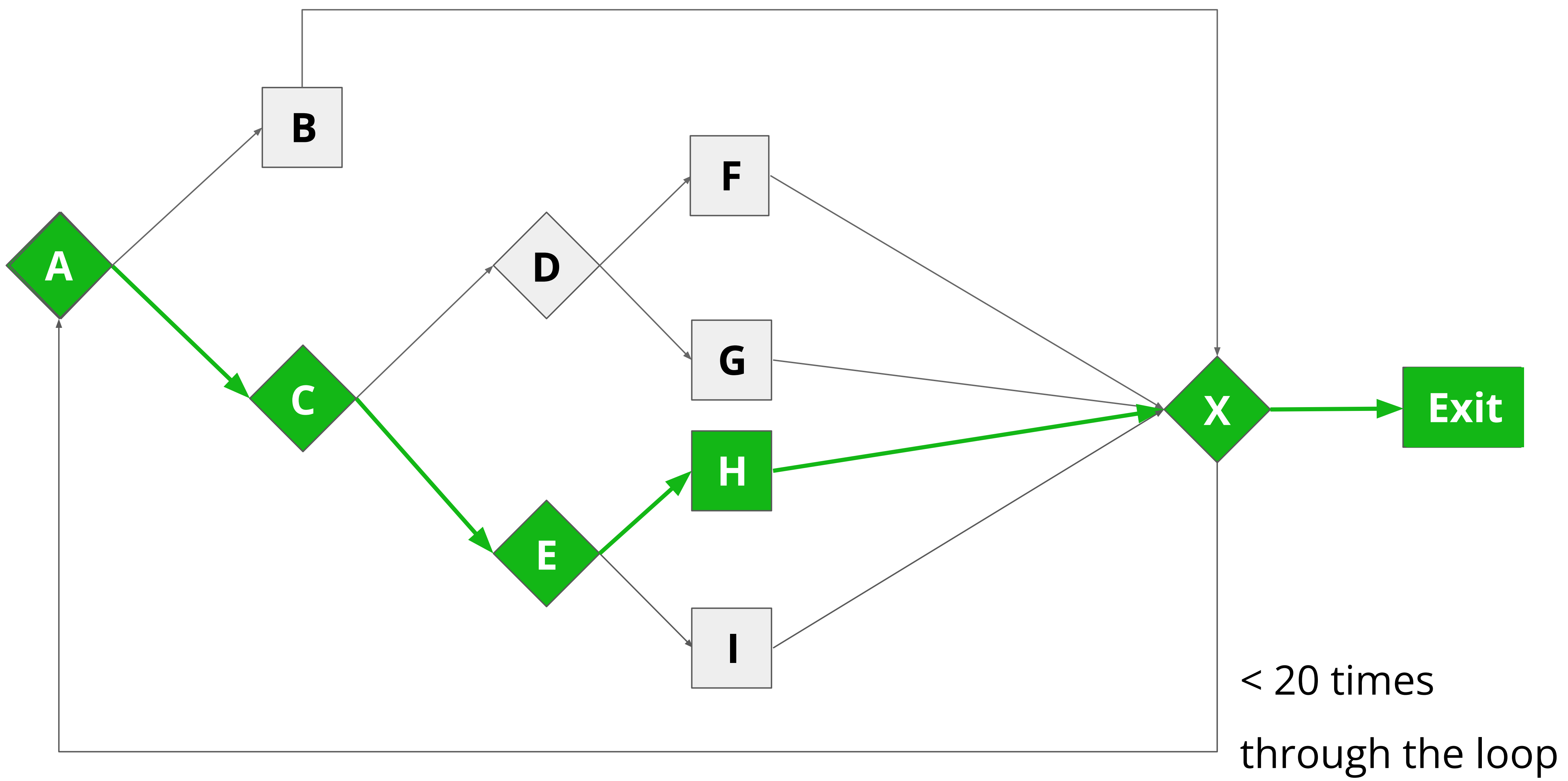


< 20 times

through the loop

Third: **A→C→D→G→X→EXIT**.

Sequences

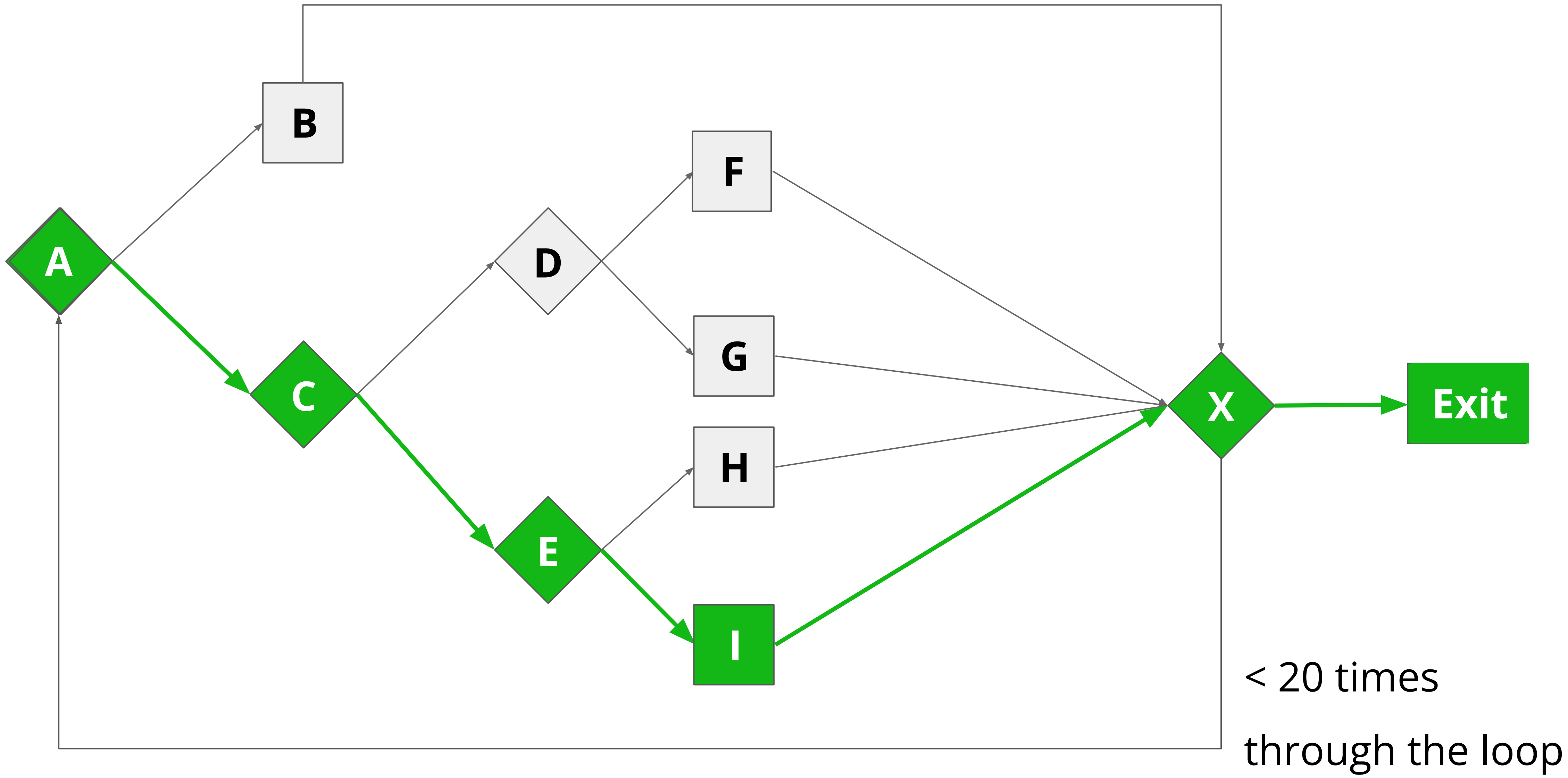


< 20 times

through the loop

Fourth: **A→C→E→H→X→EXIT**.

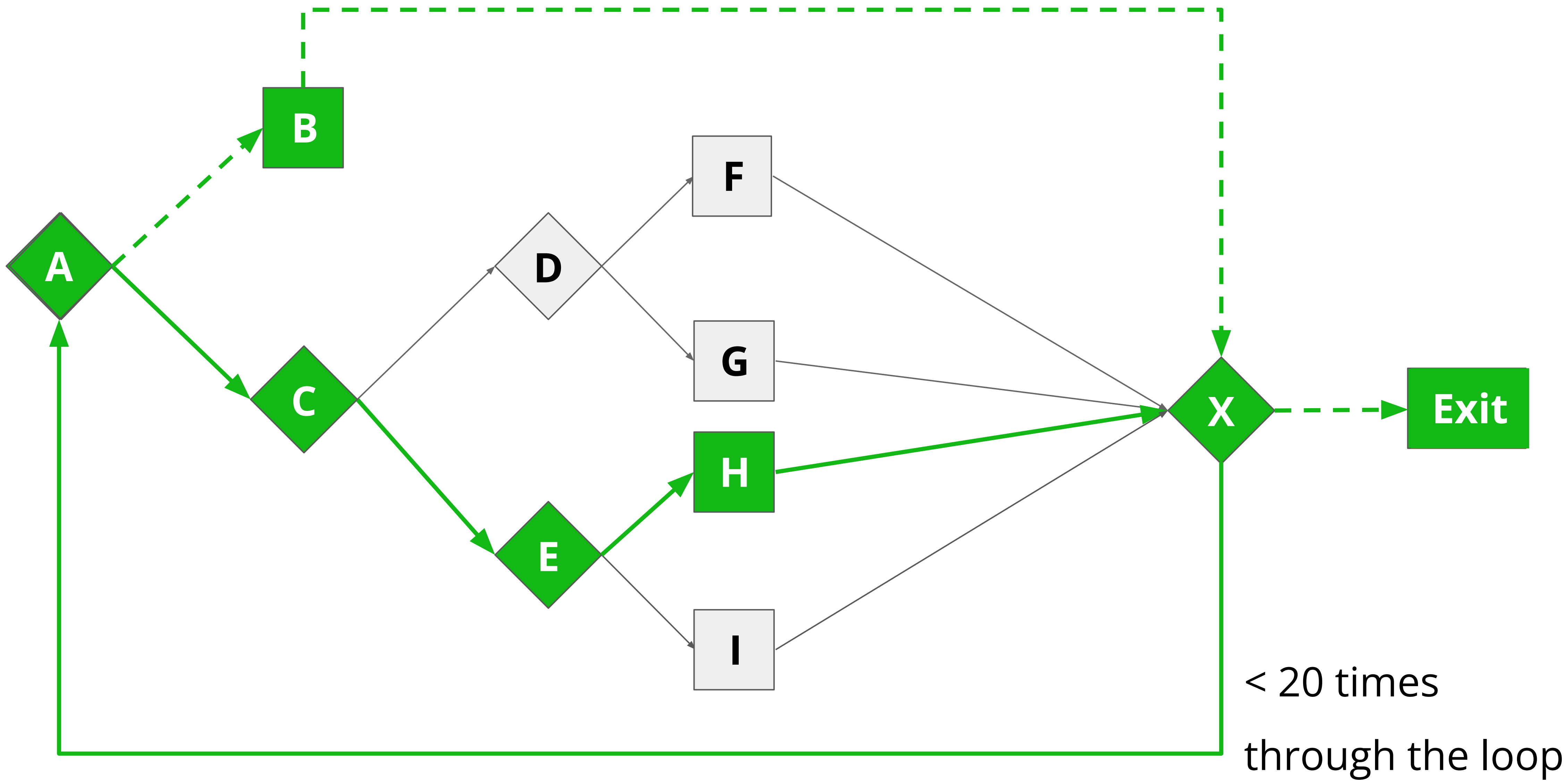
Sequences



Fifth: **A→C→E→I→X→EXIT**.

There are 5 ways to get from A to EXIT if you go through X only once.

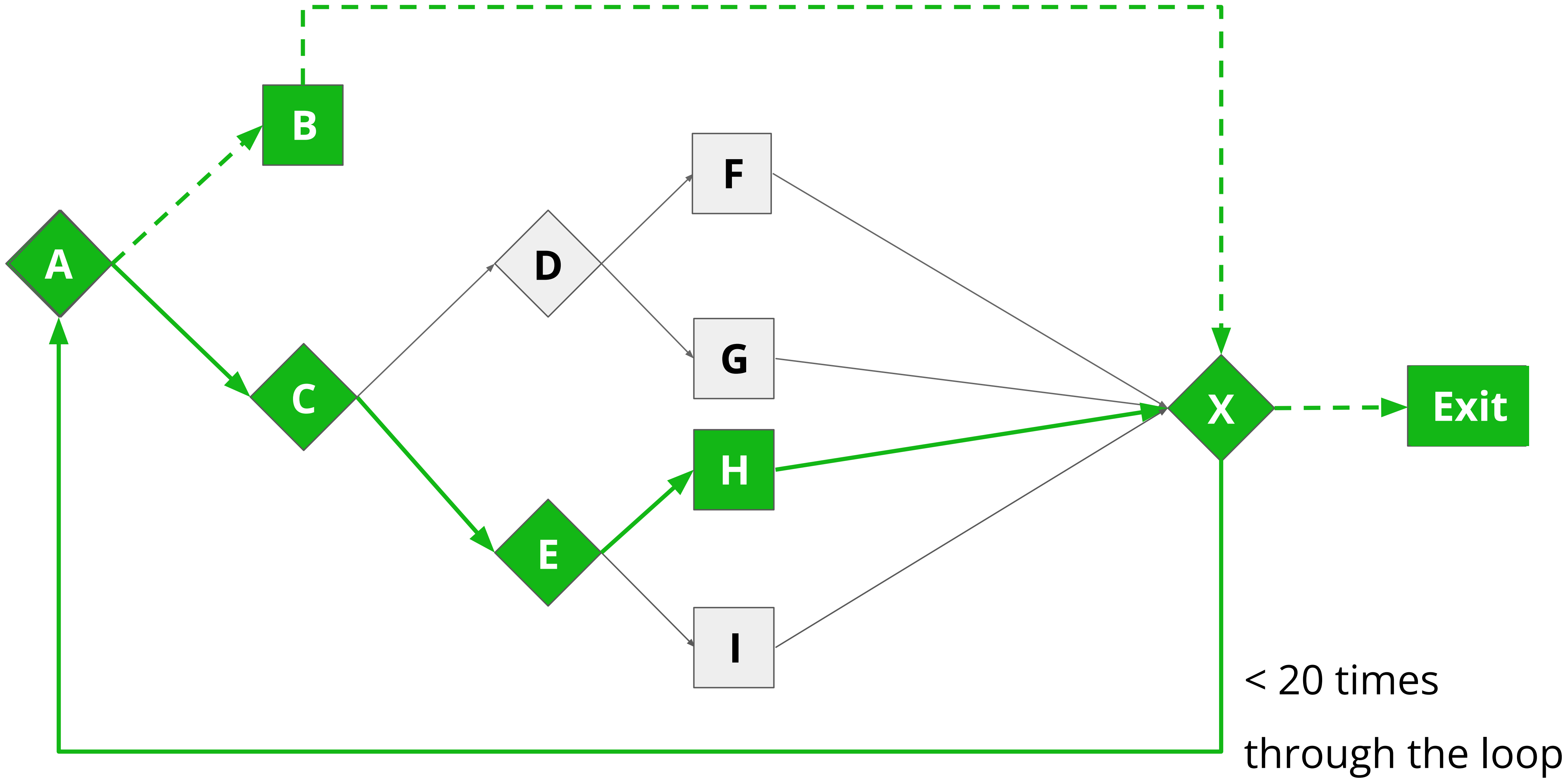
Sequences



But you can go through X more than once.

Here's another path: **A→C→E→H→X→A→B→X→EXIT**.

Sequences



< 20 times

through the loop

There are 5 ways to get to X the first time, 5 more to get back to X the second time, so there are 5 x 5 = 25 cases for reaching EXIT by passing through X twice.

Sequences

Analyzing Myers' Example

There are

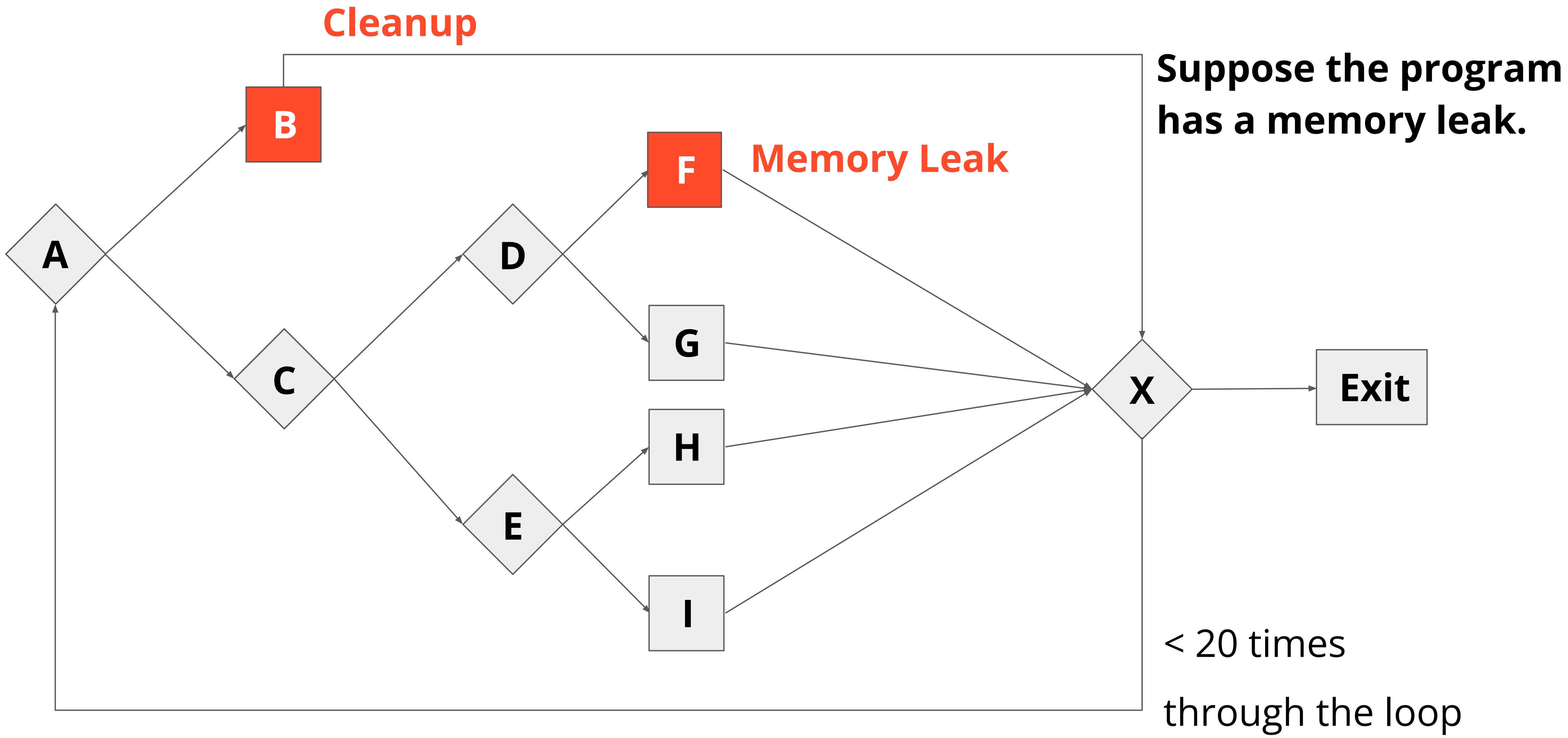
- 5 ways to get to X once, then EXIT
- 5 x 5 ways to get to X twice, then EXIT
- 5 x 5 x 5 ways to get to X three times, then EXIT

In total, there are $5^1 + 5^2 + \ldots + 5^{19} + 5^{20}$ = (approximately) 10^{14} = 100 trillion paths through the program. (This applies the combination formula we looked at before. With variables V1, V2, etc., the number of combination tests is N1 x N2 x ... etc.)

Obviously, we can't test all these paths, so we need to select a sample. A typical sample would probably include all 5 tests that get to EXIT once, at least one test that goes to EXIT all 20 times, an attempt to hit EXIT 21 times, and test that check the pairs

- **(pass through B, pass through F)**
- **(pass through G, pass through H and so on.)**

Sequences



Suppose the program has a memory leak.

Each time it hits F, it ties up more memory. If you tie up enough memory, the system crashes. Every time the program hits B, it cleans up memory, so the crash happens only if the program hits F several times without a B in its sequence.

Sequences

Analyzing Myers' Example

This example is often treated as "academic" (purely theoretical), but this kind of problem can definitely show up in real life.

Here's a bug I ran into as a programmer,
developing a phone system...

Phone System: The Telenova Stack Failure

Telenova Station Set 1. Integrated voice and data. 108 voice features, 110 data features. 1984.

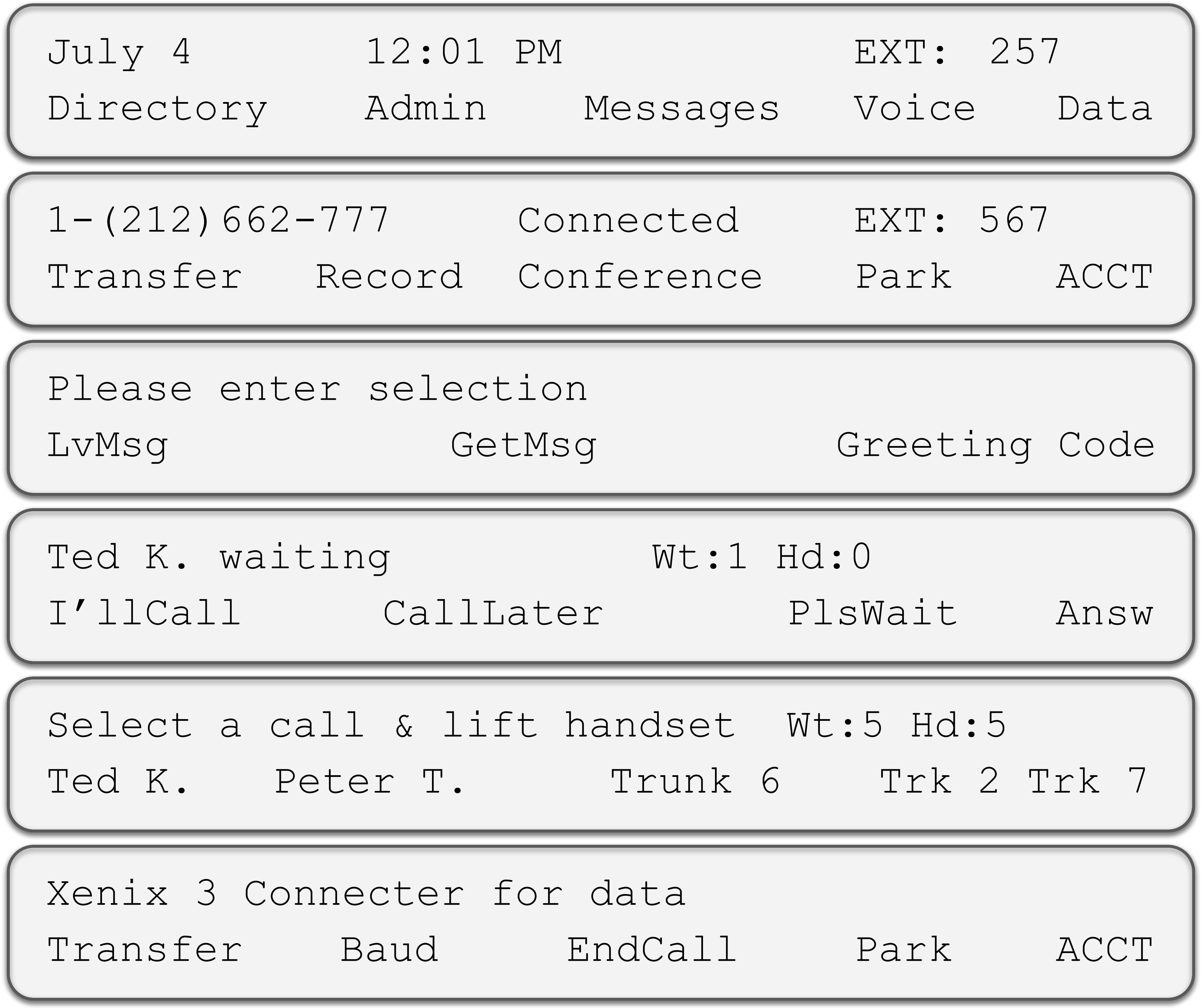
Telenova Station Set Desktop Telephone, 1982
Cooper Hewitt, Smithsonian Design Museum
https://collection.cooperhewitt.org/objects/18648527

The Telenova Stack Failure

Context-sensitive display

10-deep hold queue

10-deep wait queue



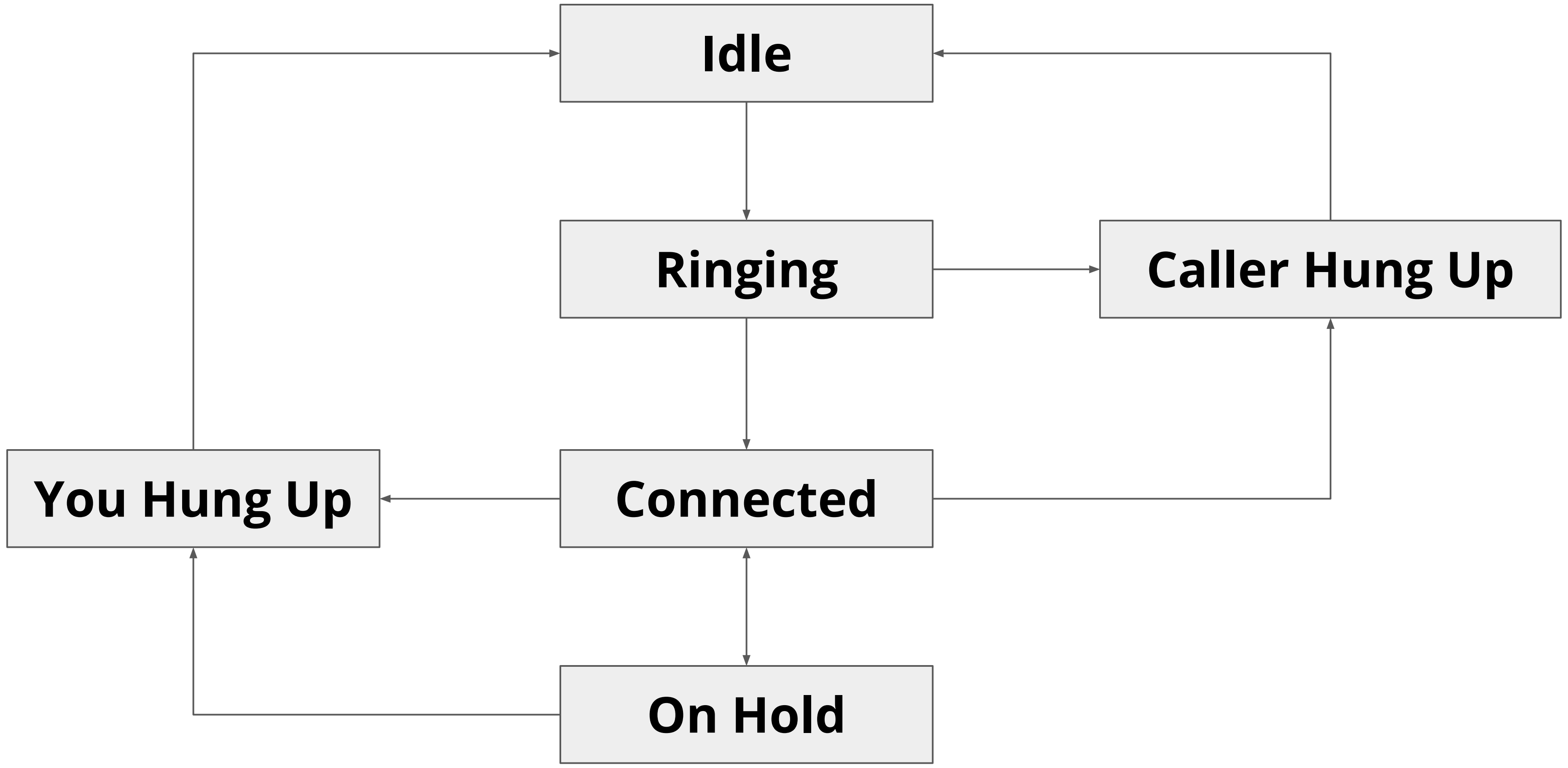
```
July 4          12:01 PM            EXT:  257
Directory       Admin     Messages  Voice     Data

1-(212)662-777       Connected      EXT: 567
Transfer     Record  Conference     Park      ACCT

Please enter selection
LvMsg               GetMsg              Greeting Code

Ted K. waiting                Wt:1 Hd:0
I'llCall      CallLater           PlsWait     Answ

Select a call & lift handset  Wt:5 Hd:5
Ted K.    Peter T.      Trunk 6     Trk 2 Trk 7

Xenix 3 Connecter for data
Transfer     Baud      EndCall      Park      ACCT
```

The Telenova Stack Failure

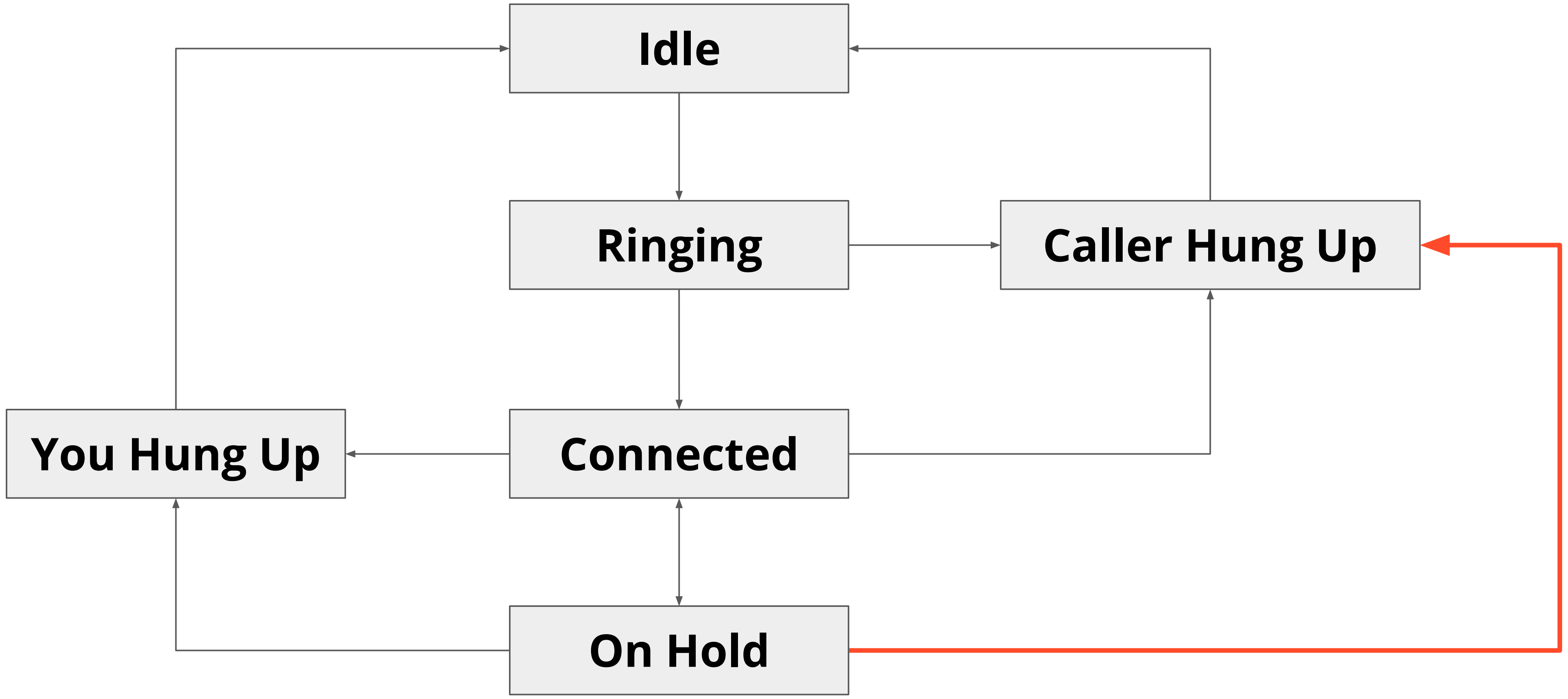
- Beta customer (a stock broker) reported random failures
- Could be frequent at peak times
- One phone would crash and reboot, then other phones crashed while the first rebooted
- On a particularly busy day, service was disrupted all (East Coast) afternoon
- We were mystified:
  - All individual functions worked
  - We had tested all statements and branches

The Telenova Stack Failure

A Simplified State Diagram Showing the Bug



The Telenova Stack Failure

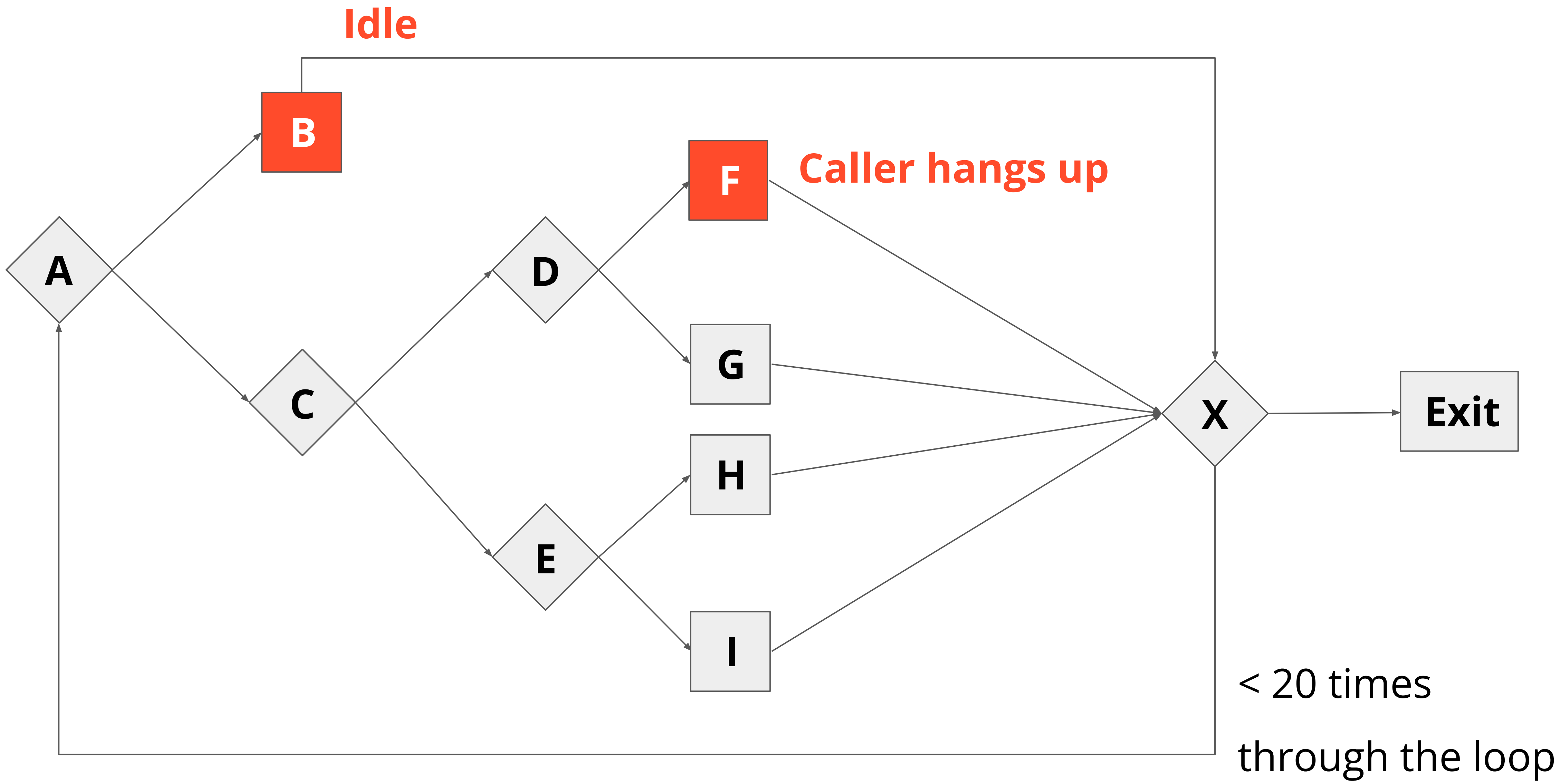


The Telenova Stack Failure

Ultimately, we found the bug in the hold queue

- Up to 10 calls on hold, each adds record to the stack
- Initially, the system checked the stack when any call was added or removed, but this took too much system time. So we dropped our checks and added these
  - Stack has room for 20 calls (just in case)
  - Stack reset (forced to zero) when we knew it *should* be empty
- The error handling made it almost impossible for us to detect the problem in the lab. Because a user couldn't put more than 10 calls on the stack (unless she knew the magic error), testers couldn't get to 21 calls to cause the stack overflow.

Sequences



The stack bug was just like this program, with a garbage collector at B (the idle state) and a stack leak at F (hang up from hold). If you hit F N times without touching B, when you try to put a 21-Nth call on hold, you overflow the stack and crash.

The Telenova Stack Failure

This example illustrates several important points:

- Simplistic approaches to path testing can miss critical defects.
- Critical defects can arise under circumstances that appear (in a test lab) so specialized that you would never intentionally test for them.
- When (in some future course or book) you hear a new methodology for combination testing or path testing, we want you to test it against this defect. If you had no suspicion that there was a stack corruption problem in this program, would the new method lead you to find this bug?

Summing Up

Testers live and breathe tradeoffs.

The time needed for test-related tasks is infinitely larger than the time available.

Time you spend on

- Analyzing, troubleshooting, and effectively describing a failure

Is time no longer available for

- Designing tests
- Documenting tests
- Executing tests
- Automating tests
- Reviews, inspections
- Supporting tech support
- Retooling
- Training other staff

Black Box Software Testing Foundations
Lecture 6
Measurement

Cem Kaner J.D., PH.D.

Professor Emeritus, Software Engineering, Florida Institute of Technology

Course Overview: Fundamental Topics

1. The Nature of Testing
   Overview and Basic Definitions

2. Why are we testing? What are we trying to learn? How should we organize our work to achieve this? ***Information objectives drive the testing mission and strategy***

3. How can we know whether a program has passed or failed a test?
   Oracles are heuristic

4. How can we determine how much testing has been done? What core knowledge about program internals do testers need to consider this question?
   Coverage is a multidimensional problem

5. Are we done yet?
   Complete testing is impossible

6. How much testing have we completed and how well have we done it? ←
   Measurement is important but hard

Today’s Readings

Required

- Cem Kaner & Walter P. Bond (2004), ”Software Engineering Metrics: What Do They Measure and How Do We Know?” https://kaner.com/pdfs/metrics2004.pdf

Useful to skim

- Robert Austin (1996), *Measurement and Management of Performance in Organizations.*
- Michael Bolton (2009), ”Meaningful Metrics”, http://www.developsense.com/blog/2009/01/meaningful-metrics/
- Doug Hoffman (2000), ”The Darker Side of Software Metrics”, https://bbst.courses/wp-content/uploads/2022/08/Hoffman_DarkerSideMetrics.pdf
- Erik Simmons (2000), ”When Will We Be Done Testing? Software Defect Arrival Modelling with the Weibull Distribution”, https://bbst.courses/wp-content/uploads/2022/08/Simmons_Weibull.pdf

What Brings Us to this Topic?

We seem to be asking quantitative questions, or questions that can be answered by traditional, quantitative research, such as:

- How much testing have we done?
- How thorough has our testing been?
- How effective has our testing been?

Are we meeting our information objectives? Do we need to adopt a different strategy?

- How much testing is enough?
- Are we done yet?

These are important questions. But they are difficult. We can't teach you how to answer them today. (We're working on that course…)

We can introduce you to the reasons that the popular, simplistic measures are often dysfunctional.

Basics of Measurement

- It’s not about counting things, it’s about **estimating the value of something**.
- We don’t count bugs because we care about the total number of bugs. **We count bugs because we want to estimate:**
    - the thoroughness of our testing, or
    - a product’s quality, or
    - a product’s reliability, or
    - the probable tech support cost, or
    - the skill or productivity of a tester, or
    - the incompetence of a programmer, or
    - the time needed before we can ship the
    - product, or
    - something else (whatever it is)…

Measurement

Measurement is the empirical, objective assignment of numbers to attributes of objects or events (according to a rule derived from a model or theory) with the intent of describing them.

Kaner & Bond discussed several definitions of measurement in Software engineering metrics: What do they measure & how do we know?

https://kaner.com/pdfs/metrics2004.pdf

Measurements Include

- The **ATTRIBUTE**: the thing you want to measure.
- The **INSTRUMENT**: the thing you use to take a measurement.
- The **READING**: what the instrument tells you when you use it to measure something.
- The **MEASURED VALUE** or the **MEASUREMENT** is the **READING.**
- The **METRIC**: the function that assigns a value to the attribute, based on the reading.
- We often say **METRIC** to refer to the **READING** or the **SCALE.**

If you're not sure what you're trying to measure, you probably won't measure it very well.

Measurement: Trivial Case

- Attribute: Width of the projector screen
- Instrument: Tape measure
- Reading: 40 inches (from the tape measure)
- Metric: inches on tape = inches of width

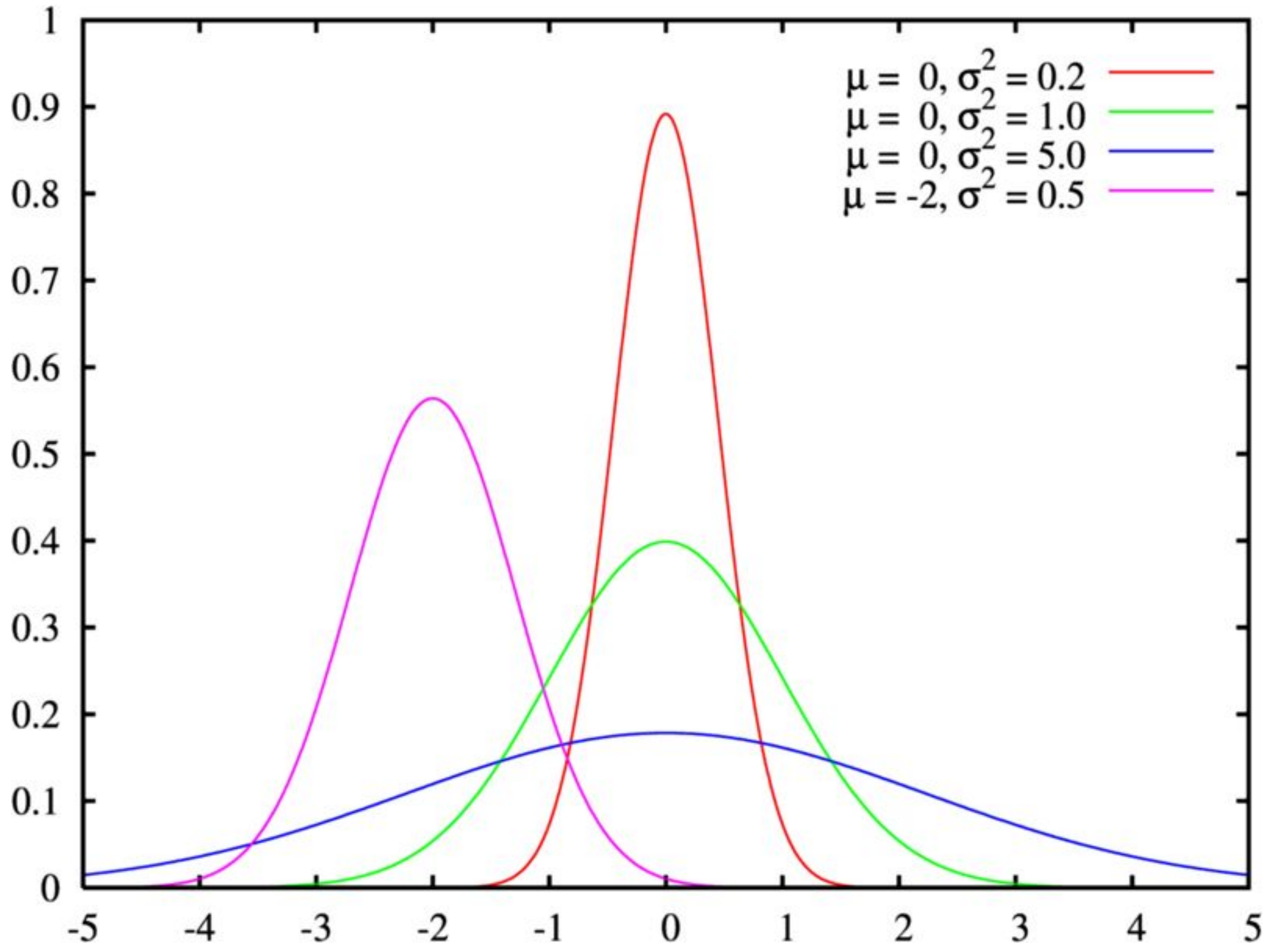
Measurement: Trivial Case

Even simple measurements have complexities:

- Measurement error (random variation in reading the tape)
- Precision of the measurement (inches? miles?)
- Purpose of the measurement
- Scope of the measurement (just this one screen?)
- Scale of the measurement (what you can read off the tape)

Measurement Error

- Measure the same thing 100 times and you'll get 100 slightly different measurements.
- Frequently, the distribution of measurement errors is Gaussian (a.k.a. Normal).



https://commons.wikimedia.org/wiki/File:Normal_distribution_pdf.png

Precision of Measurement

What are the units on your measuring instrument?

- inches? yards?
- if your tape measure has a mark every mile (like mile markers on the highway), do you think your measurements will be accurate to the nearest inch?
- how accurate is your measurement of a 1 mile road:
  - measured in inches with a 36-inch yardstick
    - high precision, but
    - high measurement error

Purpose of This Measurement

Why do you care how wide the projector screen is?

- estimate the size of text that will appear on the screen and thus its visibility?
- decide whether it will fit in your truck?
- reserve a room that isn't too small for the screen?
- estimate and control manufacturing error?
- chastise someone for ordering a screen that is too big or too small?

To control manufacturing error (width of the screen), you usually want high consistency and precision of measurements.

Try using a "five-why" analysis to figure out your underlying purpose.

http://en.wikipedia.org/wiki/5_Whys

Scope of Measurement

- Just this one screen?
- Just screens in this building?
- Just screens from this manufacturer?
- Just screens manufactured this year?

As the scope broadens, the more variables come into play, introducing more causes for measurement error.

Scale of Measurement

The ”scale” attaches meaning to the number.

For example, we can read ”40” from the tape measure, but that doesn't tell us much:

40 what?

In this case, our scale is ”inches.”

We know several things about inches:

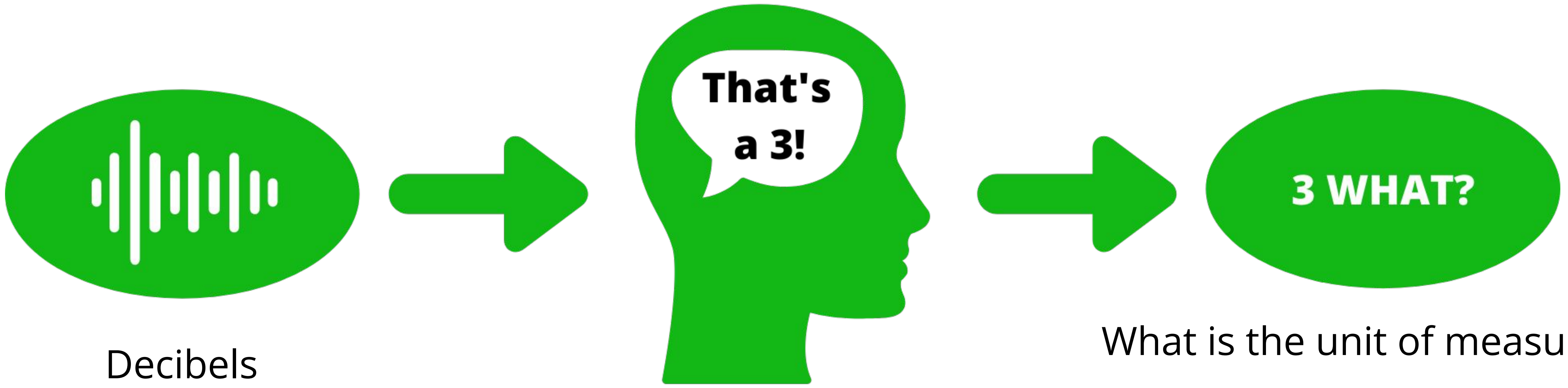
- We have an agreed standard for how long an inch is. We can check a tape measure against that standard.
- Every inch is the same as every other inch. They are all the same length.
- Three inches is three times as long as one inch.

Scale of Measurement

It is easy to count things, but **unless we have a model that**:

- maps the count to a scale, and
- maps that scale to the scale of the the underlying attribute

we won't know how to interpret the count. It will just be a number.



That's a 3!

3 WHAT?

Decibels (physical measurement)

Rates loudness

What is the unit of measurement?

What are our sound-inches?

How much bigger is 6 than 3?

Is a 6 twice as loud as a 3?

Scale of Measurement

Ratio Scale: a / b = (k * a) / (k * b) (for any constant, k)

e.g., the tape measure

20 inches / 10 inches = 200 inches / 100 inches.

- “An absolute zero is always implied”

See S.S. Stevens (1946), ”On the theory of scales of measurement”, *Science*, https://psychology.okstate.edu/faculty/jgrice/psyc3214/Stevens_FourScales_1946.pdf

Scale of Measurement

Interval Scale: a - b = (k + a) - (k + b)

e.g., Fahrenheit or Centigrade temperatures

- These have no true zero, so ratios are meaningless
  - 100° Fahrenheit = 37.8° Centigrade
  - 50° Fahrenheit = 10.0° Centigrade
  - 100 / 50 ≠ 37.8 / 10
- But intervals are meaningful
  - The difference in temperature between 100 and 75 Fahrenheit is the same as the difference in temperature between 75 and 50 (25° Fahrenheit and 13.9° Centigrade) in each case.
- We can have an interval scale when we don't have (don't know/use) true zero (compare to Kelvin temperature scale, which has true zero).

Multiplying interval-scale numbers is meaningless. (A product of two interval-scale numbers has no unambiguous mathematical meaning.)

Scale of Measurement

Ordinal Scale: If a > b and b > c, then a > c

e.g. winners of a race or a cooking contest

- 1st place is better than 2nd place.
- 2nd place is better than 3rd place.
- How much better?
  - Better.

Adding or multiplying ordinal-scale numbers is meaningless.

Scale of Measurement

Nominal Scale: a = b (a and b have the same name) if and only if a and b are the same e.g. names of people or companies

- Joe is Joe.
- Joe is not Susan.
- If Joe runs in a race and has a ”2” on his back, that doesn't mean he is faster or slower than Susan (she has a ”1” on her back). It just means that
  - Joe has the label ”1”,
  - Susan has the label ”2”.

Ranking or adding or multiplying nominal-scale numbers is meaningless.

Non-Trivial Measurement

Relationship between attributes and measuring instruments are often **not** straightforward. Some Attributes:

- quality
- reliability
- productivity
- supportability
- size of a program
- predicted schedule
- speed of a program
- predicted support cost
- complexity of a program
- extent of testing done so far
- quality of testing done so far
- completeness of development

What's the tape measure for this attribute?

Measurements Include

- The **ATTRIBUTE**: the thing you want to measure.
- The **INSTRUMENT**: the thing you use to take a measurement.
- The **READING**: what the instrument tells you when you use it to measure something.
- The **MEASURED VALUE** or the **MEASUREMENT** is the **READING.**
- The **METRIC**: the function that assigns a value to the attribute, based on the reading.
- We often say **METRIC** to refer to the **READING** or the **SCALE.**

If you don't know what you're trying to measure, you won't measure it well.

And you can do a lot of damage in the process.

Construct Validity

(Measurement Validity)

Construct validity:

- **Does this measure what I think it measures?**
  - Does this measure the attribute?
  - (Social scientists often say ”construct” where I say ”attribute”)
- Most important type of validity
- Widely discussed in measurement theory
  - But our field routinely skips the question, ”what is the attribute”
  - search ACM’s Digital Library or IEEE portal for ”construct validity” or ”measurement validity”

Valid metrics are extremely useful.

Invalid metrics cause dysfunction.

Surrogate (or Proxy) Measures

”Many of the attributes we wish to study do not have generally agreed methods of measurement. To overcome the lack of a measure for an attribute, some factor which **can be** measured is used instead. This alternate measure is presumed to be related to the actual attribute with which the study is concerned. These alternate measures are called surrogate measures.”

A widely used opportunity for disaster

See Johnson, M.A. (1996) *Effective and Appropriate Use of Controlled Experimentation in Software Development Research*, Master's Thesis (Computer Science), Portland State University, https://pdxscholar.library.pdx.edu/cgi/viewcontent.cgi?article=6361&context=open_access_etds

Surrogate (or Proxy) Measures

Example

- We don’t know how to measure tester productivity
- So let’s count bugs
  - Bugs are our surrogate for productivity
  - We assume that they must be correlated with productivity
  - And they’re easy to count

Predictable Mischief…

If we reward testers based on their bug counts, how much time will testers spend:

- Documenting their tests?
- Coaching other testers?
- Researching and polishing bug reports to make their bug easier to understand, assess and replicate?
- Running confirmatory tests (such as regression tests) that have control value (e.g. build verification) but little bug-find value?
- Hunting variants of the same failure to increase bug count?

Using surrogate measures can make things worse, while providing little useful information.

Yet, the ease of using them makes them quite popular.

We’ve Seen This Before

(Coverage)

People optimize what we measure them against, at the expense of what we don’t measure.

- Driving testing to achieve ”high” coverage is likely to yield a mass of low-power tests.
- Brian Marick discusses this in ”How to Misuse Code Coverage”, http://www.exampler.com/testing-com/writings/coverage.pdf
- What other side-effects might we expect from relying on coverage numbers as measures of **how close we are to completion of testing?**
- How is this different from using coverage measures to tell us **how far we are from an adequate level of testing?**

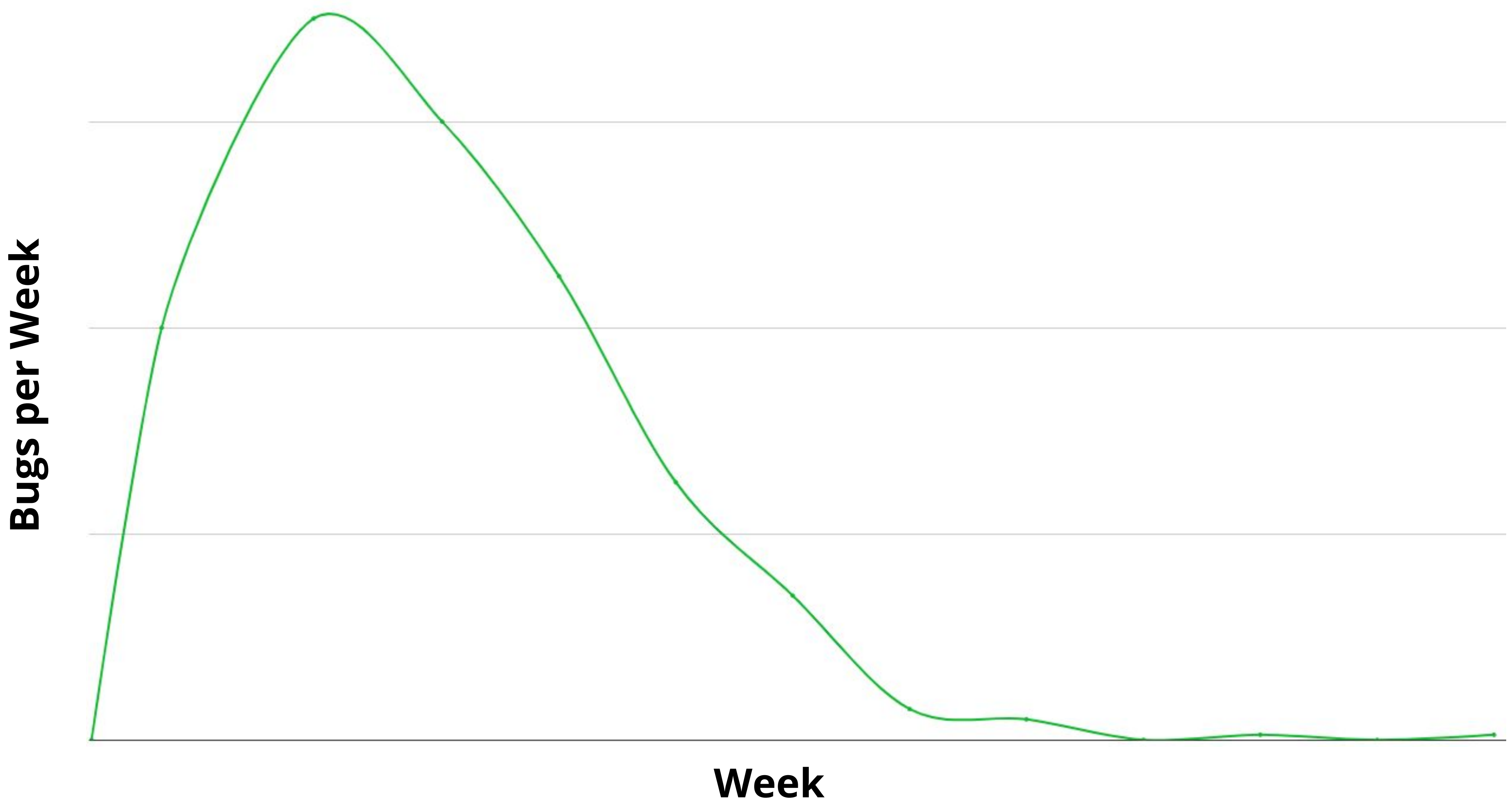
For more on measurement distortion and dysfunction,
read Bob Austin’s book, *Measurement and Management of Performance in Organizations*.

Another Example of Bug Counting

- Some people think the Weibull reliability model can be applied as model of testing progress.
- They estimate likely ship date by using **bug-find rates** to estimate parameters of the Weibull curve.

The Weibull Curve

New bugs found per week (”Defect arrival rate”)



Related measures:

- **Bugs still open (each week)**
- **Ratio of bugs found to bugs fixed (per week)**

Weibull Model Assumptions

These assumptions are **wildly implausible as models of testing:**

- Testing occurs in a way similar to the way the software will be operated.
- All defects are equally likely to be encountered.
- Defects are corrected instantaneously, without introducing additional defects.
- All defects are independent.
- There is a fixed, finite number of defects in the software at the start of testing.
- The time to arrival of a defect follows the Weibull distribution.
- The number of defects detected in a testing interval is independent of the number detected in other testing intervals for any finite collection of intervals.

See Erik Simmons (2000), ”When Will We Be Done Testing? Software Defect Arrival Modeling with the Weibull Distribution.”
https://bbst.courses/wp-content/uploads/2022/08/Simmons_Weibull.pdf

The Weibull Model

- An advocate of this approach asserts:

  ”Luckily, the Weibull is robust to most violations.”
- From a purely curve-fitting point of view, this is correct: The Weibull distribution has a shape parameter that allows it to take a very wide range of shapes. If you have a curve that generally rises then falls (one mode), you can approximate it with a Weibull.

The Weibull Model

This illustrates the use of surrogate measures

- we don’t have an attribute description or model for the attribute we really want to measure,
- so we use something else, that is convenient, and allegedly ”robust,” in its place.

Side Effects (the Predictable Mischief) of Bug Curves

When development teams are pushed to show project bug curves that look like the Weibull curve, they are pressured

- to show a rapid rise in their bug counts,
- an early peak,
- and a steady decline of bugs found per week.

Under the model, a rapid rise to an early peak predicts a ship date much sooner than a slower rise or a more symmetric curve. In practice, project teams (including testers) in this situation often adopt dysfunctional methods, doing things that will be bad for the project over the long run in order to make the numbers go up quickly.

For more observations of problems like these in reputable software companies, see Doug Hoffman, ”The Dark Side of Software Metrics”, https://bbst.courses/wp-content/uploads/2022/08/Hoffman_DarkerSideMetrics.pdf

What Is the Predictable Mischief?

Early testing:

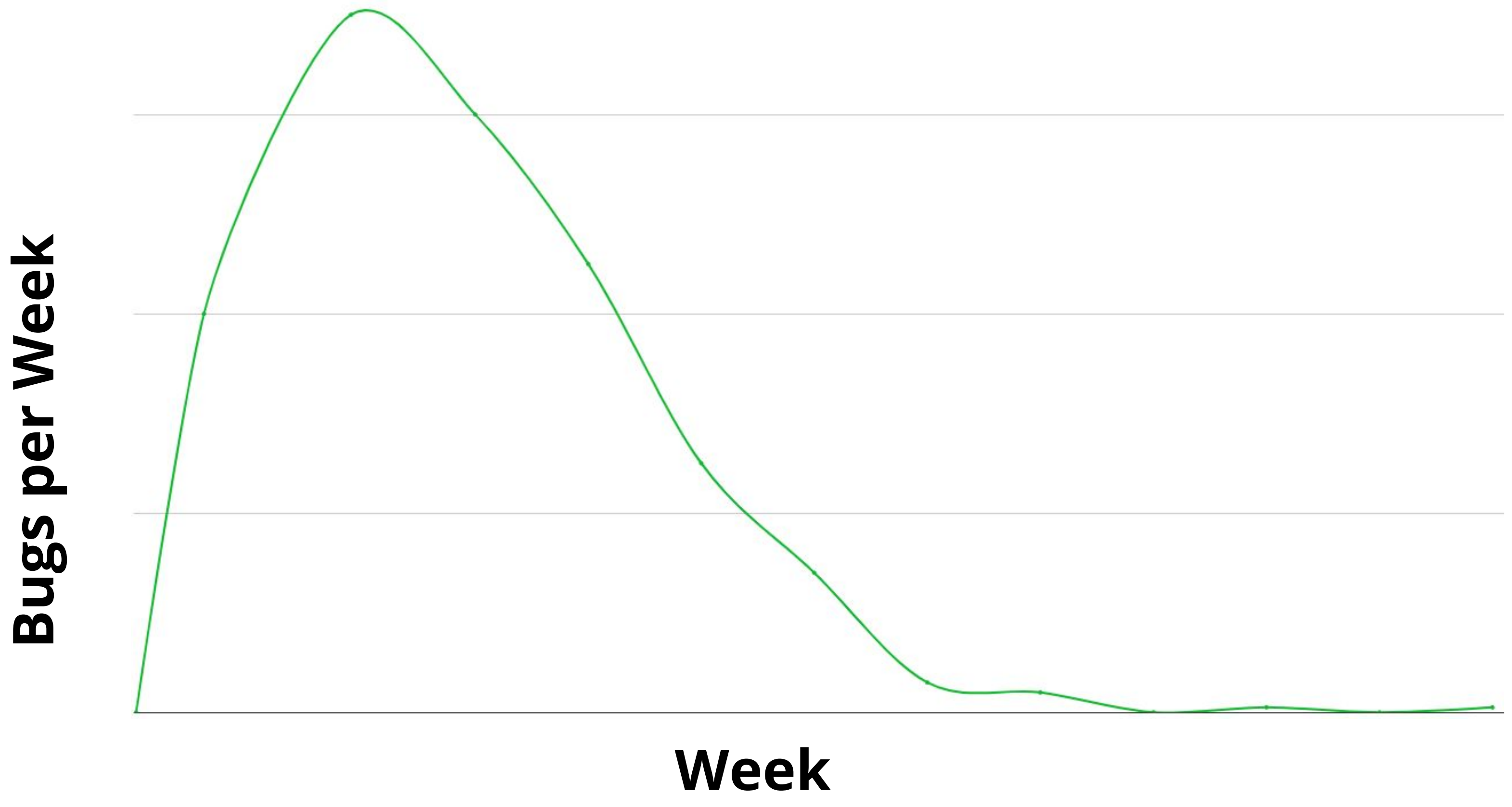
- Run tests of features known to be broken or incomplete.
- Run multiple related tests to find multiple related bugs.
- Look for easy bugs in high quantities rather than hard bugs.
- Less emphasis on
  - infrastructure,
  - automation architecture,
  - tools and documentation.
- More emphasis on bug finding. (Short term payoff but long term inefficiency.)

The goal is to find lots of bugs early.

Get to that peak in the curve right away.

After the Peak

After we get past the peak, the expectation is that testers will find fewer bugs each week than they found the week before. Based on the number of bugs found at the peak, and the number of weeks it took to reach the peak, the model can predict bugs per week in each subsequent week.



More Predictable Mischief

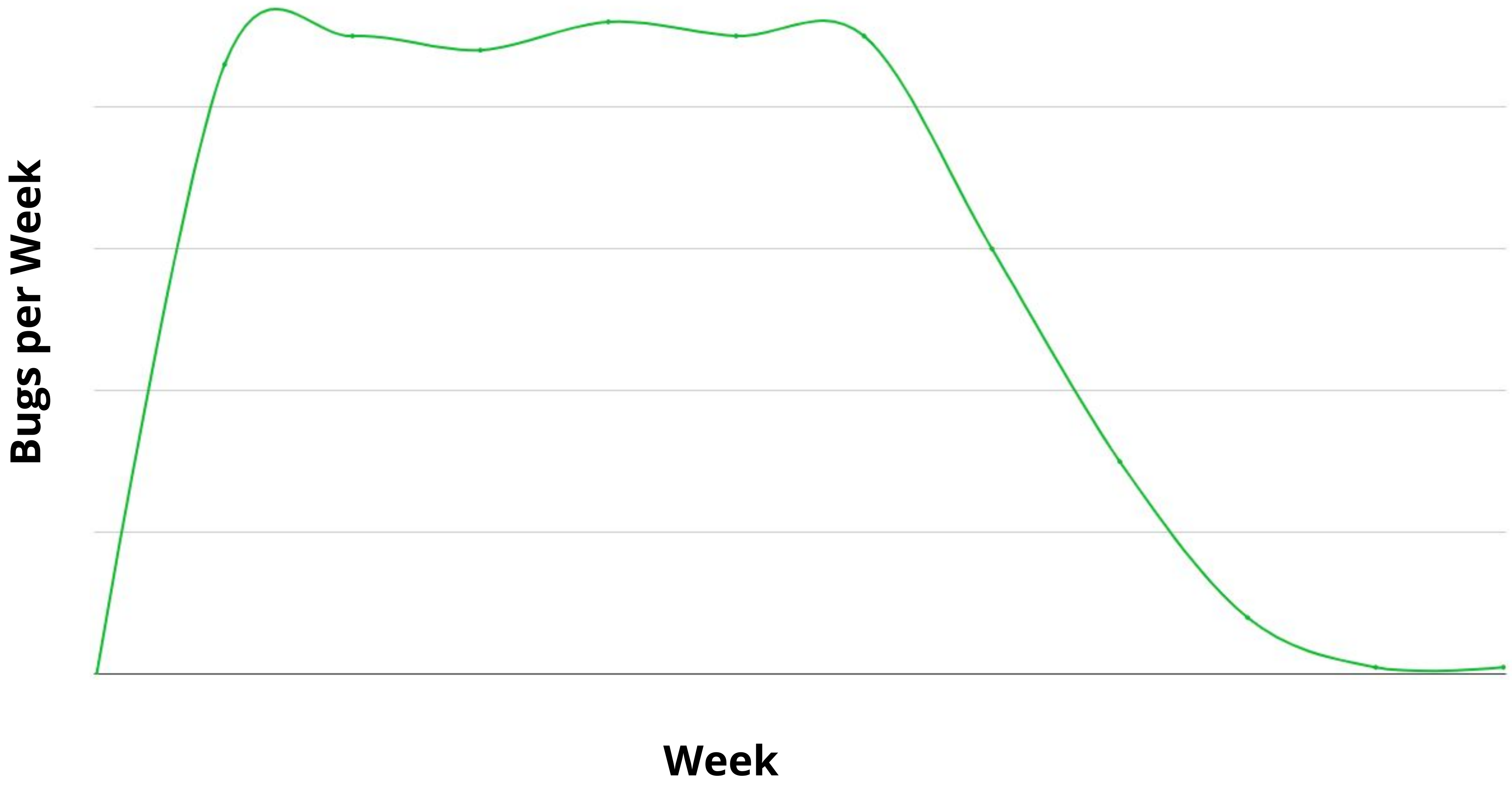
Later in the project:

- Run lots of already-run regression tests
- Don’t look as hard for new bugs
- Shift focus to status reporting
- Classify unrelated bugs as duplicates
- Close related bugs as duplicates, hiding key data about the symptoms/causes of a problem
- Postpone bug reporting until after a measurement checkpoint (milestone) (Some bugs are lost)
- Programmers ignore bugs until testers report them
- Testers report bugs informally, keep them out of the tracking system
- Project team sends testers to irrelevant tasks before measurement checkpoints
- More bugs are rejected, sometimes taken personally

We expect fewer bugs every week.

Bad Models Are Counterproductive

Shouldn’t we strive for **this**?



Distortion and Dysfunction

People optimize what we measure them against, at the expense of what we don’t measure.

- A measurement system yields **distortion** if it creates incentives for a person to make the measurements **look better** rather than to optimize for achieving the organization's actual goals.
- A system is **dysfunctional** if optimizing for measurement yields so much distortion that the result is a **reduction of value:** the organization would have **been better off with no measurement** than with this measurement.

For more on measurement distortion and dysfunction, read Bob Austin’s book, *Measurement and Management of Performance in Organizations*.

Recap: Three Examples

- Measuring the effectiveness of testing by counting bugs is fundamentally flawed. Therefore measuring the effectiveness of a testing strategy by bug counts is probably equally flawed.
- Measuring code coverage not only misleads us about how much testing there has been. It also creates an incentive for programmers to write trivial tests.
- Measuring progress via bug count rates not only misleads us about progress. It also drives test groups into dysfunctional conduct.

What Should We Do?

- This is a difficult, unsolved problem
- General recommendations
- Details to come in later courses

Imagine evaluating employee performance:

- Break down the job into a list of key tasks.
- For each of this person's key tasks, take a small sample (10?) of this person's work.
- Carefully evaluate each type of work, looking at the details.
- **Then** you can rank performance, or assign metric summary numbers to the individual tasks.
- And combine ratings across tasks for an employee profile.

We'll study an example of task evaluation in the Bug Advocacy course.