L10: Software Specifications

Software Analysis

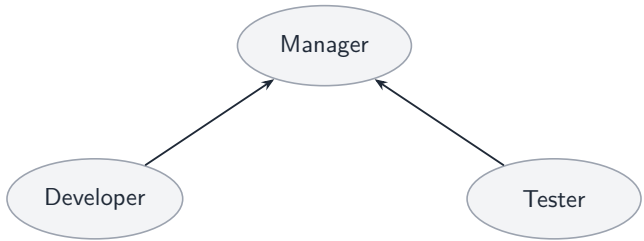
IIT Guwahati · 18 August 2026

What Is a Specification?

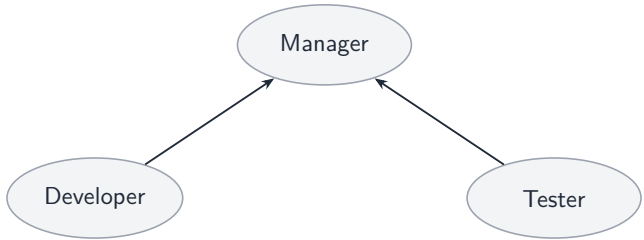
A **specification** is a high-level description of the software's *intended* behaviour.

Naik, *Software Specifications*, CIS 5470, Penn

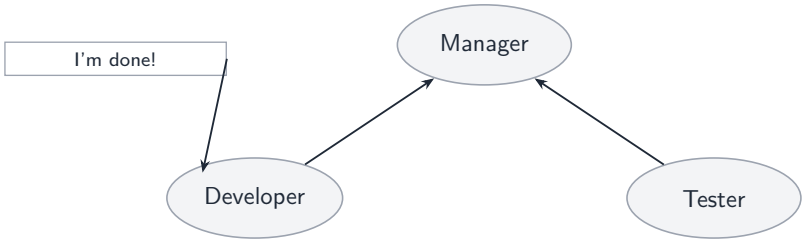
Software Development Scenario



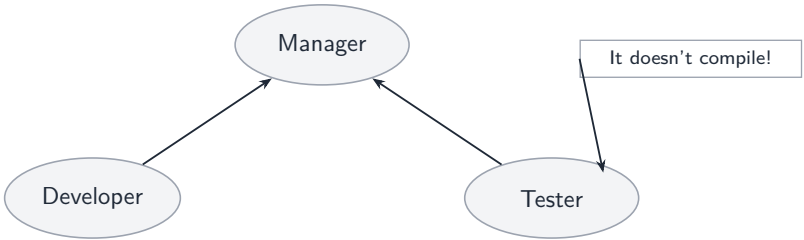
Software Development Scenario



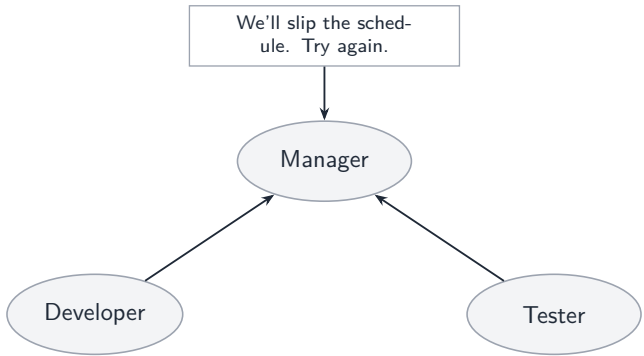
Software Development Scenario



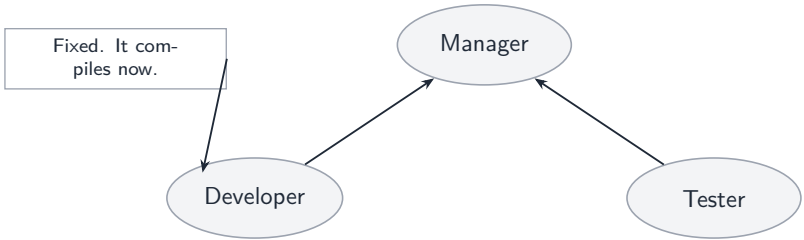
Software Development Scenario



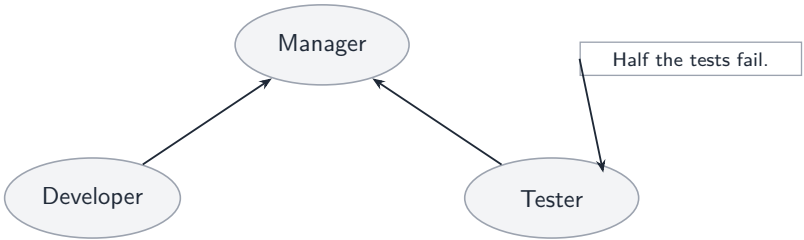
Software Development Scenario



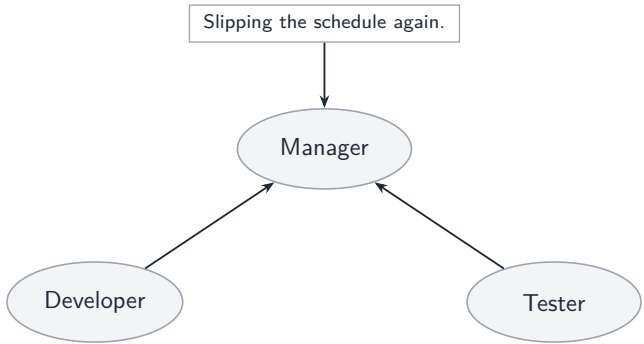
Software Development Scenario



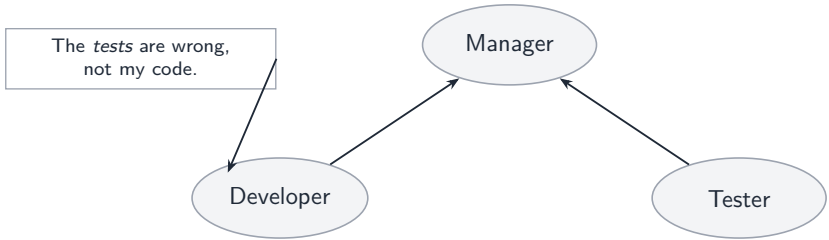
Software Development Scenario



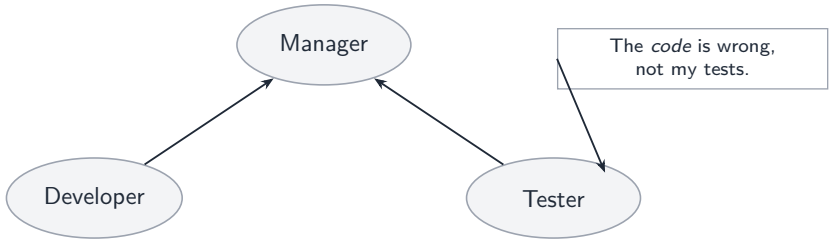
Software Development Scenario



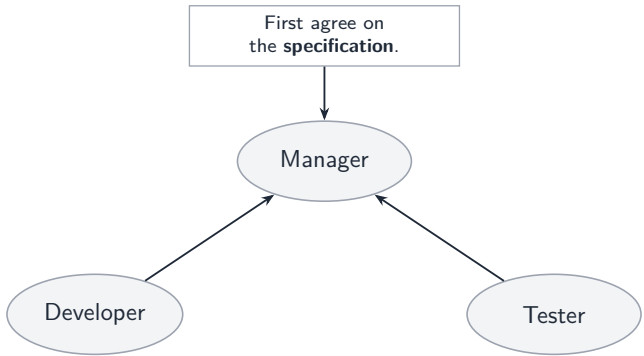
Software Development Scenario



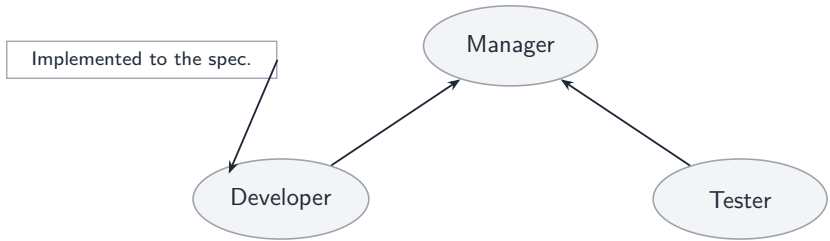
Software Development Scenario



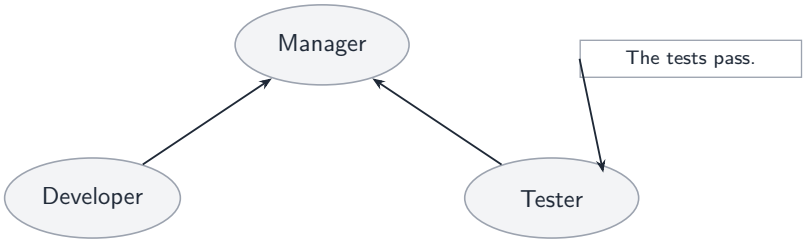
Software Development Scenario



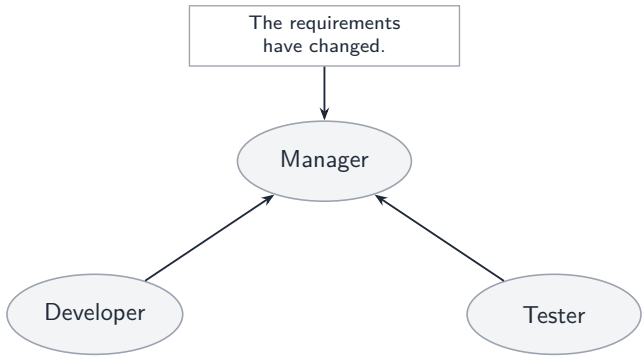
Software Development Scenario



Software Development Scenario



Software Development Scenario



Key Observations

- Specifications must be **explicit**

Key Observations

- Specifications must be **explicit**
- **Independent** development and testing

Key Observations

- Specifications must be **explicit**
- **Independent** development and testing
- Resources are **finite**

Key Observations

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

The Need for Specifications

- Testing checks whether the implementation agrees with the specification

The Need for Specifications

- Testing checks whether the implementation agrees with the specification
- Without a specification, there is nothing to test

The Need for Specifications

- Testing checks whether the implementation agrees with the specification
- Without a specification, there is nothing to test
- Testing is a form of **consistency checking** between implementation and specification

The Need for Specifications

- Testing checks whether the implementation agrees with the specification
- Without a specification, there is nothing to test
- Testing is a form of **consistency checking** between implementation and specification
- Both artifacts may be wrong, and the check still succeeds

Independent Development and Testing

- The **implementation** and its specification should be developed independently when possible

Independent Development and Testing

- The **implementation** and its specification should be developed independently when possible
- Independent artifacts are less likely to contain the **same mistake**

Independent Development and Testing

- The **implementation** and its specification should be developed independently when possible
- Independent artifacts are less likely to contain the **same mistake**
- A specification remains useful even when written by the developer

Independent Development and Testing

- The **implementation** and its specification should be developed independently when possible
- Independent artifacts are less likely to contain the **same mistake**
- A specification remains useful even when written by the developer
- A specification describes a **specific facet** of behaviour and is usually much simpler than the implementation

Other Observations

- **Resources are finite**: only finitely many tests can be written and run

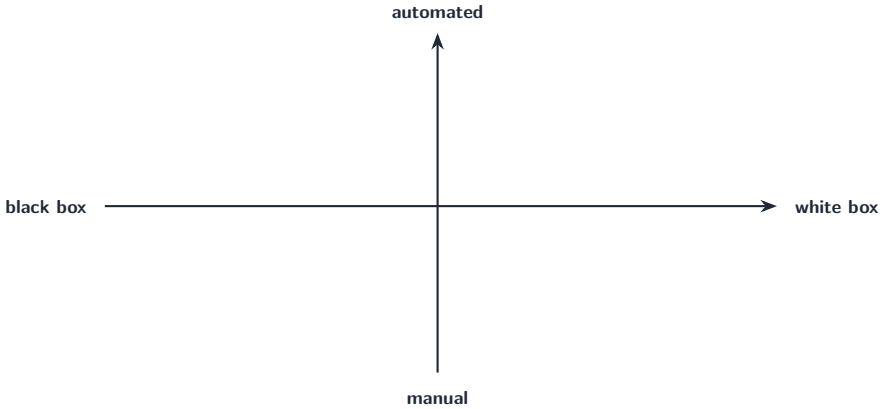
Other Observations

- **Resources are finite**: only finitely many tests can be written and run
- **Specifications evolve**: a test suite valid today need not stay valid

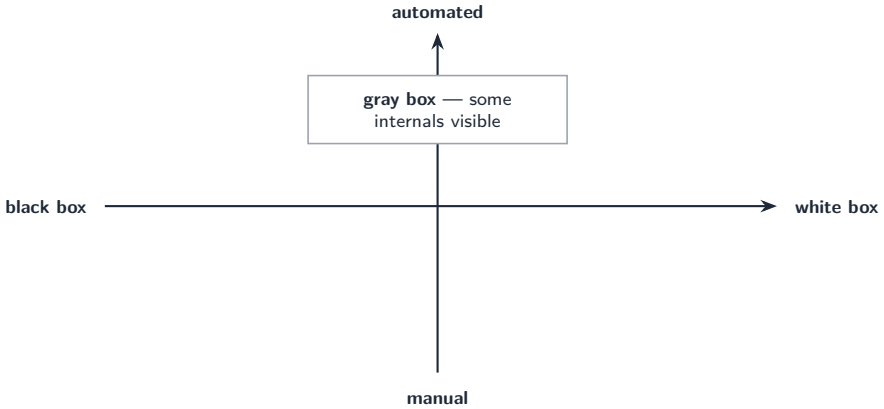
Other Observations

- **Resources are finite**: only finitely many tests can be written and run
- **Specifications evolve**: a test suite valid today need not stay valid
- Testing is necessary, ongoing and resource-intensive, which motivates **automated testing**

Classification of Testing Approaches

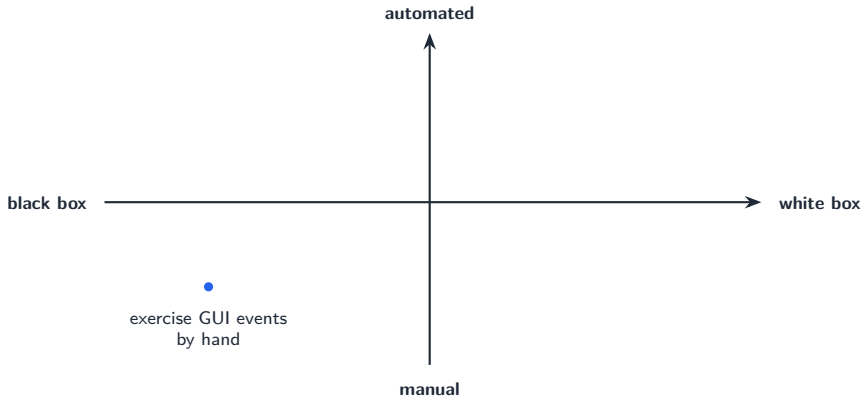


Classification of Testing Approaches

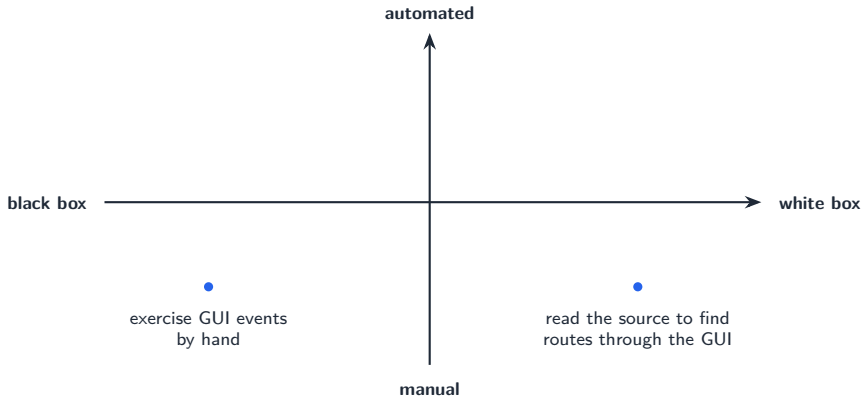


Both axes are continua, not discrete categories.

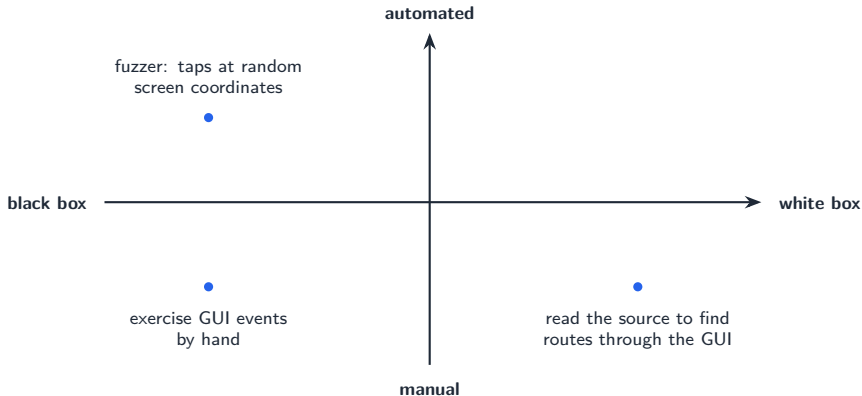
Example: Testing an Android Application



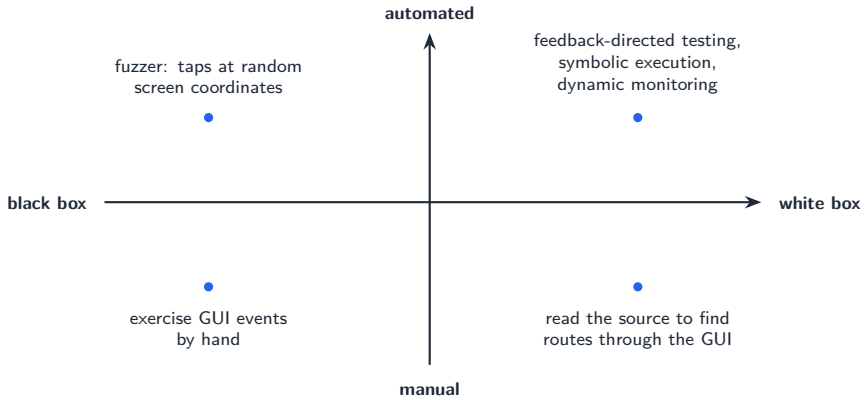
Example: Testing an Android Application



Example: Testing an Android Application



Example: Testing an Android Application



Automated vs. Manual Testing

Automated

Finds bugs more quickly

No need to write tests

No need to maintain tests when the software changes

Automated vs. Manual Testing

Automated

Finds bugs more quickly

No need to write tests

No need to maintain tests when the software changes

Manual

More efficient test suites

Potentially better coverage

Automated vs. Manual Testing

Automated

Finds bugs more quickly

No need to write tests

No need to maintain tests when the software changes

Manual

More efficient test suites

Potentially better coverage

A **semi-automated** approach combines the two: the human supplies the grammar of valid inputs, the machine generates within it.

Black-Box vs. White-Box Testing

Black box

Does not require modifying the code

Does not need to analyse or study the code

Code can be in any form: managed, binary, obfuscated

Black-Box vs. White-Box Testing

Black box

Does not require modifying the code

Does not need to analyse or study the code

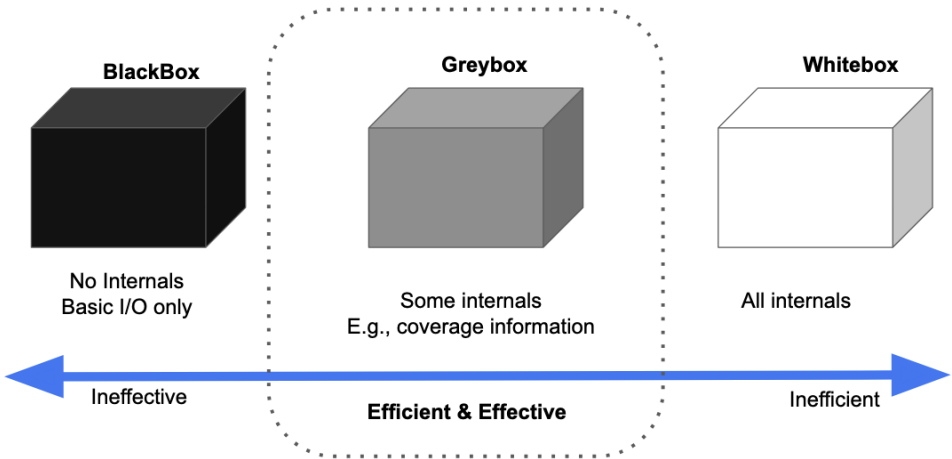
Code can be in any form: managed, binary, obfuscated

White box

More efficient test suites

Potentially better coverage

Testing Approaches



An Example: Mobile App Security

DroidKungFu, found on third-party app stores in 2011, collects sensitive device information and reports it to remote command-and-control servers.

An Example: Mobile App Security

DroidKungFu, found on third-party app stores in 2011, collects sensitive device information and reports it to remote command-and-control servers.

Black box

Start the app and monitor network activity, capturing the connection to the suspicious host.

An Example: Mobile App Security

DroidKungFu, found on third-party app stores in 2011, collects sensitive device information and reports it to remote command-and-control servers.

Black box

Start the app and monitor network activity, capturing the connection to the suspicious host.

White box

Inspect the source or binary code, revealing the call that opens that connection.

The Automated Testing Problem

```
void f(int a[30]) {
    if (a[0]) { ... } else { ... }
    if (a[1]) { ... } else { ... }
    ...                                    // 30 branch points
    if (a[29]) { ... } else { ... }
}
```

$2^{30} > 10^9$ execution paths; with a loop, the number of paths may be unbounded.

- Automated testing of programs is hard

The Automated Testing Problem

```
void f(int a[30]) {
    if (a[0]) { ... } else { ... }
    if (a[1]) { ... } else { ... }
    ...                                    // 30 branch points
    if (a[29]) { ... } else { ... }
}
```

$2^{30} > 10^9$ execution paths; with a loop, the number of paths may be unbounded.

- Automated testing of programs is hard
- Probably impossible for entire systems

The Automated Testing Problem

```
void f(int a[30]) {
    if (a[0]) { ... } else { ... }
    if (a[1]) { ... } else { ... }
    ...                                    // 30 branch points
    if (a[29]) { ... } else { ... }
}
```

$2^{30} > 10^9$ execution paths; with a loop, the number of paths may be unbounded.

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

Kinds of Specifications

Safety and Liveness

- **Safety**: something bad will never happen

  the program never reaches a bad state; e.g. assertion violations, types, typestate

Safety and Liveness

- **Safety**: something bad will never happen

  the program never reaches a bad state; e.g. assertion violations, types, typestate

- **Liveness**: something good will eventually happen

  the program eventually reaches a good state; e.g. termination, starvation freedom

Safety and Liveness

- **Safety**: something bad will never happen

  the program never reaches a bad state; e.g. assertion violations, types, typestate

- **Liveness**: something good will eventually happen

  the program eventually reaches a good state; e.g. termination, starvation freedom

Common Forms of Safety Properties

- **Types**

```
int x;
java.net.Socket s;
```

Common Forms of Safety Properties

- **Types**

  ```
  int x;
  java.net.Socket s;
  ```

- **Typestate properties**

  must not read data from a `java.net.Socket` until the socket is connected

Common Forms of Safety Properties

- **Types**

  `int x;`

  `java.net.Socket s;`

- **Typestate properties**

  must not read data from a `java.net.Socket` until the socket is connected

- **Assertions**

  implicit, e.g. `p != null` before each dereference of `p`

  explicit, e.g. `assert(y == 42)`

Common Forms of Safety Properties

- **Types**

  `int x;`

  `java.net.Socket s;`

- **Typestate properties**

  must not read data from a `java.net.Socket` until the 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)`

For `sqrt`: pre-condition `x >= 0`; post-condition: the return value `ret` satisfies `ret * ret == x`.

Common Forms of Safety Properties

- **Types**

  `int x;`

  `java.net.Socket s;`

- **Typestate properties**

  must not read data from a `java.net.Socket` until the 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**

For `sqrt`: pre-condition `x >= 0`; post-condition: the return value `ret` satisfies `ret * ret == x`.

Types

Adding the specification to the implementation allows the implementation to be checked against the specification.

```
int x;                      // specification:  x is an integer
x = read();
printf("%d", x + 1);        // implementation:  x is used as an integer
```

Naik, *Software Specifications*, CIS 5470, Penn

Types: Dynamically Typed Languages

Explicit type specifications are missing in dynamically typed or untyped programming languages.

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

Types: Dynamically Typed Languages

Explicit type specifications are missing in dynamically typed or untyped programming languages.

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

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

The Checker Framework

The Checker Framework extends Java's type system: developers add annotations that statically eliminate entire classes of runtime error, such as null pointer dereferences.

```
@NonNull   Object o;     // specification:  never null
@Nullable  String s;     // specification:  may be null

o = null;                // error:  incompatible types in assignment
```

The Checker Framework

The Checker Framework extends Java's type system: developers add annotations that statically eliminate entire classes of runtime error, such as null pointer dereferences.

```
@NonNull   Object o;     // specification:  never null
@Nullable  String s;     // specification:  may be null

o = null;                // error:  incompatible types in assignment
```

- Type annotations improve program readability

The Checker Framework

The Checker Framework extends Java's type system: developers add annotations that statically eliminate entire classes of runtime error, such as null pointer dereferences.

```
@NonNull   Object o;     // specification:  never null
@Nullable  String s;     // specification:  may be null

o = null;                // error:  incompatible types in assignment
```

- Type annotations improve program readability
- They are **machine checkable**: the compiler checks the implementation against the specification

The Checker Framework

The Checker Framework extends Java's type system: developers add annotations that statically eliminate entire classes of runtime error, such as null pointer dereferences.

```
@NonNull   Object o;     // specification:  never null
@Nullable  String s;     // specification:  may be null

o = null;                // error:  incompatible types in assignment
```

- Type annotations improve program readability
- They are **machine checkable**: the compiler checks the implementation against the specification
- The specification is re-checked as the implementation evolves

The Checker Framework

The Checker Framework extends Java's type system: developers add annotations that statically eliminate entire classes of runtime error, such as null pointer dereferences.

```
@NonNull   Object o;     // specification:  never null
@Nullable  String s;     // specification:  may be null

o = null;                // error:  incompatible types in assignment
```

- Type annotations improve program readability
- They are **machine checkable**: the compiler checks the implementation against the specification
- The specification is re-checked as the implementation evolves
- Further checkers: taint checking, GUI effects, and developer-defined checkers

The Checker Framework

The Checker Framework extends Java's type system: developers add annotations that statically eliminate entire classes of runtime error, such as null pointer dereferences.

```
@NonNull   Object o;     // specification:  never null
@Nullable  String s;     // specification:  may be null

o = null;                // error:  incompatible types in assignment
```

- Type annotations improve program readability
- They are **machine checkable**: the compiler checks the implementation against the specification
- The specification is re-checked as the implementation evolves

- Further checkers: taint checking, GUI effects, and developer-defined checkers
- Used at Amazon, Uber and Google; the nullness checker is part of Google's standard build

Typestate Properties

- Typestate refines types with **finite state** information

  a `FILE` is OPEN or CLOSED; a `Socket` is INITIALIZED, CONNECTED or CLOSED

Typestate Properties

- Typestate refines types with **finite state** information

  a `FILE` is OPEN or CLOSED; a `Socket` is INITIALIZED, CONNECTED or CLOSED

- It specifies which operations are valid in each state, and how operations affect the state

  `fread()` may be called only on a `FILE` in state OPEN; `fclose()` moves it from OPEN to CLOSED

Typestate Properties

- Typestate refines types with **finite state** information

  a `FILE` is OPEN or CLOSED; a `Socket` is INITIALIZED, CONNECTED or CLOSED

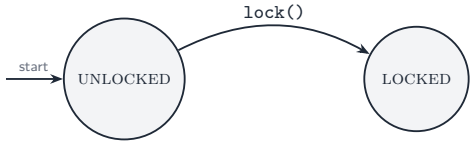
- It specifies which operations are valid in each state, and how operations affect the state

  `fread()` may be called only on a `FILE` in state OPEN; `fclose()` moves it from OPEN to CLOSED

- Also called **temporal safety properties**

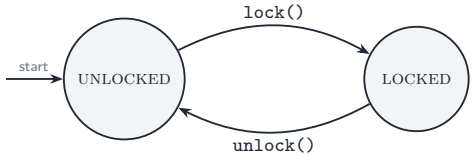
Example Property 1: Locking

Calls to `lock()` and `unlock()` must alternate. Re-acquiring an acquired lock, or releasing a released lock, is an error.



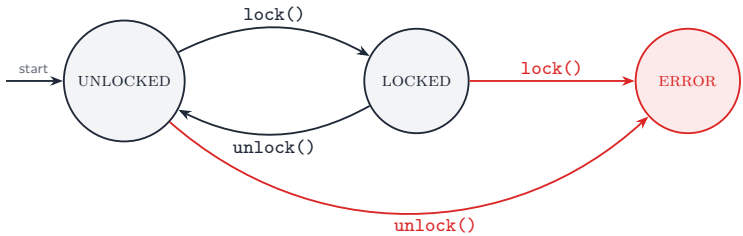
Example Property 1: Locking

Calls to `lock()` and `unlock()` must alternate. Re-acquiring an acquired lock, or releasing a released lock, is an error.



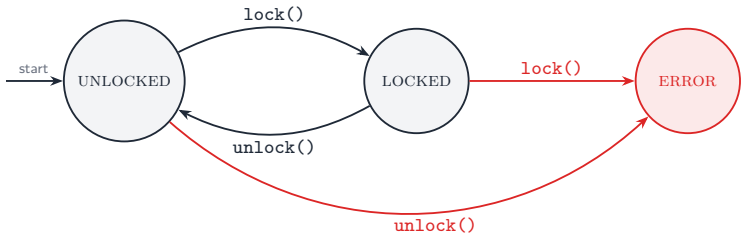
Example Property 1: Locking

Calls to `lock()` and `unlock()` must alternate. Re-acquiring an acquired lock, or releasing a released lock, is an error.



Example Property 1: Locking

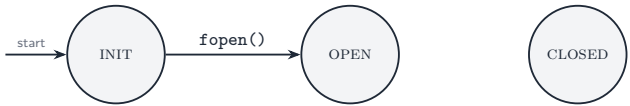
Calls to `lock()` and `unlock()` must alternate. Re-acquiring an acquired lock, or releasing a released lock, is an error.



This models `pthread_mutex_lock()` and `pthread_mutex_unlock()` in pthreads.

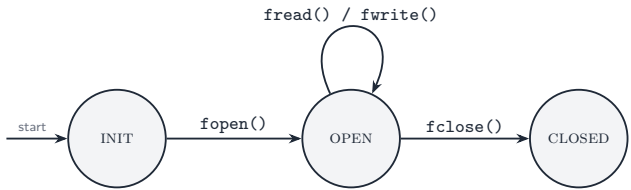
Example Property 2: File Management

A file must be opened before reading; it may be read any number of times before it is closed; and it must not be closed twice.



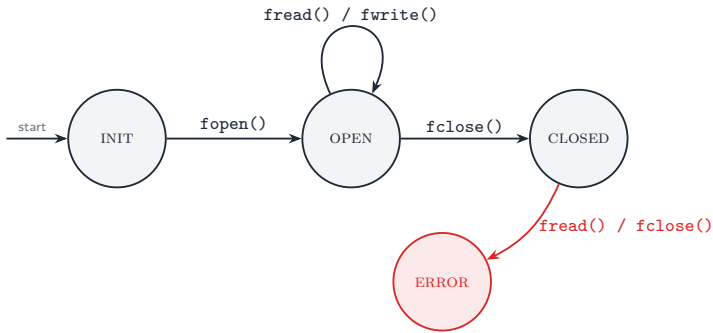
Example Property 2: File Management

A file must be opened before reading; it may be read any number of times before it is closed; and it must not be closed twice.



Example Property 2: File Management

A file must be opened before reading; it may be read any number of times before it is closed; and it must not be closed twice.



Pre- and Post-Conditions

- A **pre-condition** is a predicate assumed to hold before a function executes

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

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

Example: `pop()` on a Stack

```
// pre:  s.size() >= 1
Object pop(Stack s) {
    ...
}
// post:  s'.size() == s.size() - 1
```

Example: `pop()` on a Stack

```
// pre:  s.size() >= 1
Object pop(Stack s) {
    ...
}
// post:  s'.size() == s.size() - 1
```

- `s` is the stack on entry, `s'` the same stack on exit

Example: `pop()` on a Stack

```
// pre:  s.size() >= 1
Object pop(Stack s) {
    ...
}
// post:  s'.size() == s.size() - 1
```

- `s` is the stack on entry, `s'` the same stack on exit
- Written as comments here; we will later see machine-checkable annotations

Example: `pop()` on a Stack

```
// pre:  s.size() >= 1
Object pop(Stack s) {
    ...
}
// post:  s'.size() == s.size() - 1
```

- `s` is the stack on entry, `s'` the same stack on exit
- Written as comments here; we will later see machine-checkable annotations
- The post-condition says nothing about the remaining $n - 1$ elements

Pre- and Post-Conditions

- Most useful if they are **executable**

  written in the programming language itself, using a testing framework or an assertion

Pre- and Post-Conditions

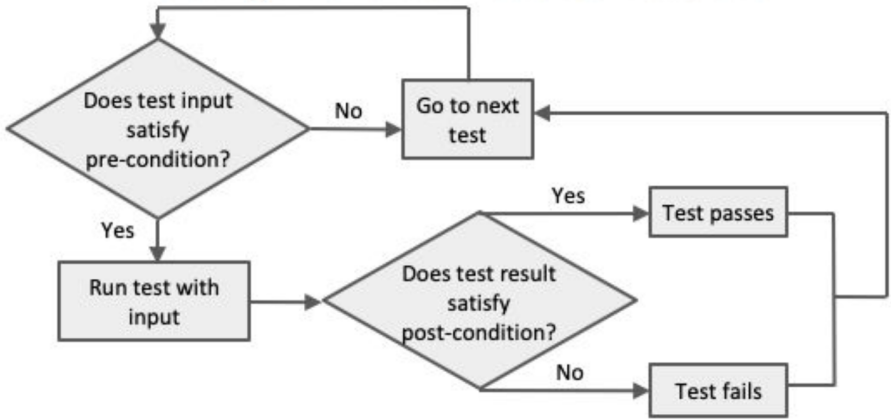
- Most useful if they are **executable**

  written in the programming language itself, using a testing framework or an assertion

- They **need not be precise**

  a precise condition may be more complex than the code it specifies; partial conditions trade precision for tractability

Using Pre- and Post-Conditions



Contracts in Testing and in Verification

$$\{P\}\ C\ \{Q\}$$

Naik, *Software Specifications*, CIS 5470, Penn

Contracts in Testing and in Verification

$$\{P\}\ C\ \{Q\}$$

- **Testing**: check P, execute C on one input, check Q

Contracts in Testing and in Verification

$$\{P\}\ C\ \{Q\}$$

- **Testing**: check P, execute C on one input, check Q
- **Verification**: assume P, reason over all inputs, prove Q

Contracts in Testing and Verification

$$\{\, x \geq 0 \,\} \qquad C:\ y = x + 1 \qquad \{\, y > 0 \,\}$$

Testing

Verification

Contracts in Testing and Verification

$$\{\, x \geq 0 \,\} \qquad C: \; y = x + 1 \qquad \{\, y > 0 \,\}$$

Testing

- Choose an input: $x = 5$

Verification

Contracts in Testing and Verification

$$\{\, x \geq 0 \,\} \qquad C: \; y = x + 1 \qquad \{\, y > 0 \,\}$$

Testing

- Choose an input: $x = 5$
- Check pre-condition: $5 \geq 0$ ✓

Verification

Contracts in Testing and Verification

$$\{\, x \geq 0 \,\} \qquad C:\ y = x + 1 \qquad \{\, y > 0 \,\}$$

Testing

- Choose an input: $x = 5$
- Check pre-condition: $5 \geq 0$ ✓
- Execute: $y = 6$

Verification

Contracts in Testing and Verification

$$\{\, x \geq 0 \,\} \qquad C:\ y = x + 1 \qquad \{\, y > 0 \,\}$$

Testing

- Choose an input: $x = 5$
- Check pre-condition: $5 \geq 0$ ✓
- Execute: $y = 6$
- Check post-condition: $6 > 0$ ✓

Test passes

Verification

Contracts in Testing and Verification

$$\{\, x \geq 0 \,\} \qquad C:\ y = x + 1 \qquad \{\, y > 0 \,\}$$

Testing

- Choose an input: $x = 5$
- Check pre-condition: $5 \geq 0$ ✓
- Execute: $y = 6$
- Check post-condition: $6 > 0$ ✓

Test passes

Verification

- Assume $x \geq 0$

Contracts in Testing and Verification

$$\{\, x \geq 0 \,\} \qquad C:\ y = x + 1 \qquad \{\, y > 0 \,\}$$

Testing

- Choose an input: $x = 5$
- Check pre-condition: $5 \geq 0$ ✓
- Execute: $y = 6$
- Check post-condition: $6 > 0$ ✓

Test passes

Verification

- Assume $x \geq 0$
- Execute symbolically: $y = x + 1$

Contracts in Testing and Verification

$$\{\, x \geq 0 \,\} \qquad C:\ y = x + 1 \qquad \{\, y > 0 \,\}$$

Testing

- Choose an input: $x = 5$
- Check pre-condition: $5 \geq 0$ ✓
- Execute: $y = 6$
- Check post-condition: $6 > 0$ ✓

Test passes

Verification

- Assume $x \geq 0$
- Execute symbolically: $y = x + 1$
- Prove:

$$x \geq 0 \Rightarrow x + 1 > 0$$

Property proved

Weakest Pre-Condition

$$\{P\} \quad C \quad \{Q\}$$

A condition P such that:

If P is true before C, then Q is guaranteed after C

The **weakest pre-condition** is the *least restrictive* condition that guarantees Q after executing C.

Quiz: Pre-Conditions

Write the weakest pre-condition that prevents any out-of-bounds or null-pointer access in the following function.

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

Quiz: Pre-Conditions

Write the weakest pre-condition that prevents any out-of-bounds or null-pointer access in the following function.

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

The *weakest* pre-condition admits the largest set of inputs while still excluding every failing execution.

Solution: Pre-Conditions

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

```
A != nullptr && B != nullptr && lenA <= lenB
```

Solution: Pre-Conditions

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

```
A != nullptr && B != nullptr && lenA <= lenB
```

- Both pointers are dereferenced, so neither may be null

Solution: Pre-Conditions

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

```
A != nullptr && B != nullptr && lenA <= lenB
```

- Both pointers are dereferenced, so neither may be null
- For every cell of `A` the corresponding cell of `B` is read, so `B` must be at least as long as `A`

Quiz: Post-Conditions

A sorting function takes a non-null integer array `A`, places its elements in sorted order in an integer array `B`, and returns `B`. Which statements form the strongest post-condition that does not improperly reject a correctly sorted array?

1. `B` is non-null
2. `B` has the same length as `A`
3. the elements of `B` contain no duplicates
4. the elements of `B` are a permutation of the elements of `A`
5. the elements of `B` are in sorted order
6. the elements of `A` are in sorted order
7. the elements of `A` contain no duplicates

Solution: Post-Conditions

A sorting function takes a non-null integer array `A`, places its elements in sorted order in an integer array `B`, and returns `B`. Which statements form the strongest post-condition that does not improperly reject a correctly sorted array?

1. `B` is non-null
2. `B` has the same length as `A`
3. the elements of `B` contain no duplicates
4. the elements of `B` are a permutation of the elements of `A`
5. the elements of `B` are in sorted order
6. the elements of `A` are in sorted order
7. the elements of `A` contain no duplicates

Solution: Post-Conditions

A sorting function takes a non-null integer array `A`, places its elements in sorted order in an integer array `B`, and returns `B`. Which statements form the strongest post-condition that does not improperly reject a correctly sorted array?

1. `B` is non-null **Yes**
2. `B` has the same length as `A` **Yes**
3. the elements of `B` contain no duplicates
4. the elements of `B` are a permutation of the elements of `A`
5. the elements of `B` are in sorted order
6. the elements of `A` are in sorted order
7. the elements of `A` contain no duplicates

Solution: Post-Conditions

A sorting function takes a non-null integer array `A`, places its elements in sorted order in an integer array `B`, and returns `B`. Which statements form the strongest post-condition that does not improperly reject a correctly sorted array?

1. `B` is non-null — **Yes**
2. `B` has the same length as `A` — **Yes**
3. the elements of `B` contain no duplicates — **No**
4. the elements of `B` are a permutation of the elements of `A`
5. the elements of `B` are in sorted order
6. the elements of `A` are in sorted order
7. the elements of `A` contain no duplicates

Solution: Post-Conditions

A sorting function takes a non-null integer array `A`, places its elements in sorted order in an integer array `B`, and returns `B`. Which statements form the strongest post-condition that does not improperly reject a correctly sorted array?

1. `B` is non-null — **Yes**
2. `B` has the same length as `A` — **Yes**
3. the elements of `B` contain no duplicates — **No**
4. the elements of `B` are a permutation of the elements of `A` — **Yes**
5. the elements of `B` are in sorted order — **Yes**
6. the elements of `A` are in sorted order
7. the elements of `A` contain no duplicates

Solution: Post-Conditions

A sorting function takes a non-null integer array `A`, places its elements in sorted order in an integer array `B`, and returns `B`. Which statements form the strongest post-condition that does not improperly reject a correctly sorted array?

1. `B` is non-null — **Yes**
2. `B` has the same length as `A` — **Yes**
3. the elements of `B` contain no duplicates — **No**
4. the elements of `B` are a permutation of the elements of `A` — **Yes**
5. the elements of `B` are in sorted order — **Yes**
6. the elements of `A` are in sorted order — **No**
7. the elements of `A` contain no duplicates — **No**

Solution: Post-Conditions

A sorting function takes a non-null integer array `A`, places its elements in sorted order in an integer array `B`, and returns `B`. Which statements form the strongest post-condition that does not improperly reject a correctly sorted array?

1. `B` is non-null — **Yes**
2. `B` has the same length as `A` — **Yes**
3. the elements of `B` contain no duplicates — **No**
4. the elements of `B` are a permutation of the elements of `A` — **Yes**
5. the elements of `B` are in sorted order — **Yes**
6. the elements of `A` are in sorted order — **No**
7. the elements of `A` contain no duplicates — **No**

These four conditions constitute the entire specification of the output of a sorting function.

Writing the Post-Condition in Code

```
int[] B = sort(A);

assert B != null;
assert B.length == A.length;
for (int i = 0; i < B.length - 1; i++)
    assert B[i] <= B[i+1];
// Homework->permutation:  compare the element counts of A and B
```

Writing the Post-Condition in Code

```
int[] B = sort(A);

assert B != null;
assert B.length == A.length;
for (int i = 0; i < B.length - 1; i++)
    assert B[i] <= B[i+1];
// Homework->permutation:  compare the element counts of A and B
```

- The first three conditions are straightforward assertions

Writing the Post-Condition in Code

```
int[] B = sort(A);

assert B != null;
assert B.length == A.length;
for (int i = 0; i < B.length - 1; i++)
    assert B[i] <= B[i+1];
// Homework->permutation:  compare the element counts of A and B
```

- The first three conditions are straightforward assertions