CIS 547
Software Analysis

Software Specifications

Mayur Naik

Penn Engineering

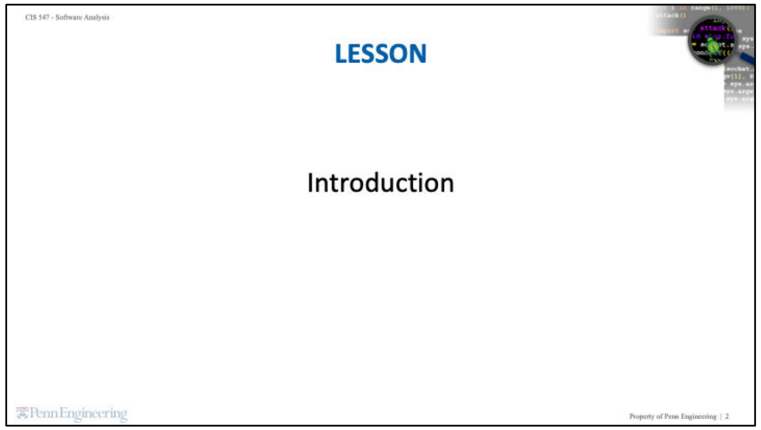
In the first module, we learned the basics of analyzing the implementation of a given piece of software. Perhaps as important as the implementation is the specification -- a high-level description of the software's intended behavior.

In this module, we will learn about software specifications, and the role they play in ensuring software quality.

While every software analysis needs a specification to begin with, in this module, we will focus on the use of specifications in the simplest and most common form of analyzing software that is practiced in industry – testing.

We will introduce the landscape of testing methods along two key dimensions and motivate automated testing – a topic that we will delve deeper into later in the course.

We will study different kinds of formal specifications for use in testing. Since specifications can be tedious for programmers to write, we will look at a technique to automatically infer them, from the program's implementation itself. We will conclude by discussing methods to measure the adequacy of a test suite.



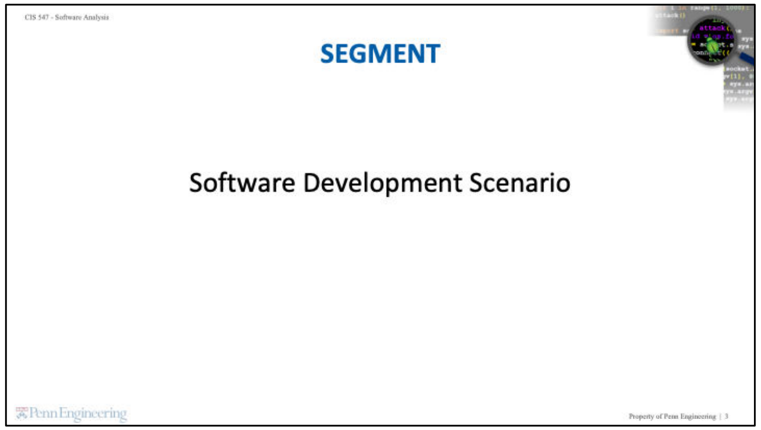
CIS 547 - Software Analysis

LESSON

Introduction

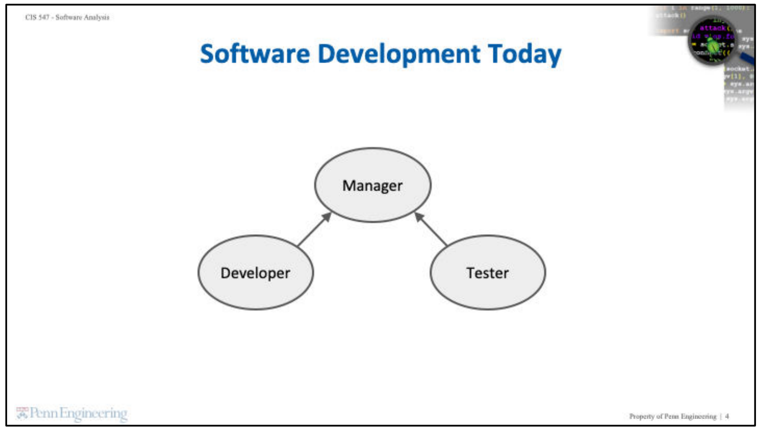
Penn Engineering

Property of Penn Engineering | 2



SEGMENT

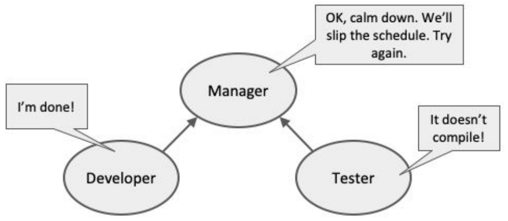
Software Development Scenario



Software Development Today

Let’s start with an overview of how a typical software development team works. A team typically consists of at least one developer, at least one tester, and a manager to which both the developers and testers report.

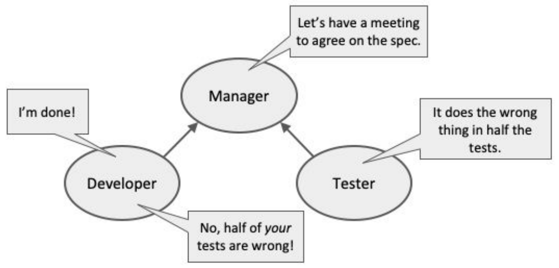
A Typical Scenario



The developer finishes writing some code for a project, and then sends it to the tester to make sure the code works.

The tester, however, is unable to even compile the code, and the manager has to push back the schedule so the problem can be fixed.

A Typical Scenario

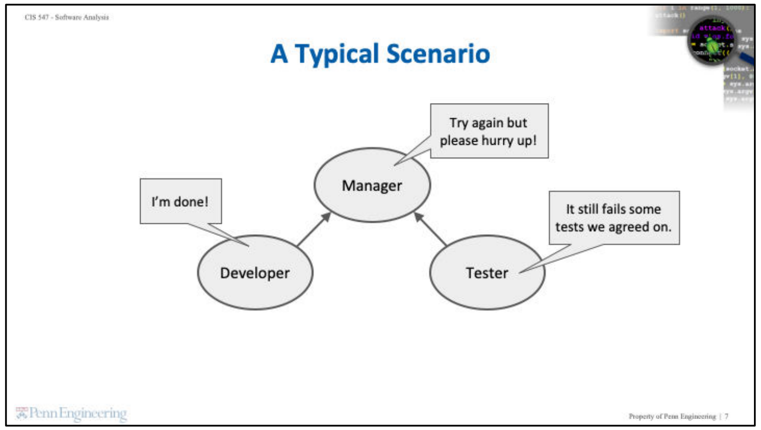

Let's have a meeting to agree on the spec.
I'm done!
Manager
It does the wrong thing in half the tests.
Developer
Tester
No, half of *your* tests are wrong!

The developer fixes the code and sends it to the tester again.

This time, the tester is able to compile the code, and runs a test suite against the compiled code, but finds that the code does the wrong thing in half the tests.

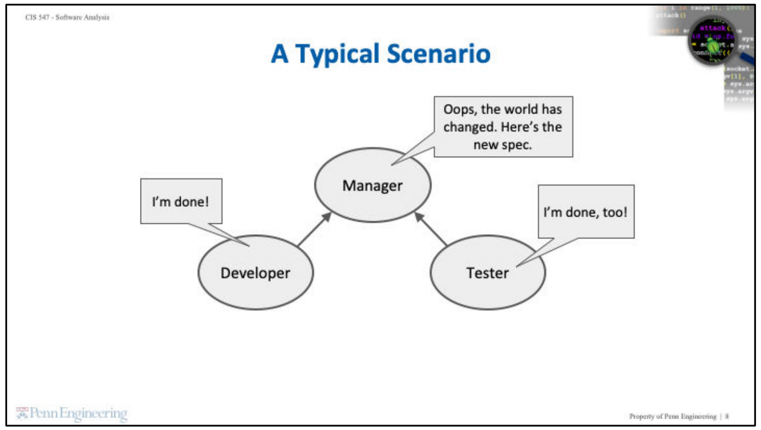
This time, the developer claims that the problem lies not in the code but in the test suite, and that half of the tests in the test suite must be wrong.

So the manager has to step in and make sure the developer and tester agree on the specifications for the program.

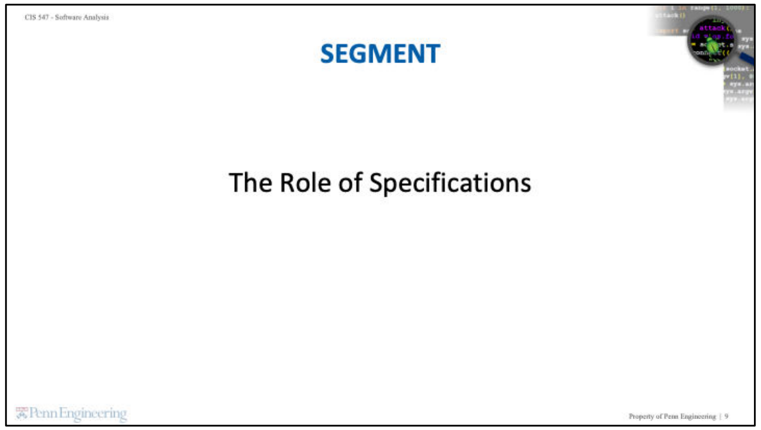

CIS 547 - Software Analysis
A Typical Scenario
Try again but please hurry up!
Manager
I'm done!
It still fails some tests we agreed on.
Developer
Tester
Penn Engineering
Property of Penn Engineering | 7

The developer fixes the code again according to the newly decided specifications, but the tester still finds problems with the code.

So the manager has to push back the project again.


CIS 547 - Software Analysis
A Typical Scenario
Oops, the world has changed. Here's the new spec.
Manager
I'm done!
I'm done, too!
Developer
Tester
Penn Engineering
Property of Penn Engineering | 8

Finally, the developer finishes implementing the program to the satisfaction of the tester. But then the requirements for the program change, and the team is back to square one.

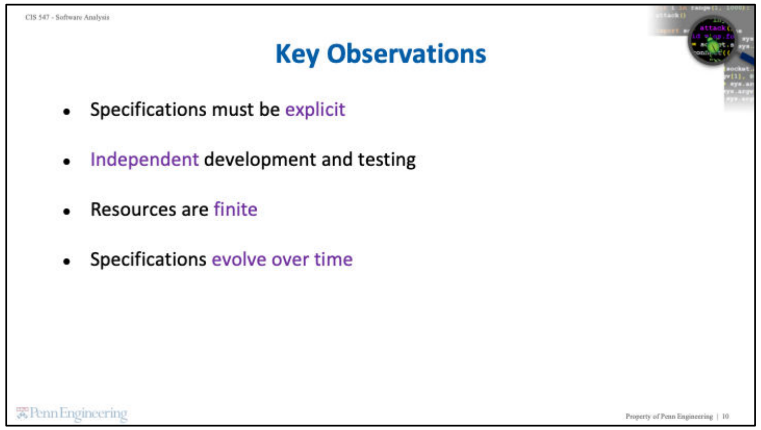

CIS 547 - Software Analysis

SEGMENT

The Role of Specifications

Penn Engineering

Property of Penn Engineering | 9


CIS 547 - Software Analysis

Key Observations

- Specifications must be explicit
- Independent development and testing
- Resources are finite
- Specifications evolve over time

Penn Engineering

Property of Penn Engineering | 10

We can make several key observations from this typical development scenario.

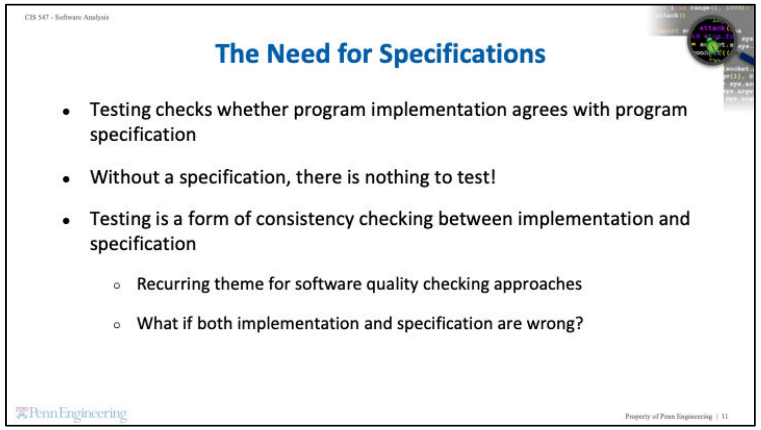
First, program specifications have to be made explicit, or else there cannot be effective communication between those implementing code and those testing it.

Second, development and testing of a program are often done independently. This helps to ensure that errors in the code are caught.

Third, there are only finitely many resources available to development teams; not every bug can be found and caught.

Fourth, program specifications are not static; they evolve over time, and programming teams need to be able to adapt to these changes.

Let's delve deeper into each of these observations.


CIS 547 - Software Analysis

The Need for Specifications

- Testing checks whether program implementation agrees with program specification
- Without a specification, there is nothing to test!
- Testing is a form of consistency checking between implementation and specification
    - Recurring theme for software quality checking approaches
    - What if both implementation and specification are wrong?

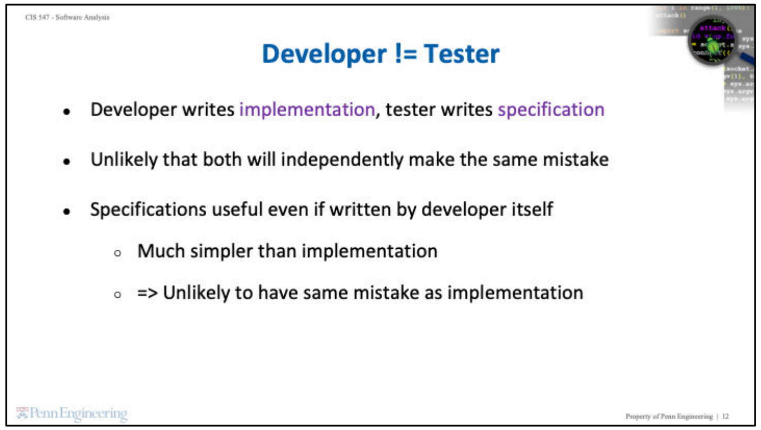
Penn Engineering   Property of Penn Engineering | 11

The first observation from the scenario we saw was that specifications must be explicit. This is because the goal of testing is to check whether the implementation of the program agrees with a specification of the program. We will come to the topic of what specifications look like shortly.

But the main thing to note for now is that, without a specification, there is nothing to test! Testing, then, can be viewed as a form of consistency checking between the program's implementation and specification. This is a recurring theme for all software quality checking approaches including testing.

This points to a potential shortcoming of all these approaches including testing: both the implementation and specification could be wrong, and these approaches would not be able to notice the problem, as they would declare the two to be consistent.

This leads us to our second observation.


CIS 547 - Software Analysis

Developer != Tester

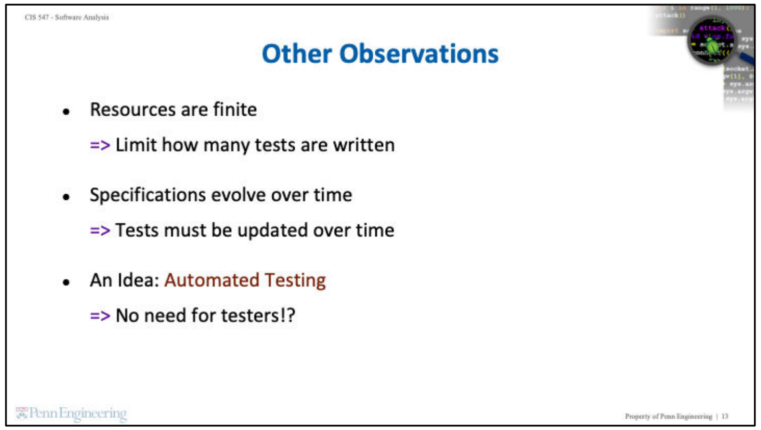
- Developer writes implementation, tester writes specification
- Unlikely that both will independently make the same mistake
- Specifications useful even if written by developer itself
    - Much simpler than implementation
    - => Unlikely to have same mistake as implementation

Penn Engineering   Property of Penn Engineering | 12

The second observation from the scenario we saw was that the developer and tester were independent. The developer writes the program implementation, that is, what the program actually does, while the tester writes a program specification, that is, a facet of what the program is expected to do.

Since the two parties are independent, it is less likely that both will make the same mistake, that is, the developer will commit a bug in the program's implementation and the tester will affirm that same bug in the specification. It is due to this independence that consistency checking makes sense.

So we saw that testing is useless without specifications. We can also ask the converse question: are specifications useless without testers? That is, suppose you are a developer and there is no independent tester for the program you are developing. Then, does it make sense for you to write specifications? The answer is yes. This is because, even though the same person -- that is, the developer -- writes both the implementation and the specification, the specification artifact is typically much simpler than the implementation artifact. This is because each specification checks only one facet of the implementation. So the specification is still unlikely to contain the same bug as the implementation.



Other Observations

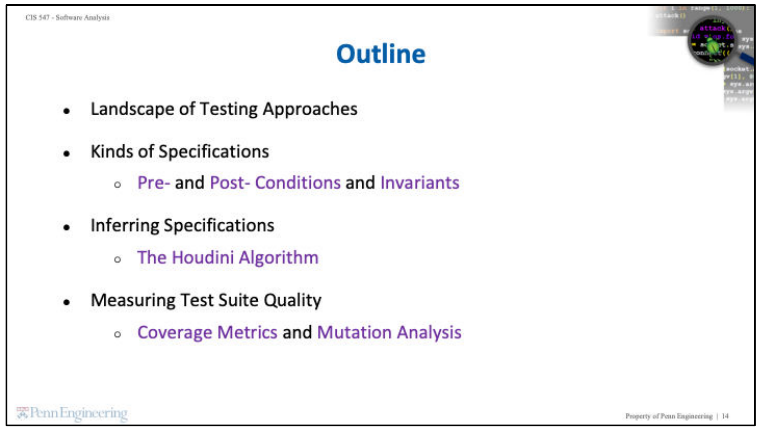
- Resources are finite
  => Limit how many tests are written
- Specifications evolve over time
  => Tests must be updated over time
- An Idea: Automated Testing
  => No need for testers!?

Additionally, the resources available to a development team are limited and must be prioritized. There is a finite number of tests that can be written and run for a given piece of code. The tests we write for a program will only be able to check certain facets of the program; they won't be able to check every possible use case.

And the specifications for a program change over time, so a given set of tests for a program will not necessarily remain valid for the lifetime of the software. Resources must be spent on updating test suites in order to reflect the new specifications.

Given all these observations---which we can summarize as the fact that testing is a necessary yet ongoing, resource-intensive process---wouldn't it be nice to be able to *automate* the process of writing and running tests? In the extreme case, we'd never need another QA department ever again!

While we are certainly not at the point of obsoleting human testing, this idea of automated testing is attractive. In future modules, we will explore some of the basics of automated testing and introduce you to some of the automated testing techniques currently used in the industry today.



Outline

- Landscape of Testing Approaches
- Kinds of Specifications
  - Pre- and Post- Conditions and Invariants
- Inferring Specifications
  - The Houdini Algorithm
- Measuring Test Suite Quality
  - Coverage Metrics and Mutation Analysis

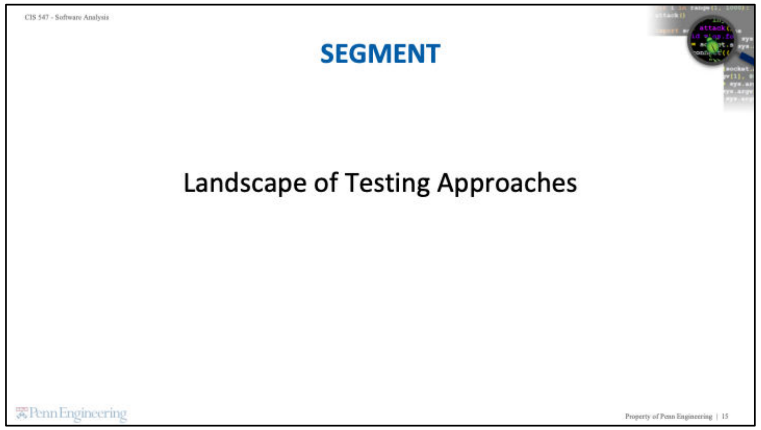
Here is a road map for the remainder of this module.

We will begin by surveying the landscape of testing paradigms, comparing and contrasting their costs and benefits. By the end of this module, you should have an understanding of what characterizes different testing strategies and should be able to select an appropriate strategy given a set of priorities and constraints.

Next, we will look at specifications in more detail. In particular, you will see how to use pre- and post-conditions and invariants to specify the behavior of units of a program such as loops, methods, and classes.

Then, we will discuss an algorithm to automatically infer specifications, called the Houdini Algorithm. The Houdini algorithm makes the costly task of writing specifications easier.

Finally, we will wrap up the module by studying two methods of quantifying the quality of a given set of tests, or test suite. These methods will allow you to determine whether a test suite for a program is doing its job or whether it needs to be augmented with more tests.

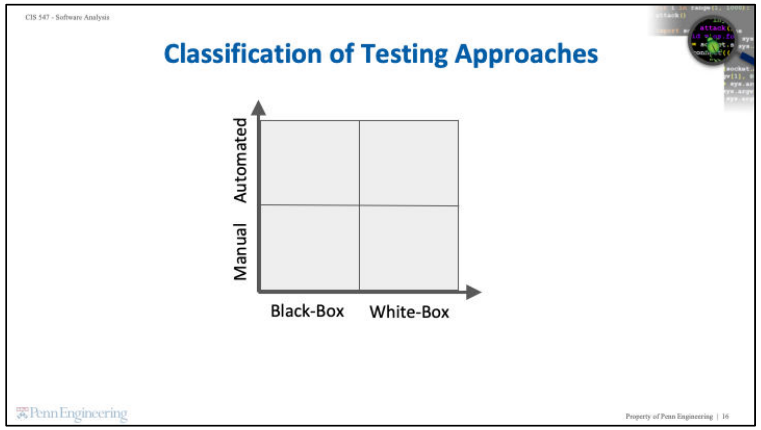

CIS 547 - Software Analysis

SEGMENT

Landscape of Testing Approaches

Penn Engineering

Property of Penn Engineering | 15


CIS 547 - Software Analysis

Classification of Testing Approaches

Penn Engineering

Property of Penn Engineering | 16

Approaches to testing software can be classified according to two orthogonal axes: manual vs. automated and black-box vs. white-box.

The vertical axis describes the amount of human participation in the testing process: testing that requires more human direction falls closer to the manual side of the axis, while testing that is performed primarily without human direction falls closer to the automated side of the axis.

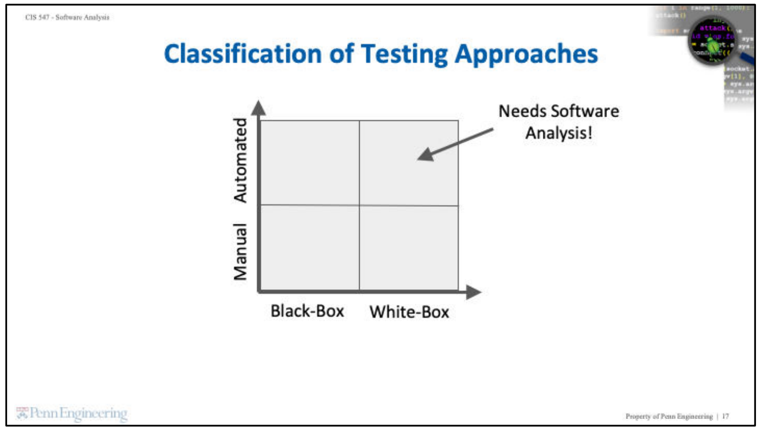
The horizontal axis describes the amount of access the testing apparatus has to the tested program's source code.

Black-box testing refers to testing where the tester can see nothing about the tested program's internal mechanisms: as though the program is contained inside an opaque box. The tester can only issue inputs to the program, observe the program's outputs, and determine whether they meet the specifications required of the program.

White-box testing refers to testing in which the internal details of the program being tested are fully available to the tester. The tester can use these internal details to perform a more precise analysis of the tested program and uncover inputs that are more likely to trigger buggy behavior.

Testing approaches need not be strictly black-box or white-box; some internal details may be available to the tester while others are hidden. These sorts of testing approaches are called “gray-box” approaches.

Similarly, testing approaches need not be fully manual or fully automatic. It is better to think of these axes as continua instead of as discrete categories.



Classification of Testing Approaches

Let’s look at some specific examples of testing approaches and see where they fall along the two axes.

One testing approach is for the tester to tinker with observational behavior of a program: for example, exercising different GUI events of an Android app. This sort of testing would fall somewhere near the bottom-left of this diagram because the tester is only issuing commands to and observing outputs from the program under testing, and this is done on a manual basis. [mark X in bottom left]

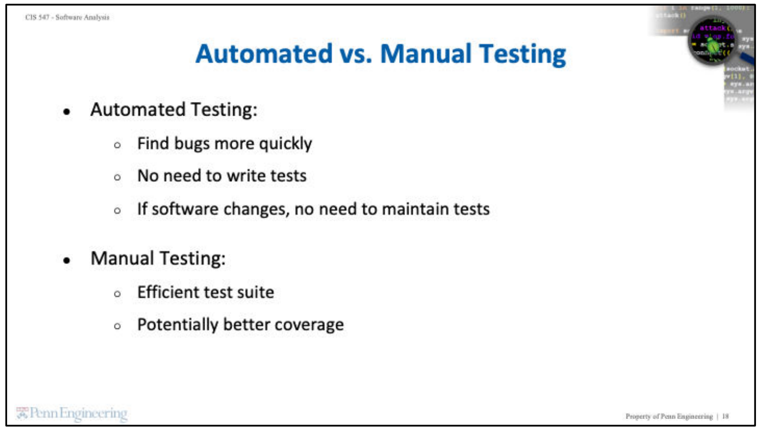
Now, if we were to modify this testing approach so that the tester looks at the source code in order to determine what possible routes there are through the app’s GUI, then we have taken the original approach and moved it rightward to the white-box side of the diagram [mark X in bottom right]

You are probably familiar with both of these types of testing. Testing approaches we will learn in this course primarily will focus on the top two quadrants, that is, how to leverage modern tools and techniques to automate testing.

For example, instead of manually activating GUI events for the Android app, we might use an automated approach such as a fuzzer to automatically issue tap commands to random coordinates of the smartphone. This takes the original black-box approach and moves it upward to the automated side of the diagram. [mark X in top left] This is not a very sophisticated approach, but it has many advantages that we will see in the next module, on random testing.

We might also take approaches such as feedback-directed random testing (issuing random commands that change in response to feedback issued by the GUI), perform symbolic execution that needs to inspect the source code in order to test effectively (i.e. perform static analysis), or even monitor the code as it is being tested (i.e. perform dynamic analysis) in order to discover future tests intelligently. The approaches take us to the top-right quadrant of the diagram, in which we are performing automated, white-box testing of a program [mark X in top right]. We've already alluded to some of these approaches earlier in the first module; we will discuss them later in the course.

Next let's look at pros and cons of different approaches along each of these two dimensions.



CIS 547 - Software Analysis

Automated vs. Manual Testing

- Automated Testing:
  - Find bugs more quickly
  - No need to write tests
  - If software changes, no need to maintain tests
- Manual Testing:
  - Efficient test suite
  - Potentially better coverage

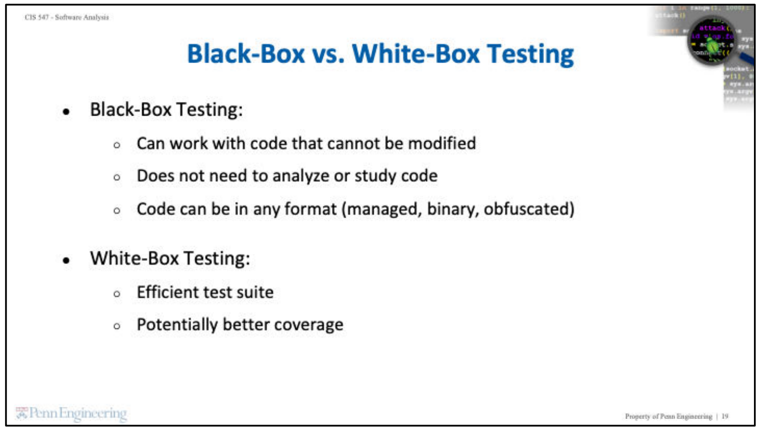
Penn Engineering

Property of Penn Engineering | 18

One of the advantages of automated testing over manual testing is that we can potentially spot bugs more quickly through the speed advantage of a computer over a human in issuing inputs and checking outputs. Additionally, there is no need to write tests: they are generated by the computer itself. Moreover, if the software changes, there is no need to update the tests by hand, as the computer will generate new tests relevant to the updated software.

On the other hand, an advantage of manual testing is that humans are potentially better able to select an efficient set of tests: computer-generated test suites can be rather bloated. Additionally, humans can potentially construct a test suite with better code coverage than a computer program could, though this is not guaranteed.

The ideal approach will often lie in the combination of automated and manual approaches: a semi-automated approach. An example of such an approach is where a human specifies the format or the grammar of valid inputs to a program, so that the automated testing that follows does not waste resources generating invalid inputs that do not exercise any interesting functionality of the program.



Black-Box vs. White-Box Testing

- Black-Box Testing:
  - Can work with code that cannot be modified
  - Does not need to analyze or study code
  - Code can be in any format (managed, binary, obfuscated)
- White-Box Testing:
  - Efficient test suite
  - Potentially better coverage

Black-box testing has many advantages over white-box testing. First, it does not require modifying the code, that is, introducing probes into the code. This is a big advantage because in many practical cases, the tester does not have the liberty to modify code.

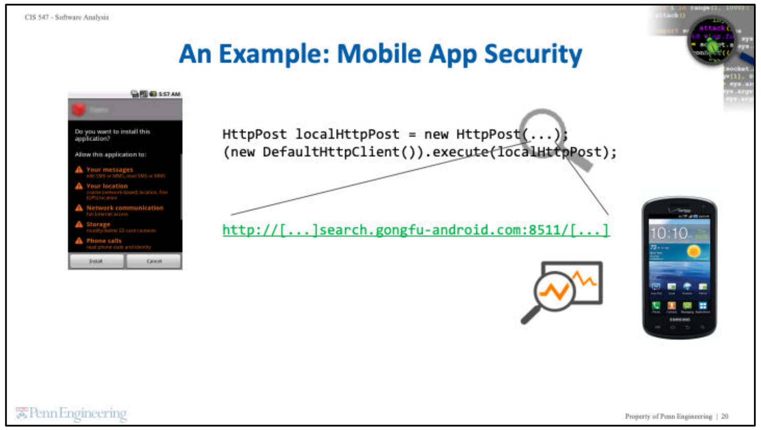
For instance, consider an Android application. Its code comprises not only the code of the application itself but also parts of the Android framework that the application is built upon. So a tester would need to modify not only the application's code but also the entire framework's code.

Additionally, black-box testing does not need to study the code to be tested. The problem with analyzing this code is that it might be too low-level of an analysis: the tool might get lost in details of the program and lose sight of the bigger picture that observing inputs and outputs provides.

Moreover, black-box testing can be performed on any format of code, whether it is managed code, binary code, obfuscated code, etc.

The advantages of white-box testing over black-box testing are similar to those of manual over automated testing: the tester is potentially able to construct a more efficient suite of test cases with potentially better coverage than black-box testing could.

Both kinds of approaches are useful, and typically a combination is needed. Let's take a look at a concrete example next that illustrates black-box and white-box testing.


CIS 547 - Software Analysis

An Example: Mobile App Security

Do you want to install this application?

Allow this application to:

- Your messages
- Your location
- Network communication
- Storage
- Phone calls

Install | Cancel

```
HttpPost localHttpPost = new HttpPost(...);
(new DefaultHttpClient()).execute(localHttpPost);
```

http://[...]search.gongfu-android.com:8511/[...]

Property of Penn Engineering | 20

Let's compare and contrast black-box and white-box testing for the problem of determining whether an Android app is malicious.

Consider the DroidKungFu malware, which was found to be in circulation on eight 3rd-party app stores based out of China around 2011. Prior to installation, this app asks for the following permissions. [Permissions on the left of the slide appear]

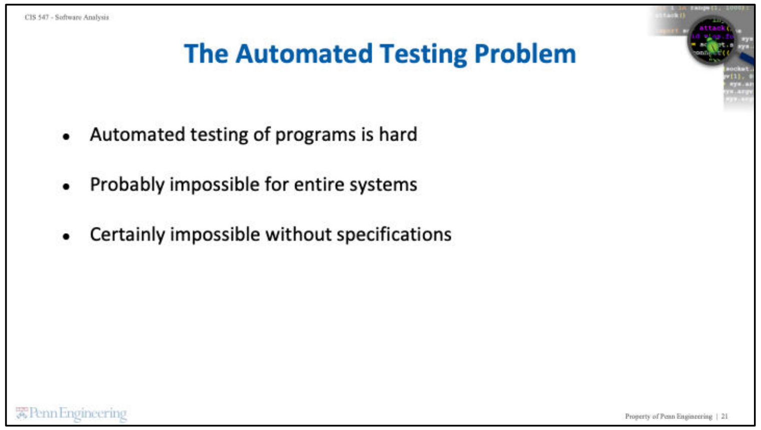
Once installed, the app attempts to collect sensitive information from the compromised device, and reports it to remote command & control servers at multiple web locations such as this one.

Black-box testing would detect this malware by merely starting the app and monitoring the network activity of the phone, thereby capturing the attempt to connect to this suspicious web location.

White-box testing, on the other hand, would involve inspecting the source or binary code of the app, instead of observing the app's input-output behavior in the case of black-box testing. This inspection in turn would reveal this call in the code to connect to this suspicious web location.

Notice that both the approaches detect the same malicious behavior but they do so in fundamentally different ways. Of course, either of these approaches could fail to detect this malicious behavior if they are not careful enough; a combined approach

would reduce the chance of such failure.


CIS 547 - Software Analysis

The Automated Testing Problem

- Automated testing of programs is hard
- Probably impossible for entire systems
- Certainly impossible without specifications

Penn Engineering

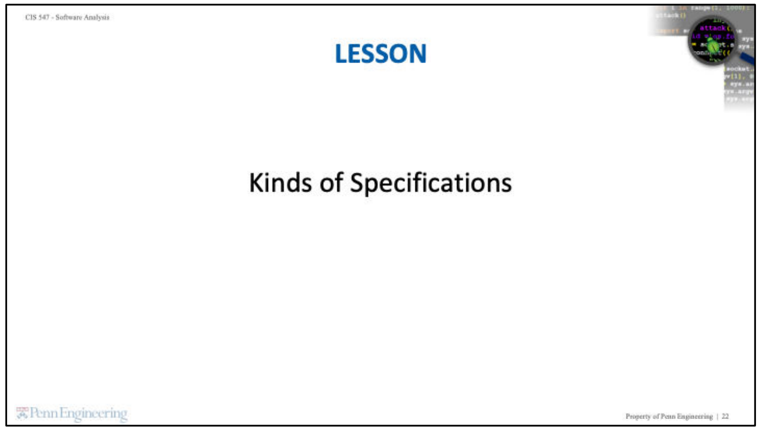
Property of Penn Engineering | 21

Despite the attractiveness of automating away all of our testing, there are several constraints that prevent us from making testing entirely automatic.

As we will see, testing is a hard enough problem even for a small piece of code. The number of pathways through a program increases exponentially with the number of branch-points in the program: just 30 if-else statements yield over one billion possible routes that need to be tested to verify that the program works in all conditions. And if a program has a loop, there could potentially be an infinite number of routes through the code, in which case it becomes impossible to test the code under all possible conditions.

Moving beyond the scope of a single file of code to an entire system, the problem of testing quickly becomes intractable. The best we can hope to do in many cases is separate code into small components, each of which is tested separately.

And if we don't have a specification for our program, then no testing can be done at all, let alone automated testing! So let's start by looking at how to define the specifications for a program.


CIS 547 - Software Analysis

LESSON

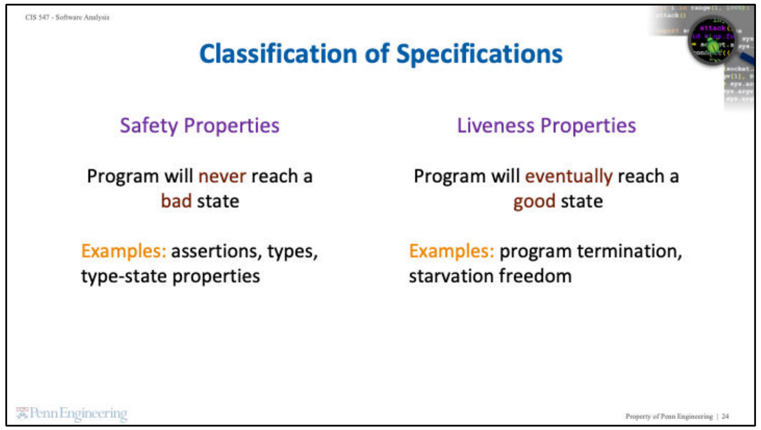
Kinds of Specifications

Penn Engineering

Property of Penn Engineering | 22

SEGMENT

Safety and Liveness



Classification of Specifications

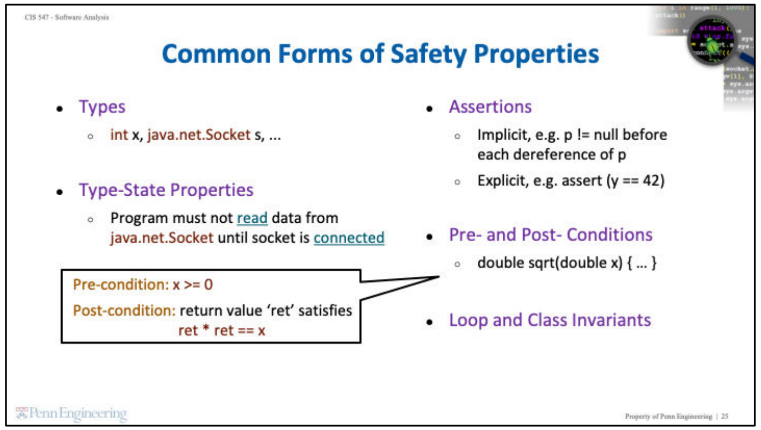
Safety Properties	Liveness Properties
Program will never reach a bad state	Program will eventually reach a good state
Examples: assertions, types, type-state properties	Examples: program termination, starvation freedom

We can classify specifications broadly into two major categories: safety properties and liveness properties.

Safety properties state that something bad will never happen, for example, an assertion violation. We will momentarily look at other forms of safety properties such as types and type-state properties.

In contrast, liveness properties state that something good will eventually happen. Examples of liveness properties include program termination and starvation freedom. Starvation refers to a problem in scheduling algorithms wherein a process is perpetually denied necessary resources to process its work.

Throughout this course, we will focus on safety properties, since they are more commonplace in practice.



Common Forms of Safety Properties

- Types
  - int x, java.net.Socket s, ...
- Type-State Properties
  - Program must not read data from java.net.Socket until socket is connected
- Assertions
  - Implicit, e.g. p != null before each dereference of p
  - Explicit, e.g. assert (y == 42)
- Pre- and Post- Conditions
  - double sqrt(double x) { ... }
- Loop and Class Invariants

Pre-condition: x >= 0

Post-condition: return value 'ret' satisfies ret * ret == x

Let's look at some common forms of safety properties.

The form that you might be most familiar with are types. Types are safety properties as they ensure the program will never have a runtime type error, for instance, preventing an operation expecting a certain kind of value from being used with values for which that operation does not make sense.

Type-state properties are another form of safety properties. They extend types to capture temporal properties of objects. For instance, the program must not read from an object of type java.net.Socket until it is connected.

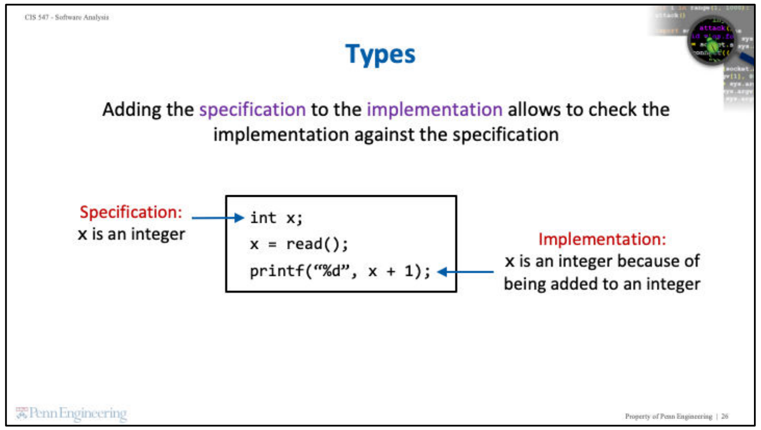
Assertions are a general form of safety properties that allow the programmer to specify that a certain predicate must always hold at a certain point in the program. Assertions can be implicit such as checking that a variable P is not null before dereferencing it each time in a managed language like Java. They can also be explicit, for example, when checking whether a variable Z equals 42 at a certain point in the program.

Pre and Post conditions further generalize assertions. Pre-conditions must be true before a statement or a method while post-conditions must be true after the statement or method is complete. For example, lets consider a square root function. Here, a plausible pre-condition could be ensuring that the input argument is non-negative. A plausible post-condition could be that the return value must indeed be equal to the

square root of the input.

Note that, while an assertion must hold unconditionally at a program point, a post-condition is required to hold at a program point such as a function exit only under the condition that the pre-condition holds at the function entry. It is in that sense that pre- and post- conditions generalize the notion of assertions.

Just as pre- and post-conditions specify properties of methods, loop invariants specify properties of loops, and class invariants specify properties of classes. Together, these forms of specifications suffice to test or even formally prove rich correctness properties of programs. Let’s look in more detail at each of these forms of specifications next, one at a time.



Types

Adding the specification to the implementation allows to check the implementation against the specification

Specification:
x is an integer

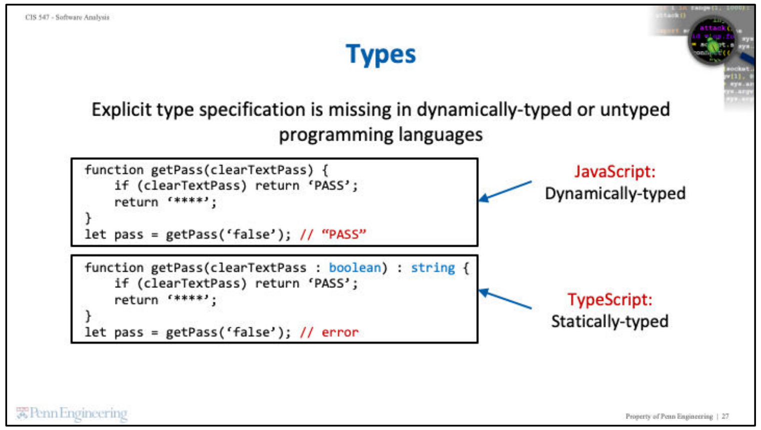
```
int x;
x = read();
printf("%d", x + 1);
```

Implementation:
x is an integer because of being added to an integer

Types are perhaps the most familiar kind of software specification. There are many benefits to types. Most simply, type annotations improve program readability. These annotations are built into the programming language. As a result, the program, documentation, and test cases all provide additional information to assist the reader in understanding the semantics.

However, we could get an equal improvement in readability by simply adding comments to describe the type of each variable. The real power of type systems lies in the fact that they are machine-checkable. The compiler can statically check the implementation against the specification and ensure they are consistent. Here, the specification is the type annotations and the implementation is the program. If they are not consistent, a type mismatch is reported to the programmer and the program does not compile. Conversely, if the program type-checks, then it is guaranteed to not have certain kinds of logic errors at runtime.

Lastly, as we observed earlier in this module, the specifications for a program often change over time. Types provide a mechanism for the specification to stay updated and continually checked by the compiler as the implementation evolves.



Types

Explicit type specification is missing in dynamically-typed or untyped programming languages

```
function getPass(clearTextPass) {
    if (clearTextPass) return 'PASS';
    return '****';
}
let pass = getPass('false'); // "PASS"
```

JavaScript:
Dynamically-typed

```
function getPass(clearTextPass : boolean) : string {
    if (clearTextPass) return 'PASS';
    return '****';
}
let pass = getPass('false'); // error
```

TypeScript:
Statically-typed

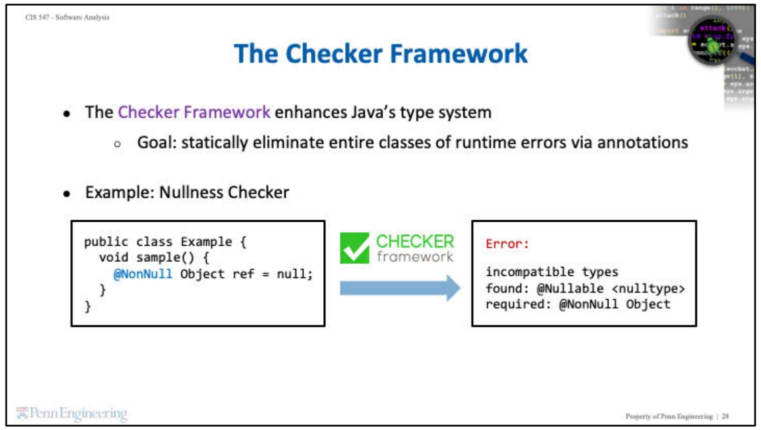
Although there are many advantages to using types, dynamic scripting languages, in which type annotations are not required, are becoming increasingly popular. For instance, in 2019, the top two languages on Github were Javascript and Python. Both of these languages lack a static type system.

Developers flock to these languages as they provide a way to quickly prototype applications. In these dynamic languages, developers don't need to write specifications. They can simply rapidly develop the implementation without paying attention to the specification.

For instance, consider the code fragment in Javascript. It inadvertently calls the getPass function with argument as string value 'false' instead of the boolean value false. The getPass function interprets the argument as the boolean value true, and returns the string value 'PASS' instead of '****'.

Now consider the code fragment in Typescript which is Microsoft's extension to Javascript with static types. In Typescript, the developer could specify the type of the argument clearTextPass as boolean, which would enable the Typescript compiler to flag the logic error before the program was deployed.

Since specifications are often tedious to write, later in this module, we will discuss a method to automatically infer specifications in the form of pre and post conditions, and invariants. In this way, programmers can get the best of both worlds through rapid development while ensuring some level of correctness.



The Checker Framework

- The Checker Framework enhances Java's type system
  - Goal: statically eliminate entire classes of runtime errors via annotations
- Example: Nullness Checker

```
public class Example {
  void sample() {
    @NonNull Object ref = null;
  }
}
```

CHECKER
framework

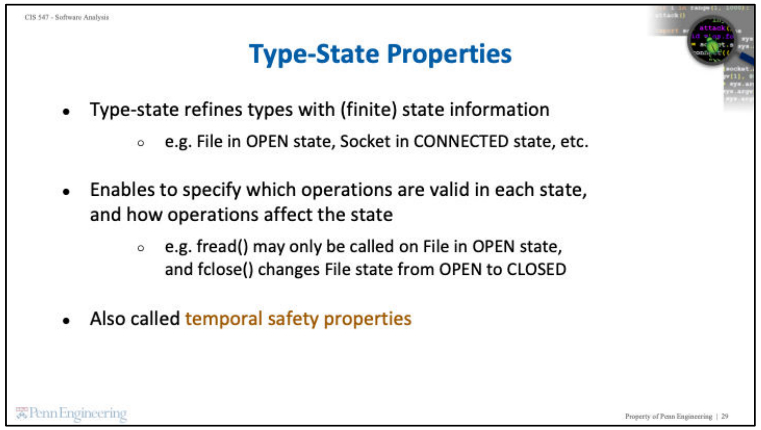
```
Error:

incompatible types
found: @Nullable <nulltype>
required: @NonNull Object
```

Types can go far beyond checking the correctness of simple logic errors. For instance, the Checker Framework is an extension to Java's type system that allows the developer to provide annotations to statically check and eliminate entire classes of runtime errors, such as null pointer dereferences.

More concretely, the Nullness checker employs two type annotations – @Nullable and @NonNull. The @NonNull annotation indicates that a variable will never be null. Conversely, the @Nullable annotation indicates that a variable may be null. As shown in this example, if null is assigned to a variable annotated NonNull, an "incompatible types" error is reported to the developer and the program does not compile.

In addition to the Nullness checker, the Checker Framework has type systems for taint checking, which checks that untrusted values do not affect sensitive computations, and checking GUI effects, such as whether UI methods are called from the current thread. The framework also provides a platform for developers to create their own checkers, which can be useful for checking idioms specific to individual codebases.

The Checker Framework is widely used in industry at large corporations such as Amazon, Uber, and Google. In fact, the Nullness checker is part of Google's standard build system.



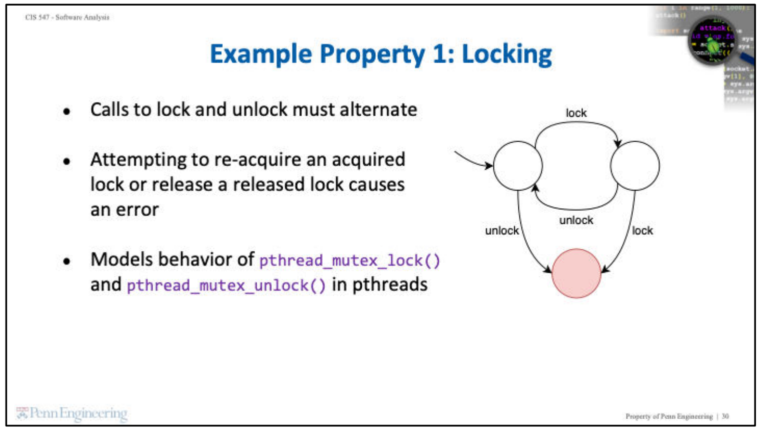
Type-State Properties

- Type-state refines types with (finite) state information
  - e.g. File in OPEN state, Socket in CONNECTED state, etc.
- Enables to specify which operations are valid in each state, and how operations affect the state
  - e.g. fread() may only be called on File in OPEN state, and fclose() changes File state from OPEN to CLOSED
- Also called temporal safety properties

A specialized but common extension to types is type-state. Indeed, the Checker Framework includes a type-state checker. As the name suggests, type-state properties associate a finite amount of state information with objects of a given type. For instance, the type of File objects may be associated with one of OPEN or CLOSED states, and the type of Socket objects may be associated with one of INITIALIZED, CONNECTED, or CLOSED states.

Such state information enables to specify which operations are valid on the object in each state, and how operations affect the state of the object. For example, that fread() may only be called on a File object in the OPEN state, and fclose() changes the File object's state from OPEN to CLOSED, thereby forbidding further calls to fread() on such a File object.

Type-state properties are also called temporal safety properties since they involve reasoning about safety as the state of objects evolves over time. Let's take a look at some example uses of type-state properties such as lock management, file management, and handling root privilege.



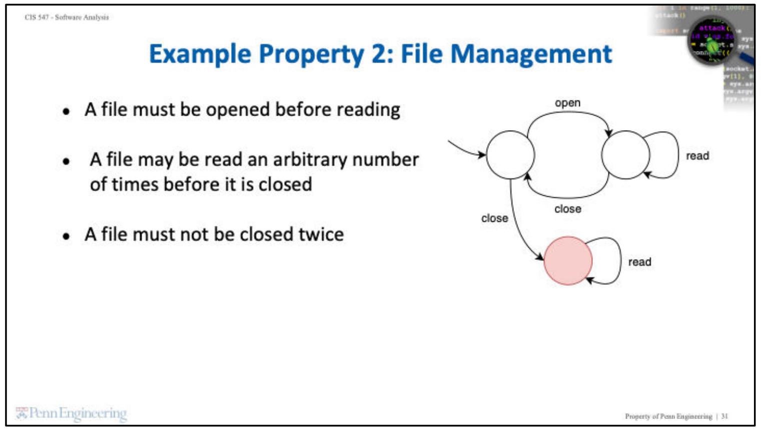
Example Property 1: Locking

- Calls to lock and unlock must alternate
- Attempting to re-acquire an acquired lock or release a released lock causes an error
- Models behavior of `pthread_mutex_lock()` and `pthread_mutex_unlock()` in pthreads

Consider the informal specification that a program must acquire and release locks in strict alternation.

Attempting to re-acquire an acquired lock, or release a released lock, causes an error. In fact, this is how the POSIX thread library's functions pthread-mutex-lock and pthread-mutex-unlock behave.

Here is a finite-state machine that formally specifies this type-state specification. The lock object is created in the unlocked state [mark x]. Performing the lock operation in this state transitions the object to the locked state [mark x], but performing the unlock operation leads to the error state [mark x]. Likewise, performing the unlock operation in the locked state transitions the object back to the initial unlocked state, but performing the lock operation leads to the error state.



CIS 547 - Software Analysis

Example Property 2: File Management

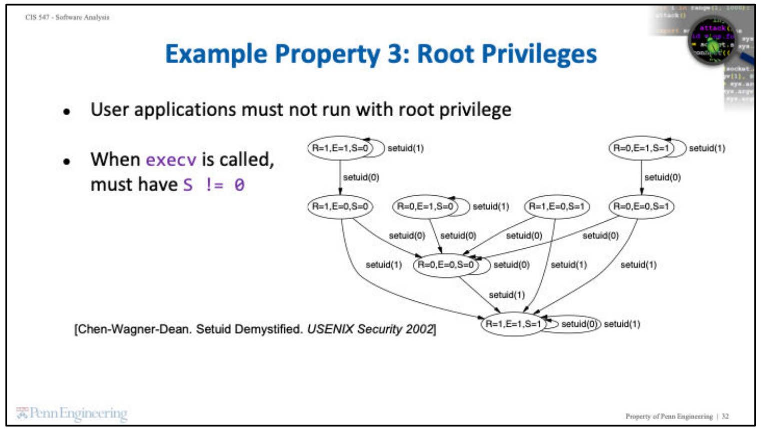
- A file must be opened before reading
- A file may be read an arbitrary number of times before it is closed
- A file must not be closed twice

PennEngineering

Property of Penn Engineering | 31

Similarly, file management imposes temporal properties of the form:
- A file must be opened before reading,
- A file may be read an arbitrary number of times before it is closed; and
- A file must not be closed twice.

Take a moment to convince yourself that this type-state specification correctly captures these temporal properties.



CIS 547 - Software Analysis

Example Property 3: Root Privileges

- User applications must not run with root privilege
- When `execv` is called, must have `S != 0`

[Chen-Wagner-Dean. Setuid Demystified. *USENIX Security 2002*]

PennEngineering

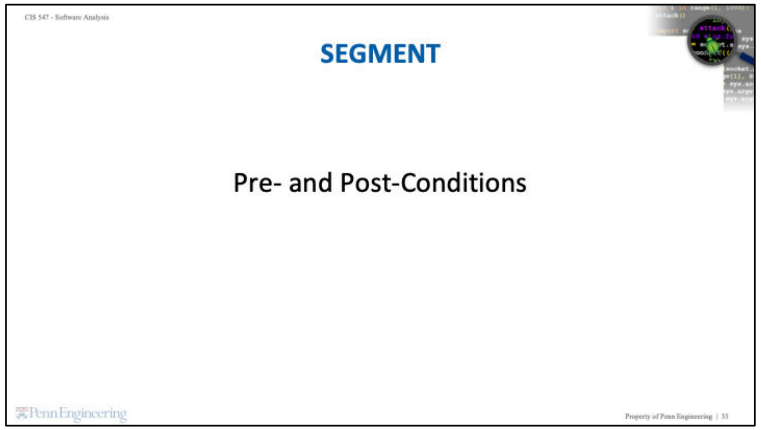
Property of Penn Engineering | 32

We just saw two simple examples expressible using type-state properties. However, type-state properties are expressive enough to specify highly complex, sophisticated safety properties.

For instance, by checking type-state, we can even avoid security vulnerabilities that arise due to an invalid sequence of operations. Suppose we would like to specify that a user application must never run with root privilege on a Linux system.

This finite state automaton describes the behavior of the setuid system call in Linux that changes the privilege of a process. Each node corresponds to a different "privilege state" of the process. A privilege state is defined by 3 bits <R, E, S>, which denote various user IDs of the process: the real user ID, the effective user ID, and the saved user ID. The transitions in the automaton show how the process's privilege changes by calling the system call setuid with argument 0 or 1 to set the R, E or S bit changed in the transition.

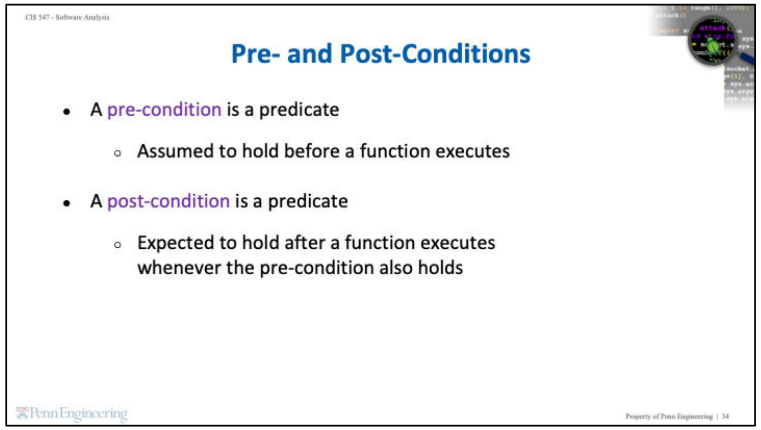
Back to the property we are trying to enforce: a user application must never run with root privilege. More concretely, the user application must never invoke system call execv in a privilege state where S is equal to zero, indicating root privilege [mark the four such states]. When expressed in terms of type-state, even this complicated property can be succinctly conveyed.


CIS 547 - Software Analysis

SEGMENT

Pre- and Post-Conditions

Property of Penn Engineering | 33


CIS 547 - Software Analysis

Pre- and Post-Conditions

- A pre-condition is a predicate
  - Assumed to hold before a function executes
- A post-condition is a predicate
  - Expected to hold after a function executes whenever the pre-condition also holds

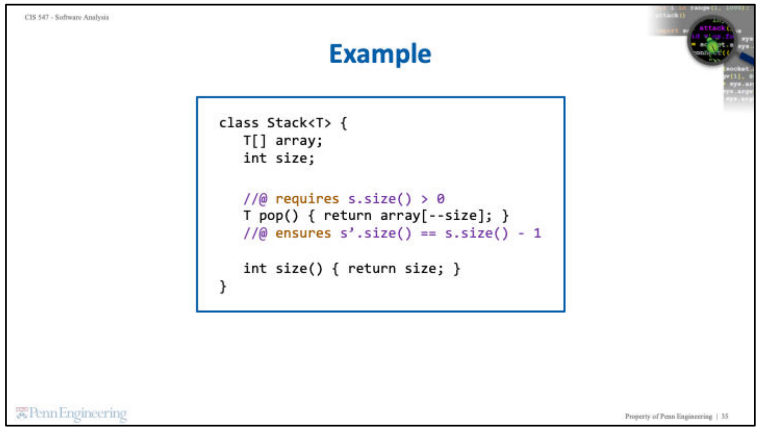
Property of Penn Engineering | 34

Now we will look at a very general specification mechanism called pre- and post-conditions.

A pre-condition is a predicate that is assumed to hold before the execution of some function, and a post-condition is a predicate that is expected to hold after the execution of a function whenever the pre-condition holds.

One use of pre-conditions is to ensure that a function does not operate in an undefined way on inputs it was not designed to handle.

Similarly, a use of post-conditions is to ensure that a function's output matches its specification: for example, a function that squares a real number should not output a negative number.


CIS 547 - Software Analysis

Example

```
class Stack<T> {
   T[] array;
   int size;

   //@ requires s.size() > 0
   T pop() { return array[--size]; }
   //@ ensures s'.size() == s.size() - 1

   int size() { return size; }
}
```

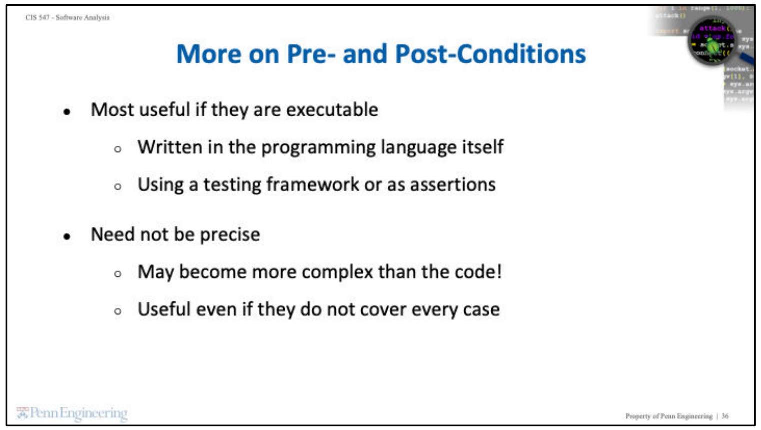
Property of Penn Engineering | 35

In this example code in Java, we see a pre-condition and a post-condition for the function pop() in a class implementing a stack data structure. For now, we have written these specifications in comments, but we will soon see examples that support writing such specifications as machine-checkable annotations in the program itself.

An assumption is made going into the pop() function that the stack has at least one element (otherwise pop() is undefined; indeed, in Java, trying to execute this pop() method with an empty Stack object would throw an exception upon trying to access the element at index -1 of array).

The post-condition asserts that the size of the Stack object after the pop (indicated by s'.size()) should be exactly one smaller than it was beforehand (indicated by s.size()). This post-condition also documents a change in the Stack object's state that outside code can rely upon whenever it calls the pop() function.

Remember that pre- and post-conditions often only partially capture the specifications of a function. For instance, the above post-condition says nothing about remaining N - 1 elements on stack (for example, whether they are in the same order as before, or even whether they are in fact the same objects as before!). Implicitly, this lack of mention is interpreted as saying that "nothing else changes" about the state of the program. These sorts of implied conditions are called frame conditions.


CIS 547 - Software Analysis

More on Pre- and Post-Conditions

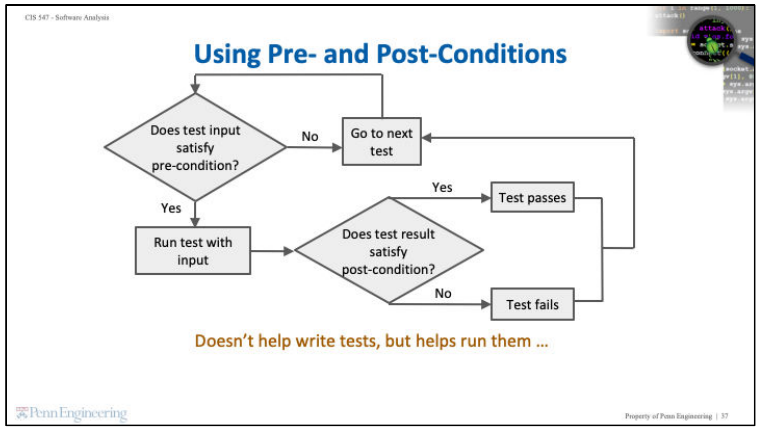
- Most useful if they are executable
  - Written in the programming language itself
  - Using a testing framework or as assertions
- Need not be precise
  - May become more complex than the code!
  - Useful even if they do not cover every case

Property of Penn Engineering | 36

As we noted earlier, pre- and post-conditions are most useful to developers and testers if they are executable. Indeed, we can write these specifications into the program itself in the same language, either using a testing framework such as JUnit or built-in functionality such as an assert statement.

Pre- and post-conditions also need not be precise statements of the required conditions on the input and output of each function. Recall earlier that the problem of testing quickly becomes intractable because of the number of possible routes that execution flow through a program can contain.

Similarly, pre- and post-conditions that perfectly describe the input and output requirements for a function may be extremely complex, perhaps even more than the code it is specifying. As such, these conditions often only check certain facets of the input and output, trading precision for tractability.



Using Pre- and Post-Conditions

Does test input satisfy pre-condition?
No
Go to next test
Yes
Run test with input
Does test result satisfy post-condition?
Yes
Test passes
No
Test fails

Doesn't help write tests, but helps run them …

The process of using pre- and post-conditions in testing is straightforward.

To perform a test, we first check that the test input satisfies the pre-condition. [Left Diamond appears]

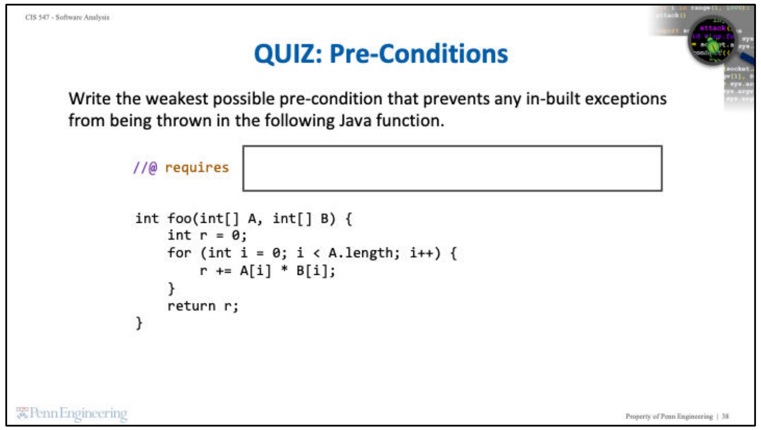
If it does not, then we skip the test and go to the next one. ["No" arrow, "Go to next test" box, and rectangular arrow out of that box appears]

If it does, then run the test with that input, ["Yes" arrow and "Run test with input" box appears] and then check that the output of the test satisfies the post-condition. [Arrow out of "Run test with input" box and Right Diamond appears]

If it does, then the test passes ["Yes" arrow and "Test passes" box appears]. Otherwise, the test fails ["No" arrow and "Test fails" box appears]. In both of these cases, we then proceed to the next test. [All the remaining arrows appear]

In this way, we check that any input satisfying the pre-condition yields a result satisfying the post-condition.

While this framework doesn't help us generate the tests, it does help significantly with automating testing runs.



QUIZ: Pre-Conditions

Write the weakest possible pre-condition that prevents any in-built exceptions from being thrown in the following Java function.

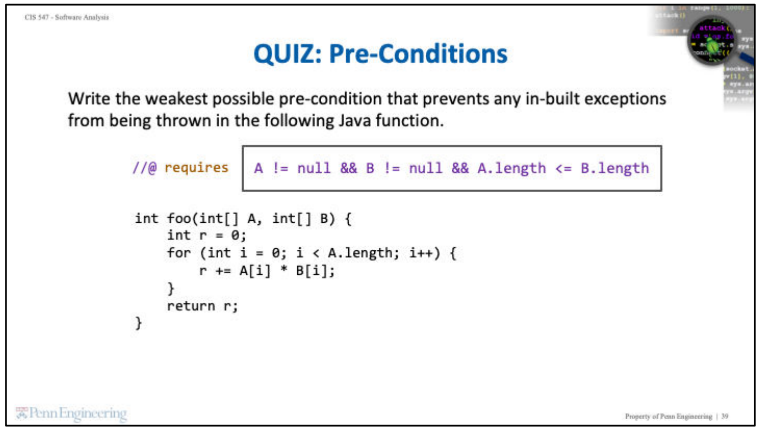
```
//@ requires

int foo(int[] A, int[] B) {
    int r = 0;
    for (int i = 0; i < A.length; i++) {
        r += A[i] * B[i];
    }
    return r;
}
```

{QUIZ SLIDE}

To check your understanding of pre-conditions, let's take a look at this Java function.

For this quiz, write the weakest possible pre-condition that prevents any built-in exceptions (such as NullPointerException and ArrayIndexOutOfBoundsException) from being thrown during any execution of the function.

Write your answer as a Java boolean expression in the text box provided. (No need to write an assert statement here: just a boolean expression.)


CIS 547 - Software Analysis

QUIZ: Pre-Conditions

Write the weakest possible pre-condition that prevents any in-built exceptions from being thrown in the following Java function.

```
//@ requires  A != null && B != null && A.length <= B.length

int foo(int[] A, int[] B) {
    int r = 0;
    for (int i = 0; i < A.length; i++) {
        r += A[i] * B[i];
    }
    return r;
}
```

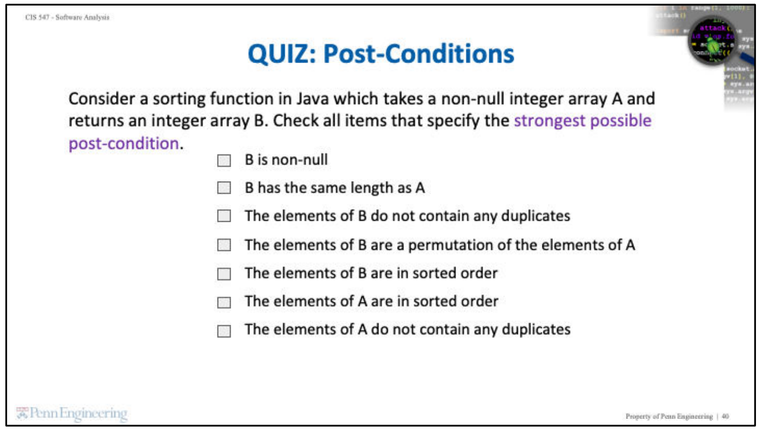
Penn Engineering   Property of Penn Engineering | 39

{SOLUTION SLIDE}

For this function, our pre-condition needs to assert something about the input arrays A and B. Because we dereference both A and B, it is essential that they not point to a null array. So A != null && B != null should be included in the pre-condition.

Additionally, observe that for every cell we access of the array A, we also access the corresponding cell of the array B. Since this happens for every cell in A, we must also require the length of B to be at least the length of A or else we will run into an out-of-bounds exception. So we add on A.length <= B.length to our pre-condition.

These are the weakest requirements we need in order for this function to execute correctly.


CIS 547 - Software Analysis

QUIZ: Post-Conditions

Consider a sorting function in Java which takes a non-null integer array A and returns an integer array B. Check all items that specify the strongest possible post-condition.

- ☐ B is non-null
- ☐ B has the same length as A
- ☐ The elements of B do not contain any duplicates
- ☐ The elements of B are a permutation of the elements of A
- ☐ The elements of B are in sorted order
- ☐ The elements of A are in sorted order
- ☐ The elements of A do not contain any duplicates

Penn Engineering   Property of Penn Engineering | 40

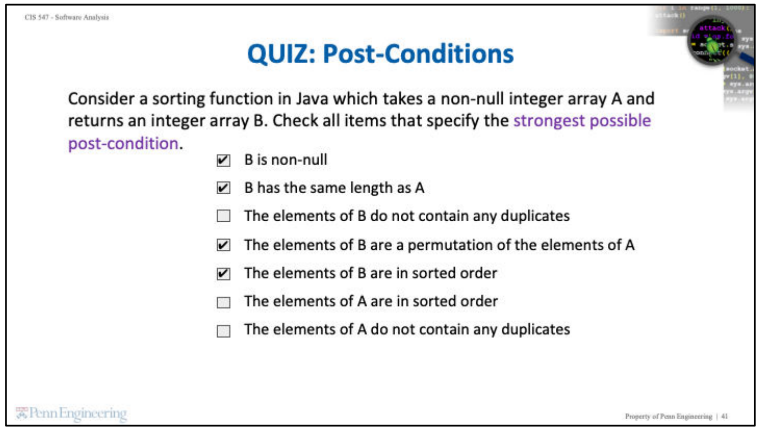
{QUIZ SLIDE}

Now, to check your understanding of post-conditions, consider the following quiz.

Given a sorting function in Java which takes a non-null integer array A, puts them in sorted order in an integer array B, and then returns B, which of the following statements must be true in order to form the strongest possible post-condition for the function that does not improperly reject a correctly sorted array?

- B is non-null
- B has the same length as A
- The elements of B do not contain any duplicates
- The elements of B are a permutation of the elements of A
- The elements of B are in sorted order
- The elements of A are in sorted order
- The elements of A do not contain any duplicates

Check all the statements that apply.



QUIZ: Post-Conditions

Consider a sorting function in Java which takes a non-null integer array A and returns an integer array B. Check all items that specify the strongest possible post-condition.

- [✓] B is non-null
- [✓] B has the same length as A
- [ ] The elements of B do not contain any duplicates
- [✓] The elements of B are a permutation of the elements of A
- [✓] The elements of B are in sorted order
- [ ] The elements of A are in sorted order
- [ ] The elements of A do not contain any duplicates

{SOLUTION SLIDE}

Must B be non-null? Yes (since we assumed the sorting function was passed a non-null array in the first place).

Must B have the same length as A? Yes, sorting the elements of an array does not change the number of elements in the array.

Must B contain no duplicates? No, this is not required of a sorted array.

Must B be a permutation of A? Yes, B should consist of the same elements as A, just in a (possibly) different order.

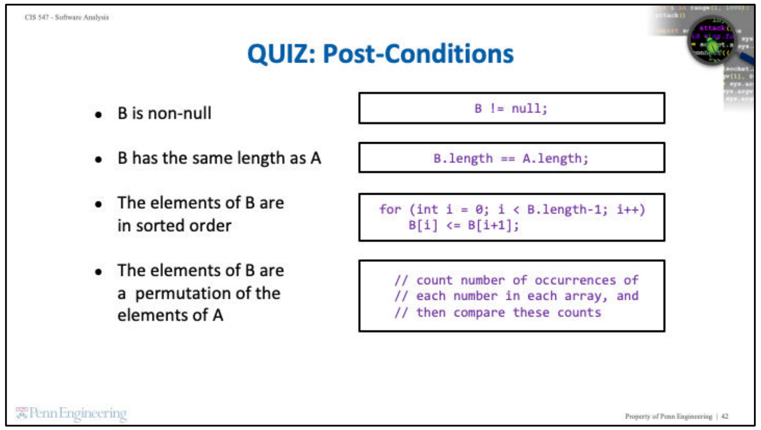
Must B be in sorted order? Yes, of course (or else the function didn't do its job!).

Must A be in sorted order? No, this needn't be required, since we are not sorting A in place.

Must A contain no duplicates? No, this isn't required either, since sorting functions should be able to handle duplicate elements.

In fact, these four conditions make up the entirety of the specification for the output of a sorting function: there is no stronger condition that can be required without rejecting

a properly sorted array.


CIS 547 - Software Analysis

QUIZ: Post-Conditions

- B is non-null
```
B != null;
```
- B has the same length as A
```
B.length == A.length;
```
- The elements of B are in sorted order
```
for (int i = 0; i < B.length-1; i++)
    B[i] <= B[i+1];
```
- The elements of B are a permutation of the elements of A
```
// count number of occurrences of
// each number in each array, and
// then compare these counts
```

Penn Engineering

Property of Penn Engineering | 42

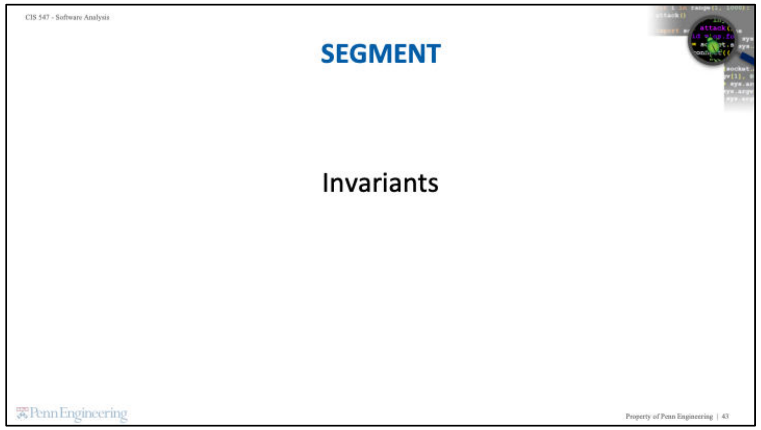
What would the sorting post-condition look like if we wrote it in executable code?

The first part of the post-condition is that B is non-null. This is easy enough to implement by adding an assertion that B does not equal the null pointer.

The second part of the post-condition, that B and A have the same length, is similarly easy to implement, by adding an assertion that B.length == A.length.

The third part of the post-condition, that the elements of B are in sorted order, is a bit more complex. Here, we would need to check that B[i] <= B[i+1] for all i between 0 and B.length - 2 (or >=, if we were doing a descending sort).

The last part of the post-condition is the most complex to implement. One strategy might be to count the number of occurrences of each element in each array and then check that these counts are the same for each array. We'll leave it as an exercise for you to implement this yourself if you like.

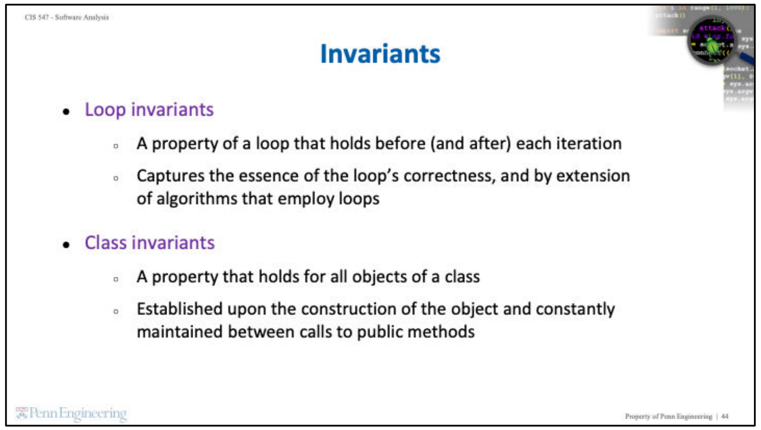

CIS 547 - Software Analysis

SEGMENT

Invariants

Penn Engineering

Property of Penn Engineering | 43


CIS 547 - Software Analysis

Invariants

- Loop invariants
  - A property of a loop that holds before (and after) each iteration
  - Captures the essence of the loop's correctness, and by extension of algorithms that employ loops
- Class invariants
  - A property that holds for all objects of a class
  - Established upon the construction of the object and constantly maintained between calls to public methods

Penn Engineering

Property of Penn Engineering | 44

A program invariant is a property that is true during the execution of the program or during some portion of it. With each invariant comes the benefit that the program can assume that the property is always true, and the responsibility that the program must never cause the invariant to become false at any point at which it is expected to be true.

There are two kinds of invariants that are particularly useful: loop invariants and class invariants.

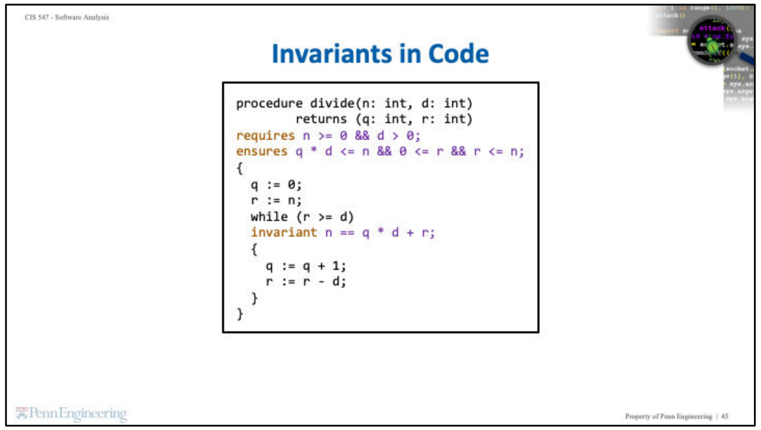
A loop invariant is a property that must hold immediately before and after each loop iteration. Note that it is possible for a loop invariant to not hold temporarily during the execution of the loop body itself.

As you might have noticed, from the definition of the loop invariant, we can have many of them–in fact infinitely many---for a single loop. For instance, "i == i" is a legitimate invariant by definition. However, one must choose a loop invariant that is strong enough to prove a non-trivial property about the loop, and the program at large. It basically needs to capture the essence of the loop's correctness.

Just as a loop invariant is a property that is expected to hold of all iterations of a loop, a class invariant is a property that is expected to hold for all instances of a class. It is established upon the construction of the object, and it must be maintained between calls to the public methods of the class. Note that, just as a loop invariant might not

hold temporarily during the execution of the loop body, a class invariant might not hold during the execution of a class method. But it must hold before and after the execution of each such public method.

Besides being useful for program understanding, invariants can also be machine-checked, by using them in conjunction with testing and formal verification – as we shall see next.


CIS 547 - Software Analysis

Invariants in Code

```
procedure divide(n: int, d: int)
        returns (q: int, r: int)
requires n >= 0 && d > 0;
ensures q * d <= n && 0 <= r && r <= n;
{
  q := 0;
  r := n;
  while (r >= d)
  invariant n == q * d + r;
  {
    q := q + 1;
    r := r - d;
  }
}
```

Penn Engineering

Property of Penn Engineering | 45

You might be asking, how can one add these invariants to code?

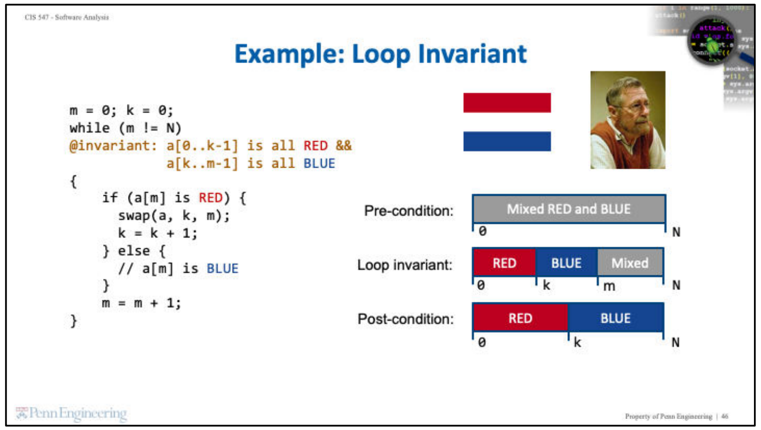
We could add invariants as plain text comments. Unfortunately, since they aren't machine checkable, that won't be very helpful.

In this program, the invariant is written in an intermediate verification language called Boogie from Microsoft. What you see here is an implementation of integer division.

With Boogie, these invariants now reside in the code. In a similar way to types and type-state specifications, these can be checked with automatic theorem provers, such as Microsoft's Z3. In this example, the loop invariant captures the essence of the correctness of the loop, and is essential to verifying that the implementation satisfies the specification of pre- and post conditions for the divide procedure, shown here [point to requires and ensures lines]

Lastly, recall that earlier in this module, we observed how developer resources are limited, and formal verification can be quite expensive. Verifying complex pre- and post- conditions can require significant annotation effort from the developer in the form of such invariants. As an alternative, the developer may choose to abandon the hope of formal verification and fall back to testing. An automated testing tool such as a fuzzer could be used to continually test these specifications by converting them into assertions.

Next, let us delve a bit deeper into loop invariants and class invariants by way of examples.



Example: Loop Invariant

```
m = 0; k = 0;
while (m != N)
@invariant: a[0..k-1] is all RED &&
            a[k..m-1] is all BLUE
{
    if (a[m] is RED) {
      swap(a, k, m);
      k = k + 1;
    } else {
      // a[m] is BLUE
    }
    m = m + 1;
}
```

Pre-condition:
Mixed RED and BLUE
0
N
Loop invariant:
RED
BLUE
Mixed
0
k
m
N
Post-condition:
RED
BLUE
0
k
N

The Dutch national flag (DNF) problem is one of the most popular programming problems proposed by Edsger Dijkstra--one of the founding fathers of the field of program correctness and verification.

The flag of the Netherlands consists of three colors: white, red, and blue. The task is to randomly arrange balls of white, red, and blue such that balls of the same color are placed together. For simplicity we will only assume two colors, Red and Blue. So it becomes the 2-color DNF problem.

First of all, the pre-condition does not restrict the order of balls, and is basically an unordered array of red and blue balls with length N. The desired post-condition is that all the elements from 0 to K-1 must be red, and all the elements from K to N-1 must be blue.

Here's a program that solves this problem. It assumes the existence of a static method called swap whereby the call swap(A, K, M) swaps the elements of array A at indices K and M.

Given this loop invariant, it is possible for an automated theorem prover to mechanically prove that this program indeed satisfies the specification comprising this pre- and post- condition. The reasoning that the theorem prover performs involves performing three simple checks. Let's go over these three checks one at a time.

The first check involves showing that the pre-condition at the entry of the program implies the loop invariant at the entry of the loop. Indeed, we can see this by plugging in k = 0 and m = 0 into the loop invariant: the loop invariant trivially holds in this case. intuitively, this check proves that if the program begins executing in any state that satisfies this pre-condition, it will arrive at the entry of the loop in a state that satisfies the loop invariant.

The second check, called the inductive step, involves showing that if the loop invariant holds at the start of the loop body in some iteration, it will also hold at the end of the loop body in that iteration. This is usually the trickiest of the three checks. So let's go through it one step at a time. There are two cases depending on whether a[m] is RED or BLUE.

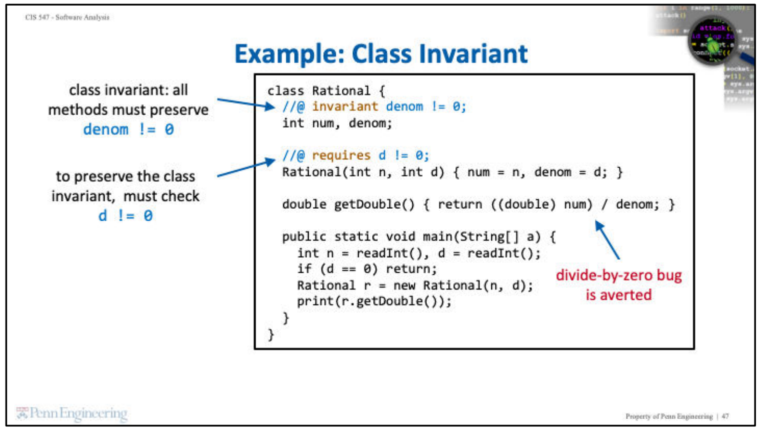
The latter case is easier so let's look at it first. If a[m] is BLUE, then the contents of array A and the integer variable k are unmodified by the loop body, and only the integer variable m is incremented. Since we assumed at the start of the loop body that all elements of the array from 0 thru k-1 were RED, and since neither the array's contents nor k changed, this part of the invariant still holds at the end of the loop body.

To prove that the other part of the invariant still holds, we assumed at the start of the loop body that all elements of the array from k thru m-1 were BLUE, and we just established in the current iteration that the element at index m is BLUE. So, even though m is incremented by 1, we reestablish this part of the invariant that all elements of the array from k thru m-1, using the incremented value of m, are all BLUE.

We are not yet finished with this check: we have yet to reason about the case where a[m] is RED. In this case, not only is m incremented, but k is also incremented, and the elements of the array A at indices k and m are swapped. It is again easy although somewhat tedious to see that, assuming both parts of the invariant hold at the start of the loop body, they will hold at the end of the loop body. The only potential concerns are that, at the end of the loop body, the element of the array at index k-1 might be BLUE instead of RED, thereby violating the first part of the invariant, or the element of the array at index m-1 might be RED instead of BLUE, thereby violating the second part of the invariant. But the swap procedure precisely takes care of both these concerns.

The third and final check involves showing that the loop invariant at the exit of the loop implies the post-condition at the exit of the program. Indeed, we can see this by plugging in m == N, the negation of the loop condition, into the invariant.

In summary, loop invariants are crucial to formally proving the pre- and post conditions of programs containing loops such as this one. Writing loop invariants is useful even if one does not go all the way to verify such correctness properties. For instance, you can simply add an assertion to check that the loop invariant holds on every iteration. Such assertions can help you to quickly discover any problems with your understanding of why the code works and catching errors when implementing a loop.


CIS 547 - Software Analysis
class invariant: all methods must preserve denom != 0
to preserve the class invariant, must check d != 0
divide-by-zero bug is averted
Penn Engineering
Property of Penn Engineering | 47

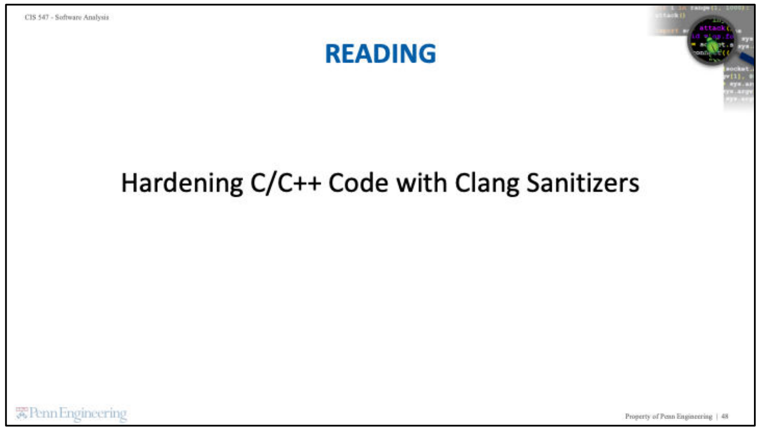
Example: Class Invariant

```
class Rational {
  //@ invariant denom != 0;
  int num, denom;

  //@ requires d != 0;
  Rational(int n, int d) { num = n, denom = d; }

  double getDouble() { return ((double) num) / denom; }

  public static void main(String[] a) {
    int n = readInt(), d = readInt();
    if (d == 0) return;
    Rational r = new Rational(n, d);
    print(r.getDouble());
  }
}
```

Next, let’s take a look at an example of a class invariant, and how they can help to formally prove the absence of a division-by-zero error.

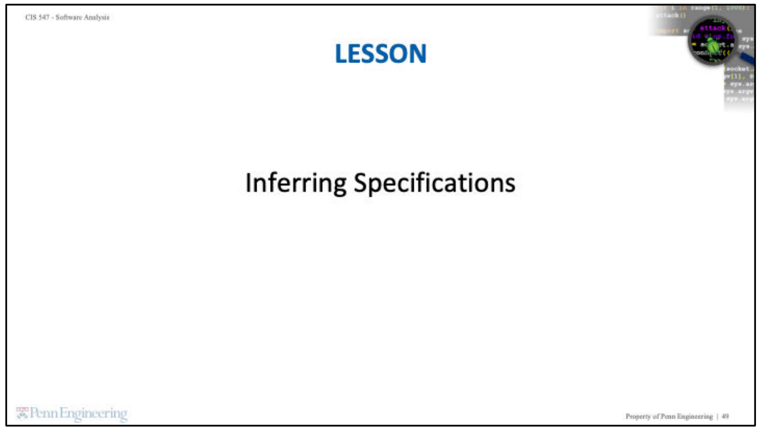
Here we see a segment of Java code, comprising a class, two integer fields, a constructor, and a method. The class is an implementation of rational numbers – numbers that can be expressed as the fraction of two integers, a numerator and a denominator. So an invariant for this class is that the denominator must be non-zero. The invariant must be established by the constructor and preserved by the getDouble() method. For this purpose, the constructor is required to have “d” not equal to zero, because d is the value that is assigned to the denominator. So we must add a pre-condition to the constructor. Indeed, this pre-condition is satisfied at the only call to the constructor, from the main method, since the call to the constructor is preceded by a check that d is non-zero [underline “if (d==0) return”].

Continuing this style of reasoning, we can assume that this class invariant, that the denominator is non-zero, holds on entry to the getDouble() method, and prove that it is preserved on exit of the method. In that step, we also end up proving that the getDouble method will not encounter a divide-by-zero error during its execution.



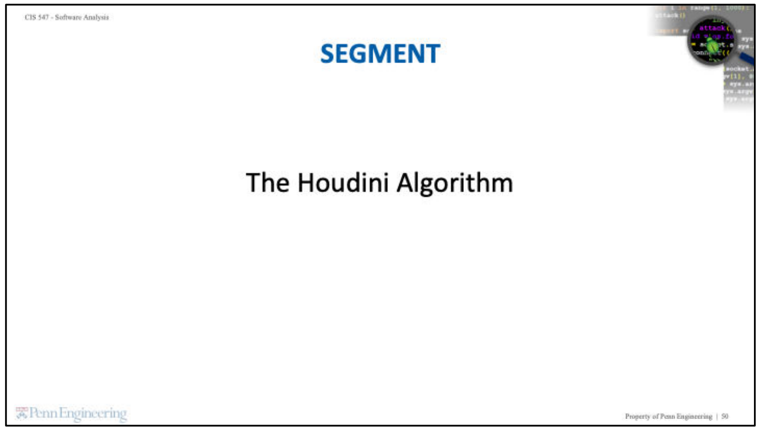
READING

Hardening C/C++ Code with Clang Sanitizers



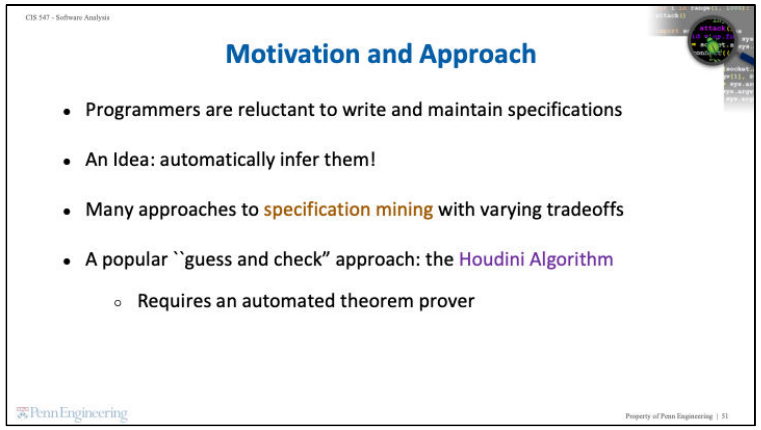
LESSON

Inferring Specifications



SEGMENT

The Houdini Algorithm



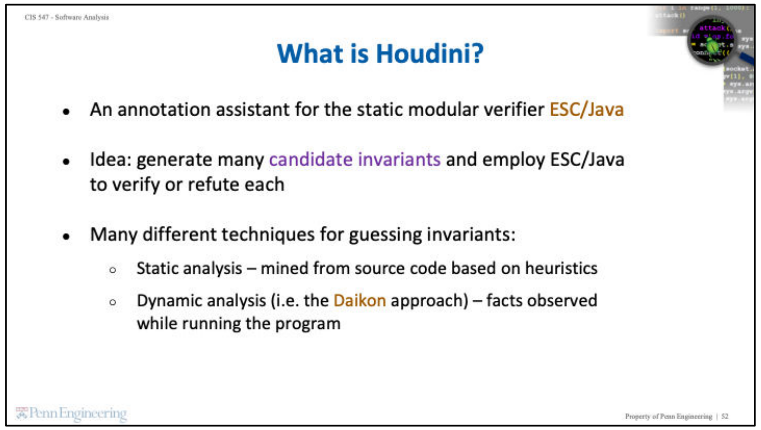
Motivation and Approach

- Programmers are reluctant to write and maintain specifications
- An Idea: automatically infer them!
- Many approaches to specification mining with varying tradeoffs
- A popular ``guess and check" approach: the Houdini Algorithm
  - Requires an automated theorem prover

We have thus far seen that specifications are very useful for improving program reliability. However, they are difficult for developers to write. Additionally, in a quickly changing codebase it would be burdensome for a developer to continuously modify the specification annotations.

An enticing idea to lower the burden on developers is to automatically infer them. There are many approaches to mine specifications from programs, with varying tradeoffs.

As a case study, we will focus on a popular guess-and-check approach to automatically infer sound pre- and post- conditions and loop and class invariants -- called the Houdini Algorithm. It relies on an automated theorem prover in order to provide the soundness guarantee.


CIS 547 - Software Analysis

What is Houdini?

- An annotation assistant for the static modular verifier ESC/Java
- Idea: generate many candidate invariants and employ ESC/Java to verify or refute each
- Many different techniques for guessing invariants:
  - Static analysis – mined from source code based on heuristics
  - Dynamic analysis (i.e. the Daikon approach) – facts observed while running the program

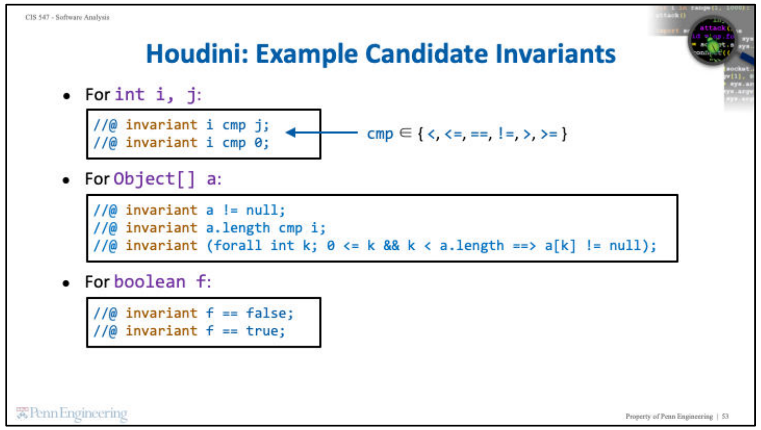
Penn Engineering

Property of Penn Engineering | 52

Houdini was originally developed as an annotation assistant for the static modular verifier ESC/Java, short for Extended Static Checker for Java programs. The algorithm has since found its way into many applications in software analysis.

The idea behind Houdini is remarkably simple: it uses a guess-and-check approach by generating many candidate invariants and checking them using ESC/Java itself. In the context of Houdini, we use the term invariant more broadly, encompassing not only loop and class invariants, but also pre and post conditions. The candidate invariants don't have to be correct. ESC/Java uses an automated theorem prover to check them in a modular fashion, iteratively discarding those that it fails to prove are valid. Since ESC/Java is sound, any remaining candidate invariants are guaranteed to be valid.

The candidate invariants can be guessed from a variety of different sources: for instance, they can be obtained using static analysis, mined from the program's source code based on heuristics, or from dynamic analysis, in the form of facts observed while running the program. This is precisely the function of the Daikon tool for inferring likely program invariants.


CIS 547 - Software Analysis

Houdini: Example Candidate Invariants

- For `int i, j`:

```
//@ invariant i cmp j;
//@ invariant i cmp 0;
```

cmp ∈ { <, <=, ==, !=, >, >= }

- For `Object[] a`:

```
//@ invariant a != null;
//@ invariant a.length cmp i;
//@ invariant (forall int k; 0 <= k && k < a.length ==> a[k] != null);
```

- For `boolean f`:

```
//@ invariant f == false;
//@ invariant f == true;
```

Penn Engineering

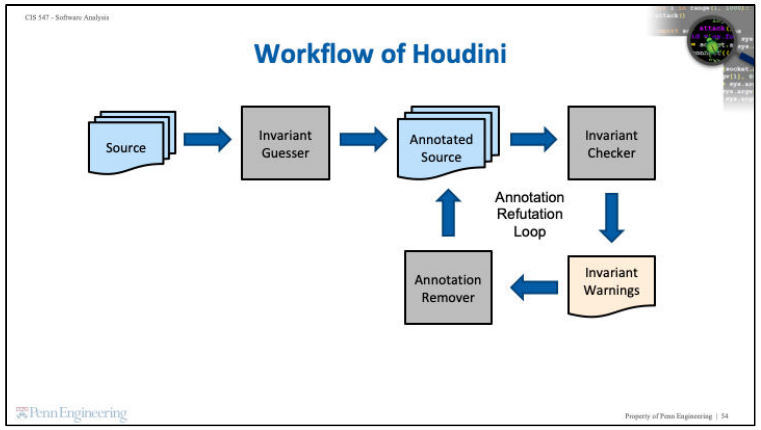
Property of Penn Engineering | 53

Let's look at some example candidate invariants for a few different types of program variables.

For integer variables, invariants might look like binary comparisons against other variables or constants.

For an array object, possible invariants include non-nullness of the object, or comparing the length of the array with other integer variables in scope. An example of a more complicated invariant is the property that all the elements of the array are non-null.

Lastly, for a Boolean variable, some possible invariants are ensuring the value of the variable is false, or conversely, true.


CIS 547 - Software Analysis

Workflow of Houdini

Penn Engineering

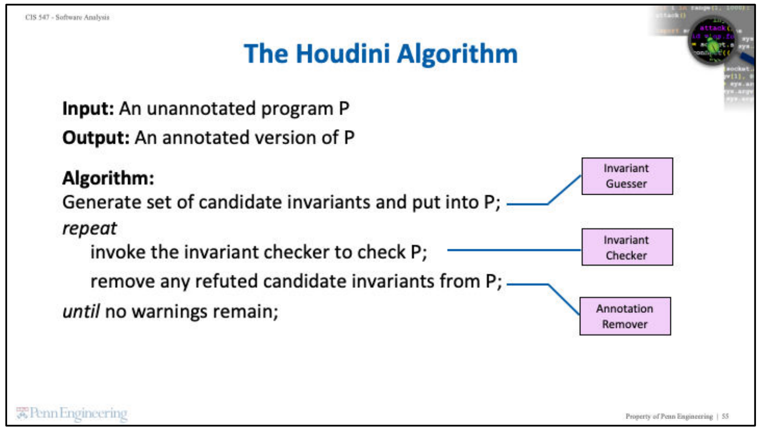
Property of Penn Engineering | 54

Let’s take a look at the workflow of Houdini.

Given the source code of the program which is not annotated with invariants, an invariant guesser generates as many candidate invariants as possible, using static analysis, dynamic analysis, or even programmer provided ones. The source code is annotated with the generated invariants, which is then checked with an invariant checker.

The invariant checker produces warnings for the invariants that it cannot prove to be valid. This could either be because the invariants are indeed invalid – there exists an execution in which they do not hold – or because the checker is incomplete – it just isn’t powerful enough to prove the invariants to be valid. Such invariants that result in warnings are removed, and the process is repeated until no new warnings are generated.

This loop is guaranteed to terminate since we begin with a finite set of candidate invariants and since we only shrink the set at each step. The remaining invariants are guaranteed to be valid since the checker is sound. However, they may not necessarily suffice to verify a property of interest, such as the absence of a divide-by-zero error in the program.


CIS 547 - Software Analysis

The Houdini Algorithm

Input: An unannotated program P

Output: An annotated version of P

Algorithm:

```
Generate set of candidate invariants and put into P;
repeat
    invoke the invariant checker to check P;
    remove any refuted candidate invariants from P;
until no warnings remain;
```

Invariant Guesser
Invariant Checker
Annotation Remover

Penn Engineering

Property of Penn Engineering | 55

To make things more concrete let's take a look at the steps of the Houdini algorithm in the form of pseudo-code.

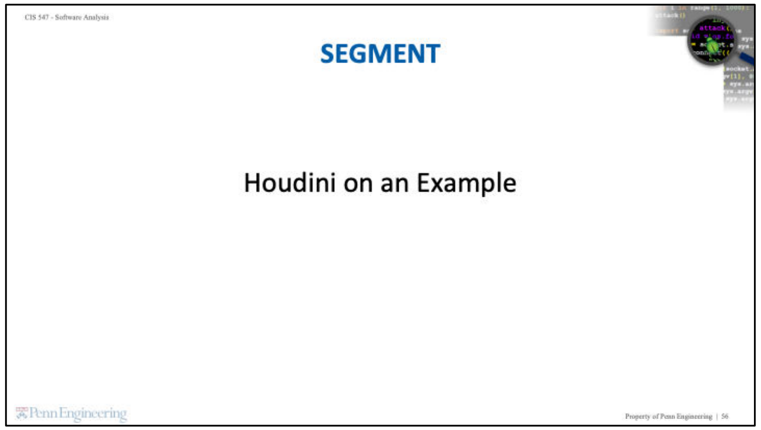
Again, keep in mind that the input is an unannotated version of the program, and the final output is an annotated version of the program.

The algorithm first generates a set of candidate invariants and inserts them into the program.

Next, the invariant checker is called on the annotated program, and warnings are inspected.

The invariants that cause warnings are removed and the new annotated version is fed to the invariant checker again.

This loop is continued until no further warnings remain.

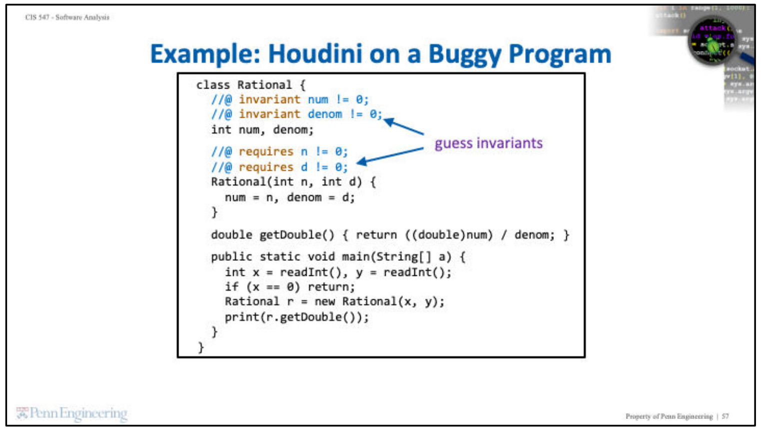


CIS 547 - Software Analysis

SEGMENT

Houdini on an Example

Property of Penn Engineering | 56



CIS 547 - Software Analysis

Example: Houdini on a Buggy Program

```
class Rational {
  //@ invariant num != 0;
  //@ invariant denom != 0;
  int num, denom;
  //@ requires n != 0;
  //@ requires d != 0;
  Rational(int n, int d) {
    num = n, denom = d;
  }
  double getDouble() { return ((double)num) / denom; }
  public static void main(String[] a) {
    int x = readInt(), y = readInt();
    if (x == 0) return;
    Rational r = new Rational(x, y);
    print(r.getDouble());
  }
}
```

guess invariants

Property of Penn Engineering | 57

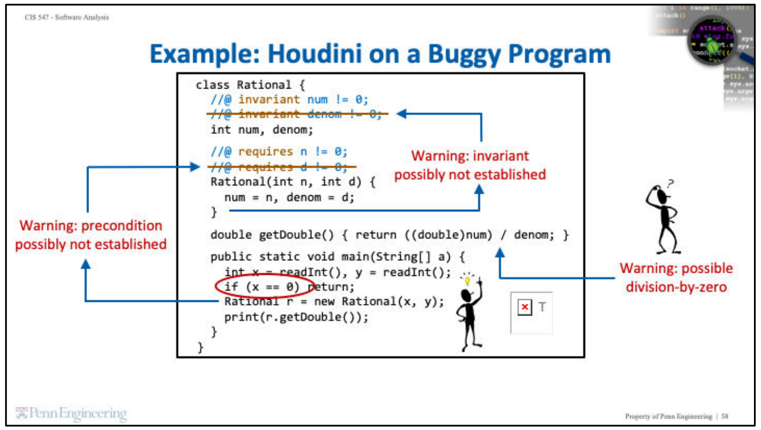
Now, as a case study, let's see the workflow of Houdini on a buggy program. Here is the same Java class for Rational numbers we saw earlier, but with a subtle bug this time.

Take a moment to read through the main method and try to find the bug. The error is very subtle, so look carefully!

Now, let's dive into the steps that Houdini takes to automatically infer invariants.

First, Houdini generate candidate invariants. Here, Houdini uses a static invariant extractor to generate the annotations shown in blue.

Then, we feed our candidates into the invariant checker.



Example: Houdini on a Buggy Program

```
class Rational {
  //@ invariant num != 0;
  ~~//@ invariant denom != 0;~~
  int num, denom;

  //@ requires n != 0;
  ~~//@ requires d != 0;~~
  Rational(int n, int d) {
    num = n, denom = d;
  }

  double getDouble() { return ((double)num) / denom; }

  public static void main(String[] a) {
    int x = readInt(), y = readInt();
    if (x == 0) return;
    Rational r = new Rational(x, y);
    print(r.getDouble());
  }
}
```

Warning: invariant possibly not established
Warning: precondition possibly not established
Warning: possible division-by-zero

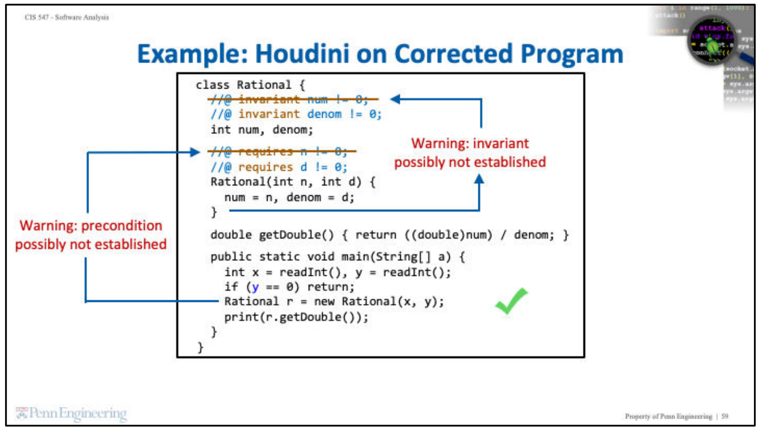
Houdini checks the annotations in a highly modular fashion. At each step, it picks a single pre-condition or a single post-condition to check. Furthermore, it analyzes the body of a single method at each step. If that method calls other methods, it plugs in the pre- and post- conditions of those methods at the call sites, effectively limiting the scope of each invocation of the invariant checker to a single method. This modular reasoning is crucial to Houdini's scalability: applying the invariant checker to entire multi-function programs at once would cause the checker to quickly run out of resources.

First, let's see how Houdini checks the annotations of the Rational constructor, in this case, just its two pre-conditions. For this purpose, the algorithm sets out to find each place in the program where the constructor is called. Here, the constructor is only called in the main method. Then, at each such call site, the algorithm checks that the precondition holds prior to the constructor's invocation. In this case, the algorithm checks the body of the main method. Here, the second argument to the constructor, the variable y, may equal zero. Because unlike the first argument, the variable x, there is no guard to ensure that it is non-zero. So, Houdini removes the candidate precondition d != 0 corresponding to the second argument, as it cannot be proven to be valid. But it preserves the candidate pre-condition n != 0 since it is proven to be valid at the only call site of the Rational constructor.

Now that we have checked the pre-conditions of the Rational constructor, Houdini picks another annotation to check, let's say the post-conditions of the Rational

constructor. There aren't any explicit post-conditions for the constructor, but there are two candidate class invariants. Recall that class invariants must be maintained following calls to public methods. Thus, we can think of class invariants as a kind of implicit post-condition. As before, while checking any annotation, Houdini assumes that all remaining annotations are valid. In particular, when checking a post-condition of a method, it assumes that its remaining pre-conditions are valid. Again, denom may be equal to zero, since it is assigned the value of variable d in the constructor and we no longer have the pre-condition that argument d is not equal to zero. As a result, Houdini removes denom != 0 from its set of candidate class invariants, as it couldn't be proven to hold for objects created by the constructor.

With the class invariant denom != 0 no longer present, ESC/Java reports a possible division-by-zero error in the body of the getDouble method, and thereby rejects the program as possibly unsafe. Now, the developer may ask, why did ESC/Java fail to prove the absence of such an error? The developer can make Houdini retrace the story back, allowing to see why it reported the error. In fact, we can follow the removed pre-condition back to the constructor call in main. Here, we can see that the d != 0 pre-condition of the constructor was removed by Houdini because it failed to prove that the corresponding argument passed to the constructor call in the main method, the variable y, is non-zero. Now the developer can spot the mistake in the check, and replace x by y.


CIS 547 - Software Analysis

Example: Houdini on Corrected Program

```
class Rational {
  ~~//@ invariant num != 0;~~
  //@ invariant denom != 0;
  int num, denom;

  ~~//@ requires n != 0;~~
  //@ requires d != 0;
  Rational(int n, int d) {
    num = n, denom = d;
  }

  double getDouble() { return ((double)num) / denom; }

  public static void main(String[] a) {
    int x = readInt(), y = readInt();
    if (y == 0) return;
    Rational r = new Rational(x, y);
    print(r.getDouble());
  }
}
```

Warning: invariant possibly not established
Warning: precondition possibly not established

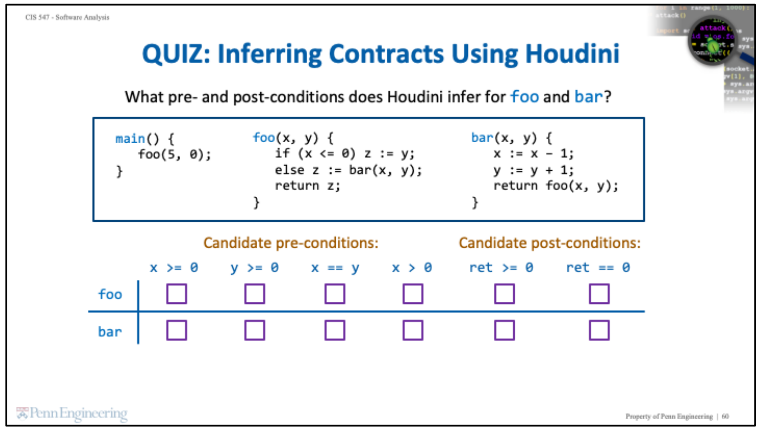
PennEngineering Property of Penn Engineering | 59

Here is the corrected version of the Rational class. Now let’s run the Houdini algorithm on the fixed version of the code.

Again, the algorithm sets out to find each place the constructor is called. At the call site in the main method, the algorithm checks whether each of the pre-conditions holds prior to the constructor’s invocation. Houdini again removes one candidate pre-condition, but this time, "n != 0" as opposed to “d != 0”.

Now, we set out to check the implicit post-conditions written as class invariants. This time, Houdini keeps the class invariant denom != 0 since it holds at the exit of the constructor, assuming the precondition d != 0 at the entry of the constructor, but it removes the class invariant "num != 0” since it cannot be proven to hold at the exit of the constructor in the absence of the pre-condition n != 0 at the entry of the constructor.

At this point, no other invariant warnings are emitted. Therefore, no refutable invariants are left, and Houdini terminates. This time, ESC/Java does not report any division-by-zero warnings: since denom != 0 is a valid class invariant, ESC/Java can use it as a pre-condition of the getDouble() method to prove that it will never divide by zero in any execution.


CIS 547 - Software Analysis

QUIZ: Inferring Contracts Using Houdini

What pre- and post-conditions does Houdini infer for foo and bar?

```
main() {
   foo(5, 0);
}

foo(x, y) {
   if (x <= 0) z := y;
   else z := bar(x, y);
   return z;
}

bar(x, y) {
   x := x - 1;
   y := y + 1;
   return foo(x, y);
}
```

	Candidate pre-conditions:				Candidate post-conditions:	
	x >= 0	y >= 0	x == y	x > 0	ret >= 0	ret == 0
foo	☐	☐	☐	☐	☐	☐
bar	☐	☐	☐	☐	☐	☐

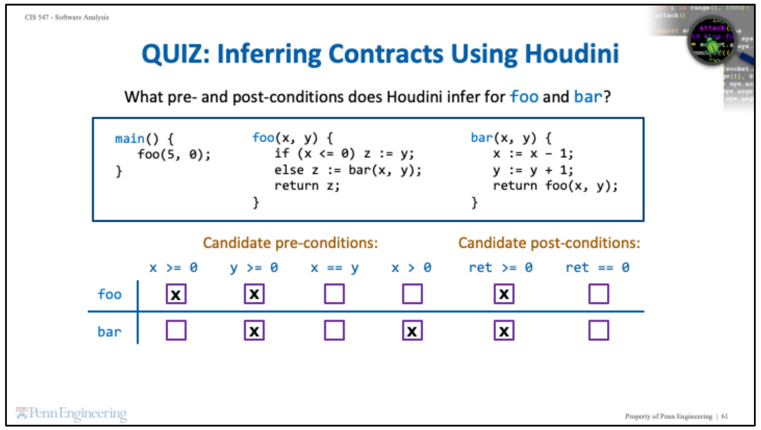
PennEngineering Property of Penn Engineering | 60

{QUIZ SLIDE}

Now, to check your understanding of program invariants and the Houdini algorithm, consider the following quiz.

There are two methods, foo and bar. The main function calls foo. Foo calls bar, and bar in turn calls foo. So this program has mutual recursion.

Recall that Houdini only checks the body of one method at a time. That is, even though foo contains a call to bar, Houdini sets out to verify foo by assuming any remaining invariants of bar are valid. And similarly, when Houdini sets out to verify bar, it assumes any remaining invariants of foo are valid.


CIS 547 - Software Analysis

QUIZ: Inferring Contracts Using Houdini

What pre- and post-conditions does Houdini infer for foo and bar?

```
main() {            foo(x, y) {                    bar(x, y) {
   foo(5, 0);          if (x <= 0) z := y;            x := x - 1;
}                      else z := bar(x, y);           y := y + 1;
                       return z;                      return foo(x, y);
                    }                              }
```

	Candidate pre-conditions:				Candidate post-conditions:	
	`x >= 0`	`y >= 0`	`x == y`	`x > 0`	`ret >= 0`	`ret == 0`
foo	☒	☒	☐	☐	☒	☐
bar	☐	☒	☐	☒	☒	☐

PennEngineering

Property of Penn Engineering | 61

{SOLUTION SLIDE}

Here are the computed invariants for methods foo and bar.

Let's start by reviewing the invariants for foo. Suppose we start by checking all of the call sites for foo.

First, by checking the call to foo in the main method, we eliminate the precondition (x == y) for foo because 5 is not equal to 0.

Next, we analyze the call to foo in bar. Here, we assume all remaining pre-conditions are valid. Most notably, x > 0 for bar. However, following the decrement operation on x in bar, this predicate may no longer hold at the call to foo, so we eliminate the precondition (x > 0) for foo.

Now, to check the post-conditions of foo, we again assume all remaining pre-conditions are valid, and check the body of foo.

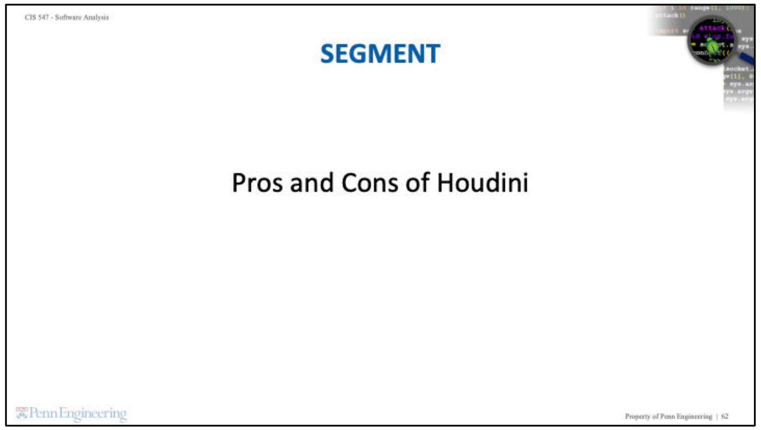
z is assigned via two paths: the if branch and the else branch. Let's review both.

In the if branch, z is set equal to y. By the pre-condition y >= 0, we can assume that ret >= 0 upon exiting this branch. However, we can remove the post-condition ret == 0.

Now, let's check the else branch. We assume the current pre and post-conditions for bar are valid, in particular, the post-condition of bar that its return result is greater-than-or-equal-to zero. Since foo goes on to return that same result after assigning it to z, we establish that foo’s return result is greater-than-or-equal-to zero and continue.

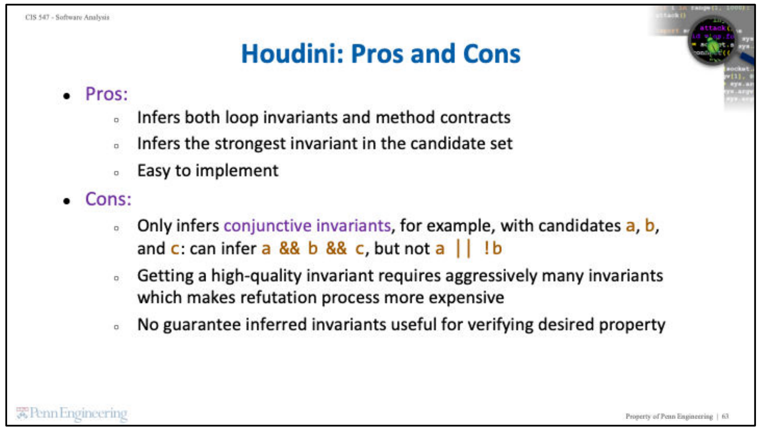
Analyzing the invariants of bar follows similarly and we leave it as an exercise.

Finally, you can do another iteration through this entire process, using only the remaining pre- and post-conditions of foo and bar, to convince yourself that none of them are removed.



SEGMENT

Pros and Cons of Houdini



Houdini: Pros and Cons

- Pros:
  - Infers both loop invariants and method contracts
  - Infers the strongest invariant in the candidate set
  - Easy to implement
- Cons:
  - Only infers conjunctive invariants, for example, with candidates a, b, and c: can infer `a && b && c`, but not `a || !b`
  - Getting a high-quality invariant requires aggressively many invariants which makes refutation process more expensive
  - No guarantee inferred invariants useful for verifying desired property

We will wrap up our discussion of specification mining by outlining the pros and cons of the Houdini approach.

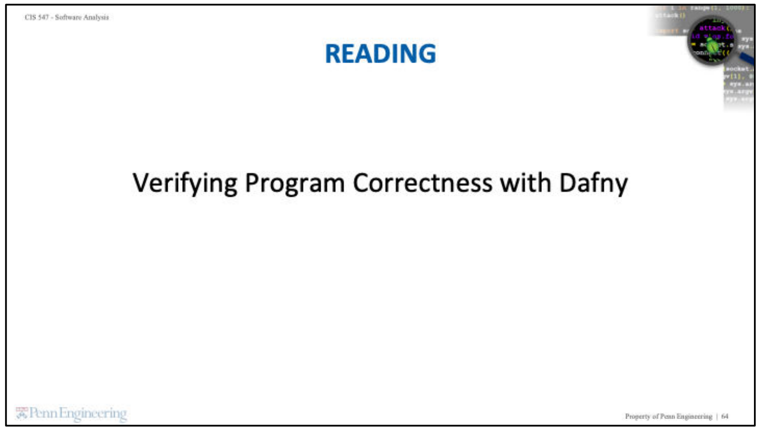
The Houdini algorithm has many advantages. It can infer both loop invariants and method contracts – the pre- and post- conditions. It also infers the strongest invariant in the candidate set, that is, the largest subset of the candidate set. Furthermore, it is easy to implement.

However, it has some limitations.

First, it only infers invariants that are in form of a conjunction of invariants. For instance, if the set of candidate invariants consists of a, b, and c, Houdini can only infer the conjunction of a, b, and c. but not the disjunction of a with the negation of b.

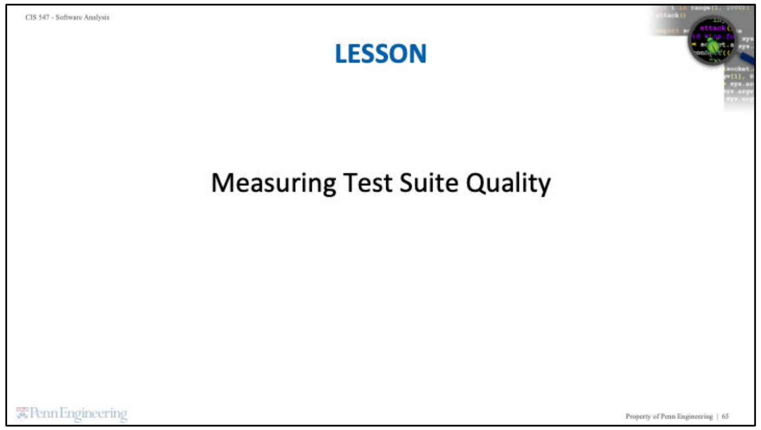
Second, getting a high-quality invariant requires generating many candidate invariants. However, this makes the refutation process more expensive and time-consuming. Therefore, there is a trade-off between the number of generated candidates and how quick the result is desired.

Lastly, there is no guarantee that the inferred invariants are actually useful for verifying correctness properties of interest, due to the above limitations.



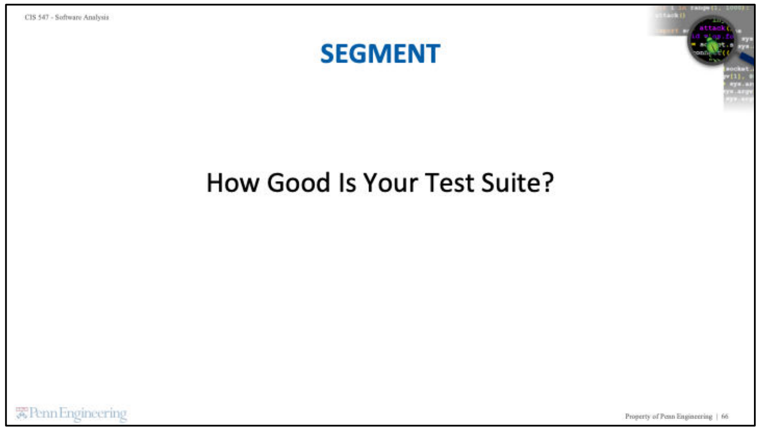
READING

Verifying Program Correctness with Dafny



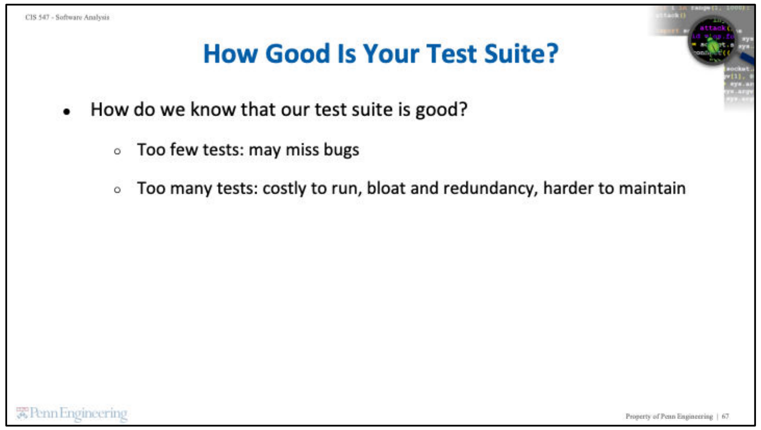
LESSON

Measuring Test Suite Quality



SEGMENT

How Good Is Your Test Suite?



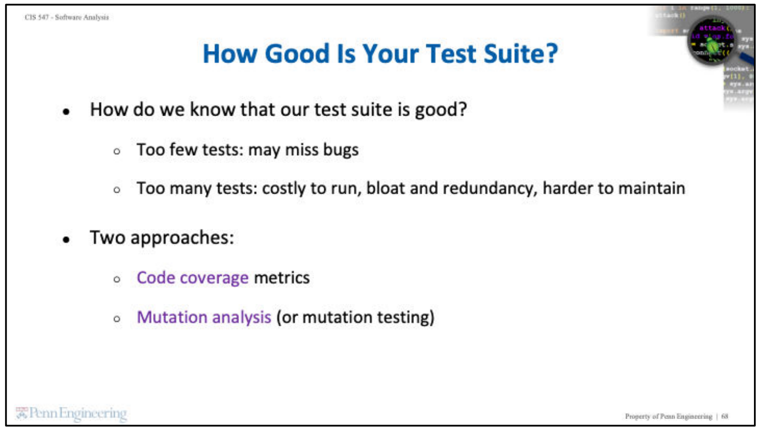
How Good Is Your Test Suite?

- How do we know that our test suite is good?
  - Too few tests: may miss bugs
  - Too many tests: costly to run, bloat and redundancy, harder to maintain

We are now familiar with ways to write or infer a program's specifications. Now, suppose we've developed a suite of test cases that check whether the program meets these specifications. Let's stop for a moment and think about the quality of our tests.

Have we included enough tests? Having too few tests is a bigger problem than having too many tests. If we have too few tests, we might miss regressions that occur as the code evolves; that is, a bug might not cause any of those few tests to fail, and thereby elude detection.

Have we included too many tests? If we have too many tests, running all of them before each code commit, or even nightly, could get expensive. Just like the problem of code bloat, we might run into the problem of test suite bloat, with lots of redundant tests. More tests are also harder to maintain and keep up-to-date than fewer tests.


CIS 547 - Software Analysis

How Good Is Your Test Suite?

- How do we know that our test suite is good?
  - Too few tests: may miss bugs
  - Too many tests: costly to run, bloat and redundancy, harder to maintain
- Two approaches:
  - Code coverage metrics
  - Mutation analysis (or mutation testing)

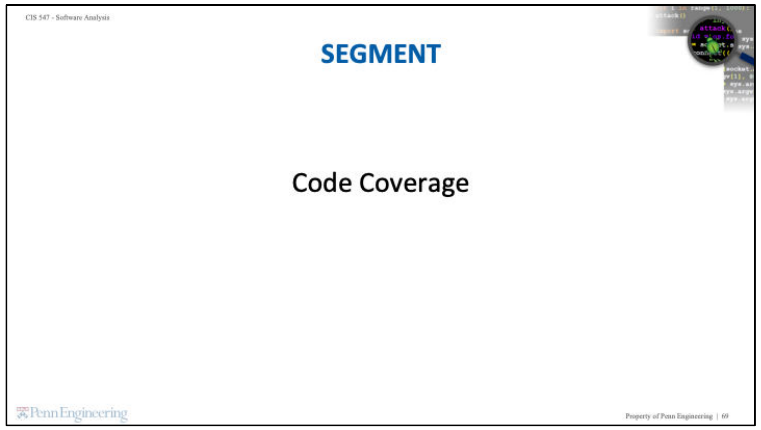
Penn Engineering

Property of Penn Engineering | 68

We will discuss two approaches to systematically quantify how good our testing suite is.

One approach to measuring the quality of our test suite is to use code coverage metrics. For example, we can check whether each statement of code has been executed at least once over the course of all tests, whether every possible route through the code has been taken, and so forth.

A second approach we can take to measure our test suite's quality is to use mutation analysis. In this approach, we randomly "mutate" the program under testing in various ways and then run the same tests on the mutant code as are used on the original code. If no tests fail on the mutant code, there is the implication that the testing suite may not be strong enough to distinguish correct from incorrect code.

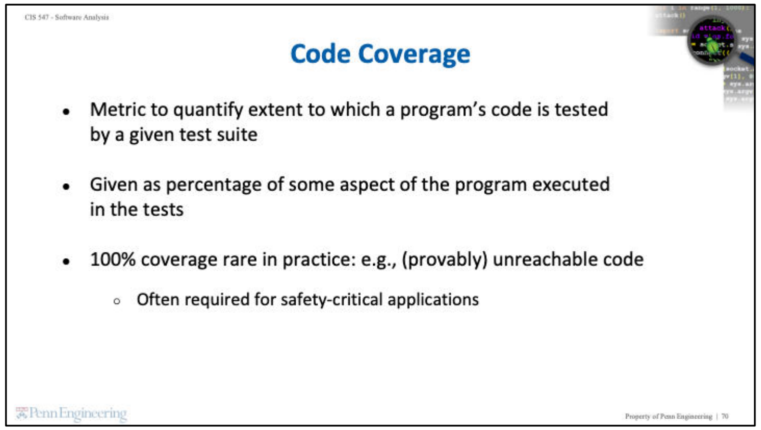

CIS 547 - Software Analysis

SEGMENT

Code Coverage

Penn Engineering

Property of Penn Engineering | 69


CIS 547 - Software Analysis

Code Coverage

- Metric to quantify extent to which a program's code is tested by a given test suite
- Given as percentage of some aspect of the program executed in the tests
- 100% coverage rare in practice: e.g., (provably) unreachable code
  - Often required for safety-critical applications

Penn Engineering

Property of Penn Engineering | 70

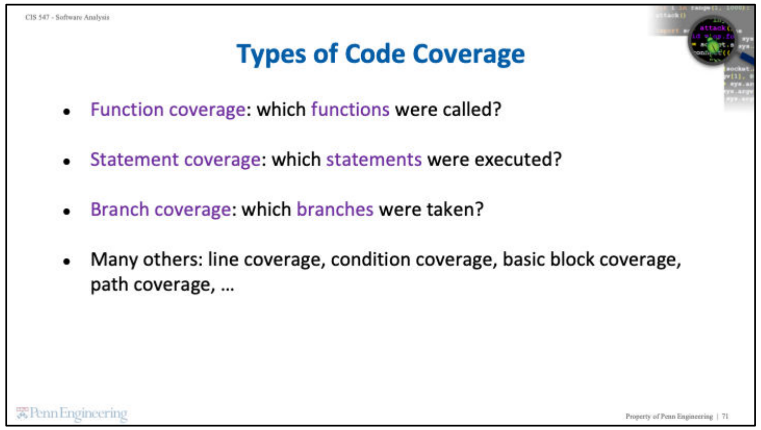
Let's study the first approach, code coverage, in some more detail.

Code coverage is a metric to quantify the extent to which a program's code is tested by a given test suite.

Code coverage is given as a percentage from 0 to 100 percent, of some aspect of the program that was executed in the tests. We will shortly see common examples of these aspects.

Higher code coverage is generally indicative of a better test suite and a better tested program. In practice, however, it's rare to see perfect or near-perfect coverage for a given program on a test suite. This often occurs because large systems have provably unreachable code.

However, some safety-critical applications (such as in airline navigation or nuclear weaponry) are often required to demonstrate perfect code coverage under some metric.


CIS 547 - Software Analysis

Types of Code Coverage

- Function coverage: which functions were called?
- Statement coverage: which statements were executed?
- Branch coverage: which branches were taken?
- Many others: line coverage, condition coverage, basic block coverage, path coverage, …

Penn Engineering

Property of Penn Engineering | 71

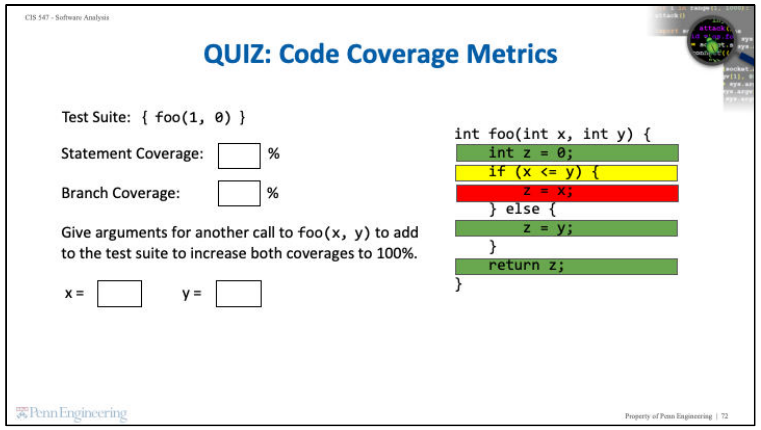
There are various aspects of programs that are commonly used to measure code coverage.

Function coverage measures the number of functions called out of the total number of functions in the program.

Statement coverage measures the number of statements that are executed out of all statements in a program.

Branch coverage measures the fraction of branches of each control structure that were taken (for example, only taking the "true" path of if-statements would result in at most 50% branch coverage).

And there are many others, such as line coverage, condition coverage, basic block coverage, and path coverage.



QUIZ: Code Coverage Metrics

Test Suite: { foo(1, 0) }

Statement Coverage: ☐ %

Branch Coverage: ☐ %

Give arguments for another call to foo(x, y) to add to the test suite to increase both coverages to 100%.

x = ☐ y = ☐

```
int foo(int x, int y) {
  int z = 0;
  if (x <= y) {
    z = x;
  } else {
    z = y;
  }
  return z;
}
```

{QUIZ SLIDE}

Let's look at coverage metrics for a small test suite.

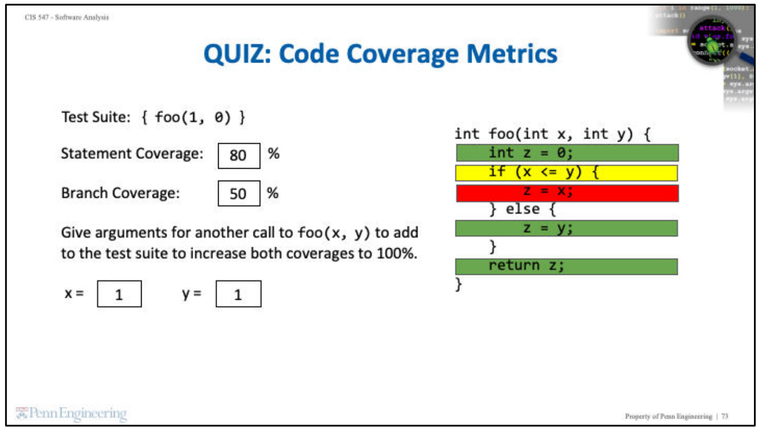
In the code snippet to the right, we will highlighted each statement according to its coverage:

Green will mean that the statement has been executed; for a control statement, this requires that the boolean statement controlling the flow evaluate to both true and false at some point during the suite.
Yellow will mean that the line is a control statement where the controlling boolean expression has only been evaluated to either true or false but not both during the set of tests.
Red will mean that the statement has not been executed in any test in the suite.

The first test we'll add to our suite is to execute foo(1,0). During this test, the statement int z = 0 is executed, so the statement is highlighted green. Then the boolean expression x <= y is evaluated to false, so we highlight the if-statement yellow and skip past z = x, leaving it red. We execute the statement z = y inside the else-block, so we highlight it green, and finally we execute the return statement, so we also highlight it green.

Now it's your turn. For this very small test suite, compute both the statement coverage

and the branch coverage. Then, suppose we want to add a new function call to our test suite. Pick values for arguments x and y that we can pass to foo in order to raise both of these coverage metrics to 100%.


CIS 547 - Software Analysis

QUIZ: Code Coverage Metrics

Test Suite: { foo(1, 0) }

Statement Coverage: 80 %

Branch Coverage: 50 %

Give arguments for another call to foo(x, y) to add to the test suite to increase both coverages to 100%.

x = 1 y = 1

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

PennEngineering

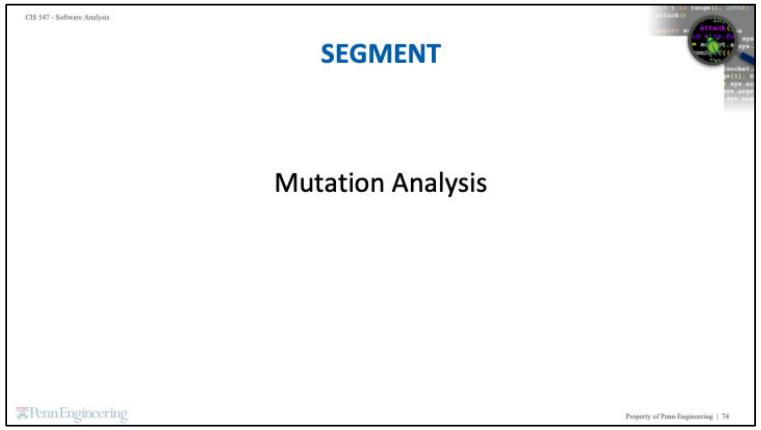
Property of Penn Engineering | 73

{SOLUTION SLIDE}

Out of the five executable statements in this function, four are executed when we call foo(1,0), giving a statement coverage metric of 80%.

And, since the if-statement's boolean expression only evaluates to false but not true when we call foo(1,0), we've only covered 50% of the possible branches in this code.

In order to increase these metrics to 100%, we need to make sure that x <= y evaluates to true. Therefore, picking any set of arguments with x <= y, for example, x and y both 1, will ensure that the "true" branch of the if-statement is followed and that the statement z = x is executed.

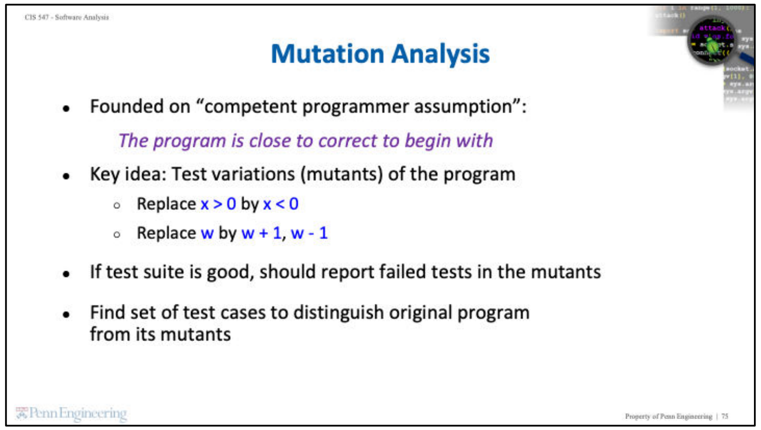

CIS 547 - Software Analysis

SEGMENT

Mutation Analysis

PennEngineering

Property of Penn Engineering | 74



Mutation Analysis

- Founded on "competent programmer assumption":

  The program is close to correct to begin with
- Key idea: Test variations (mutants) of the program
  - Replace x > 0 by x < 0
  - Replace w by w + 1, w - 1
- If test suite is good, should report failed tests in the mutants
- Find set of test cases to distinguish original program from its mutants

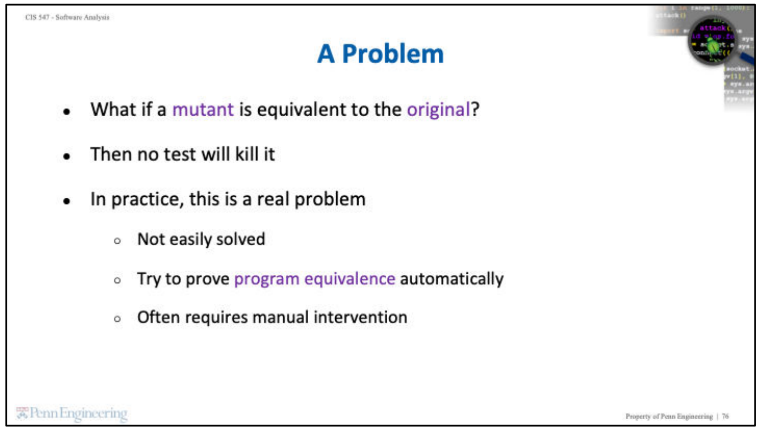
A key advantage of code coverage metrics is their simplicity: they are not difficult to measure. However, they are not perfect. It is possible to attain high code coverage with a test suite yet not discover potential bugs.

A more complex process called mutation analysis can be used to provide more confidence in one's test suite. Mutation analysis is founded on the "competent programmer assumption," which is that the program is close to being correct to begin with: the bugs we encounter are likely going to be based on small errors (such as missing a minus sign or replacing a 1 with the letter i) instead of sweeping errors in the program's overall logic.

The basic idea of mutation analysis, therefore, is to test variations---or mutants---of the program under test. For example, we must switch the direction of a greater-than sign to a less-than sign, or we might add or subtract 1 from some numerical expression.

If our test suite is robust, we should expect each of the mutants to fail some test, while the original program passes all tests. If we discover that certain mutants pass all the tests, it indicates that our test suite is not adequate, and we should therefore add new tests that distinguish the original program from its mutants.

You can read more about mutation testing at the link in the lecture handout.



A Problem

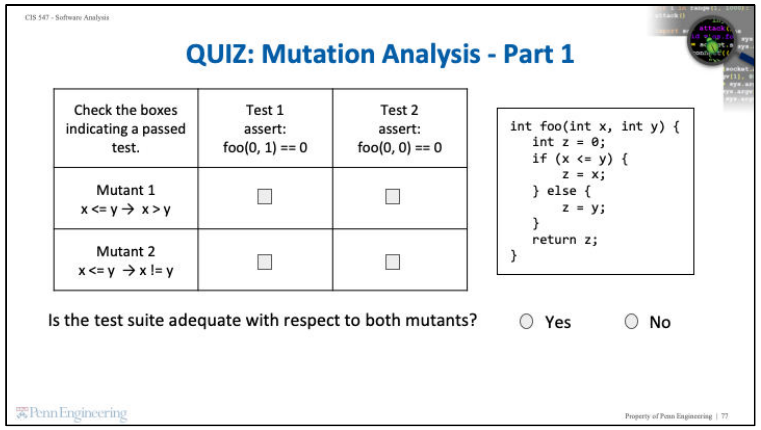
- What if a mutant is equivalent to the original?
- Then no test will kill it
- In practice, this is a real problem
  - Not easily solved
  - Try to prove program equivalence automatically
  - Often requires manual intervention

While mutation analysis is a powerful tool for measuring the quality of a test suite, one problem that could arise is that a mutant is created which is equivalent to the original program. That is, every input to the mutant program will generate the same output as the original program.

Since our testing protocol relies on observing differences between the original program's behavior and mutant programs' behavior, we will find ourselves in a situation where no test kills the equivalent mutant.

If we have such a persistent mutant, it becomes difficult to tell whether it indicates a lack of robustness in our testing or whether it is an equivalent mutant (in which case we can safely ignore the mutant).

This occurs often enough to be a practical problem, and it is not easily solved. We can try to automatically prove that two programs are equivalent, but this problem is undecidable in the general case and often requires a human to intervene and decide whether the mutant in question is indeed equivalent.



QUIZ: Mutation Analysis - Part 1

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 x <= y → x > y	☐	☐
Mutant 2 x <= y → x != y	☐	☐

```
int foo(int x, int y) {
   int z = 0;
   if (x <= y) {
      z = x;
   } else {
      z = y;
   }
   return z;
}
```

Is the test suite adequate with respect to both mutants? ○ Yes ○ No

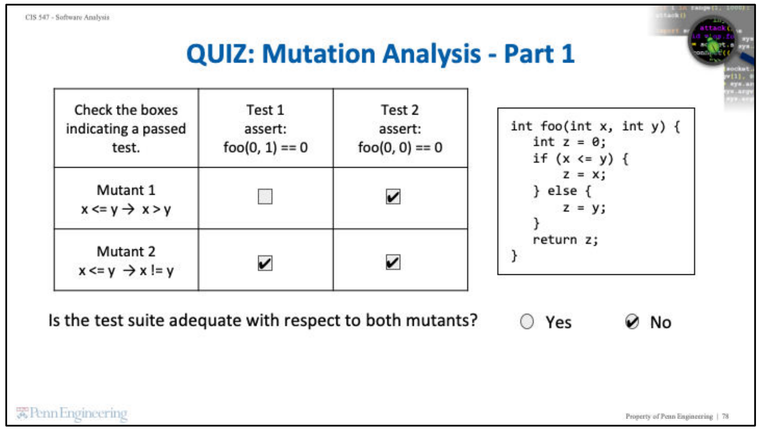
{QUIZ SLIDE}

Here's the function foo() that we saw before, and here are two possible mutations of the function. In the first mutant, the boolean expression x <= y is replaced by x > y. And in the second mutant, x <= y is replaced by x != y.

Let's consider the following test suite comprising these two tests: first, we assert that foo(0,1) == 0; second, we assert that foo(0,0) == 0. (Note that the original function foo will pass both of these tests.)

Check the boxes in the table to indicate which mutants pass which tests.

Then answer the question with either "yes" or "no": is this test suite adequate with respect to these two mutants?



QUIZ: Mutation Analysis - Part 1

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 x <= y → x > y	☐	✓
Mutant 2 x <= y → x != y	✓	✓

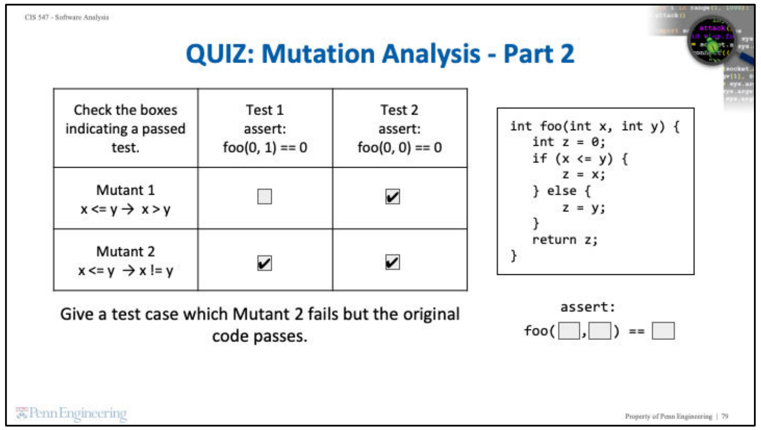
```
int foo(int x, int y) {
   int z = 0;
   if (x <= y) {
      z = x;
   } else {
      z = y;
   }
   return z;
}
```

Is the test suite adequate with respect to both mutants? Yes [✓ No]

{SOLUTION SLIDE}

In the first mutant, where x <= y is changed to x > y, the first test fails because foo(0,1) outputs 1, while the second test passes. So, for this particular mutant, our test suite is robust enough to indicate errors.

However, for the mutant in which x <= y is mutated to x != y, both the tests in our suite pass. This indicates that our test suite is NOT adequate: we need a test case which the second mutant fails but the original code passes.



QUIZ: Mutation Analysis - Part 2

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 x <= y → x > y	☐	☑
Mutant 2 x <= y → x != y	☑	☑

```
int foo(int x, int y) {
   int z = 0;
   if (x <= y) {
      z = x;
   } else {
      z = y;
   }
   return z;
}
```

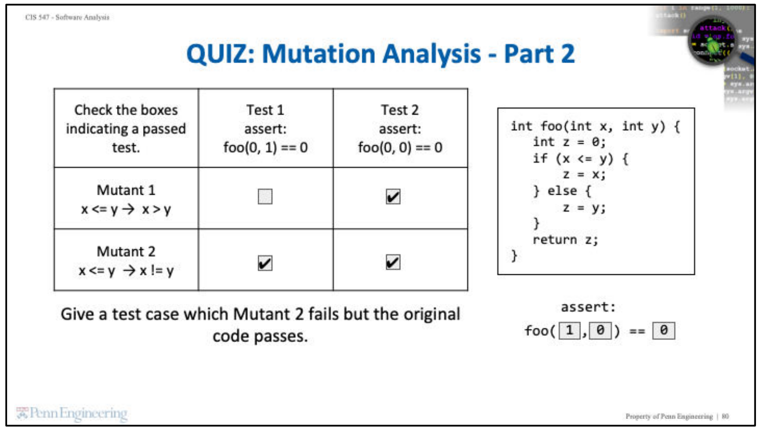
Give a test case which Mutant 2 fails but the original code passes.

```
assert:
foo(☐,☐) == ☐
```

{QUIZ SLIDE}

Let's make this test suite more robust with respect to the second mutant. We'll add a statement of the form, assert foo of blank comma blank equals blank.

By filling in the blanks with appropriate numbers, create a test case which Mutant 2 will fail but which the original code will still pass.



QUIZ: Mutation Analysis - Part 2

Check the boxes indicating a passed test.	Test 1 assert: foo(0, 1) == 0	Test 2 assert: foo(0, 0) == 0
Mutant 1 x <= y → x > y	☐	☑
Mutant 2 x <= y → x != y	☑	☑

```
int foo(int x, int y) {
   int z = 0;
   if (x <= y) {
      z = x;
   } else {
      z = y;
   }
   return z;
}
```

Give a test case which Mutant 2 fails but the original code passes.

```
assert:
foo(1,0) == 0
```

{SOLUTION SLIDE}

Our goal in this quiz is to come up with a choice of x, y, and z so that foo(x,y) == z is true for the original program and false for Mutant 2.

Notice that in the unmodified program, foo returns the minimum of its two arguments. So, in order for foo(x,y) == z to be a true expression, z needs to be the minimum of x and y.

For the mutant program, if x and y are unequal, then foo returns x; if x and y are equal, then foo returns y (which equals x). Thus, foo always returns x. So, in order for foo(x,y) == z to be a false expression, z cannot equal x.

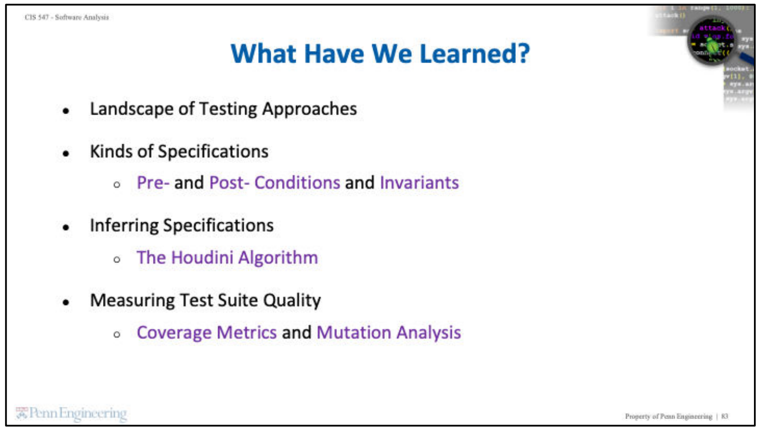
Therefore, we need to choose x, y, and z so that z is the minimum of x and y but not equal to x. This implies we need to choose y and x so that y is less than x. So, any answer where the first input is greater than the second input *and* where the second input is equal to the right-hand side of the expression will work. For example, foo(1,0) == 0.

LESSON

Review

SEGMENT

Review



What Have We Learned?

- Landscape of Testing Approaches
- Kinds of Specifications
  - Pre- and Post- Conditions and Invariants
- Inferring Specifications
  - The Houdini Algorithm
- Measuring Test Suite Quality
  - Coverage Metrics and Mutation Analysis

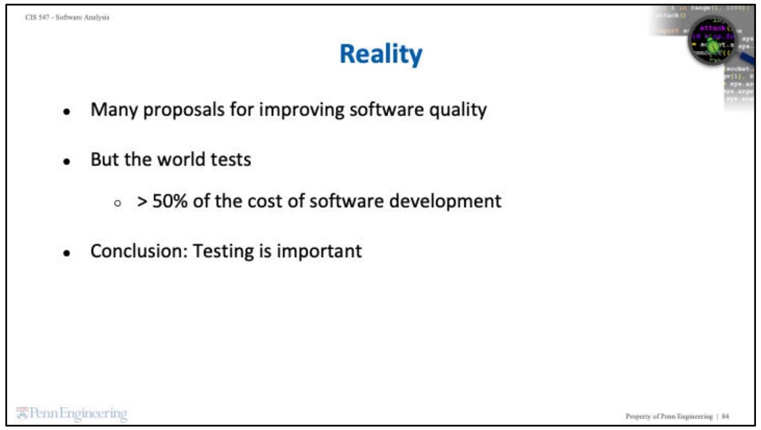
Let's take a look at what we've learned in this module on software specifications.

We began by surveying the landscape of testing paradigms, comparing and contrasting their costs and benefits.

Next, we looked at a variety of specifications. In particular, we learnt about pre- and post-conditions and invariants for specifying safety properties of units of a program such as loops, methods, and classes.

Then, we will discussed an algorithm to automatically infer specifications, called the Houdini Algorithm. The Houdini algorithm makes the costly task of writing specifications easier.

Finally, we studied two methods of quantifying the quality of a given set of tests.



Reality

- Many proposals for improving software quality
- But the world tests
  - > 50% of the cost of software development
- Conclusion: Testing is important

There are many proposals for improving software quality, including several that you will learn in this course, many of which attempt to detect program errors before the code even reaches the tester.

However, just as more than half of Microsoft's development costs go to testing, so do more than half the costs of the entire software development industry.

This is because writing error-free programs and verifying programs to be error-free are problems that are inherently undecidable: they can never be fully automated. So there will always be a need to spend resources on testing, and developing tools to improve the efficiency of the testing process.

In the next module, we will look at some more techniques to do just that, through the automated generation of test cases.