Software Engineering VU [194.020]

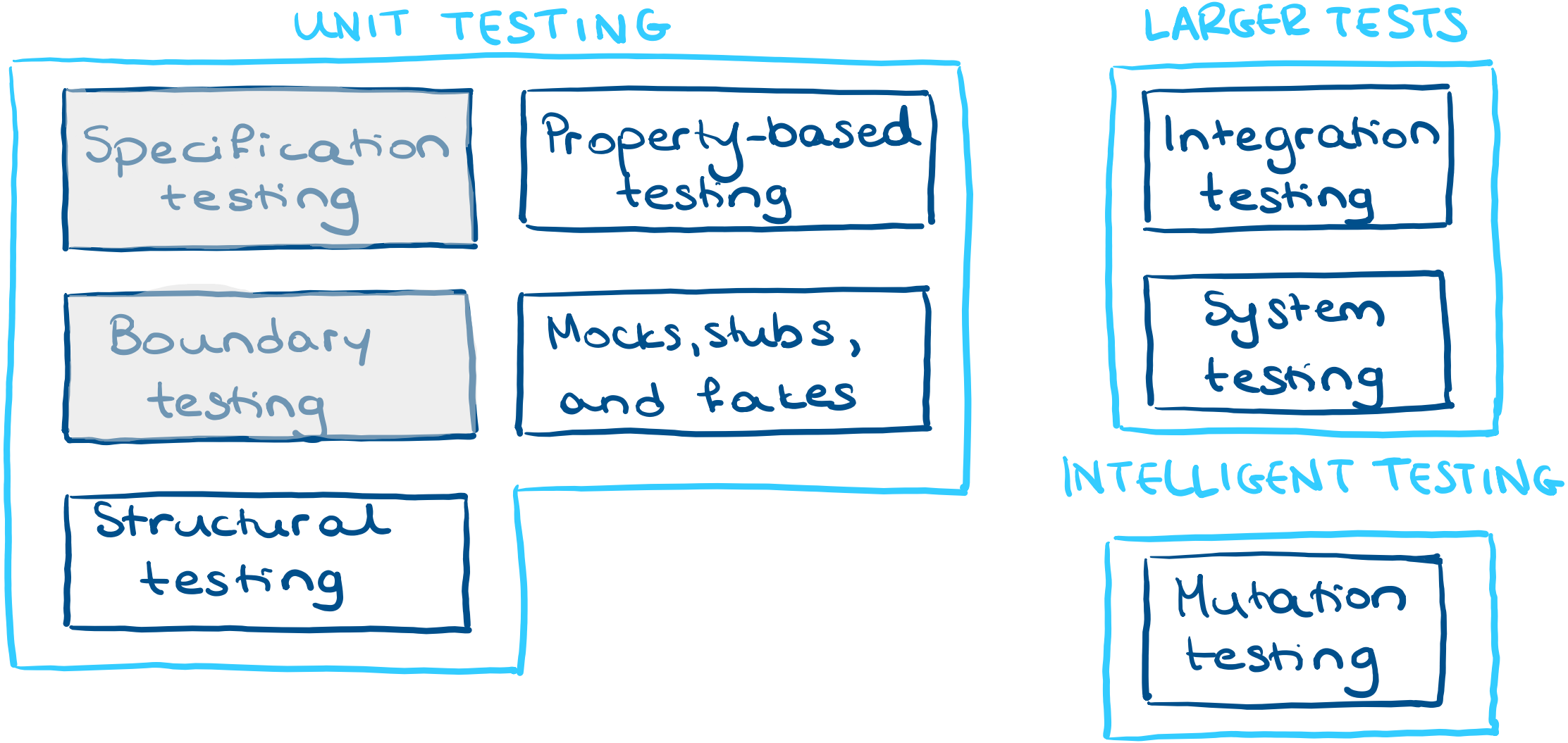
Maria Christakis

TU Wien

https://mariachris.github.io

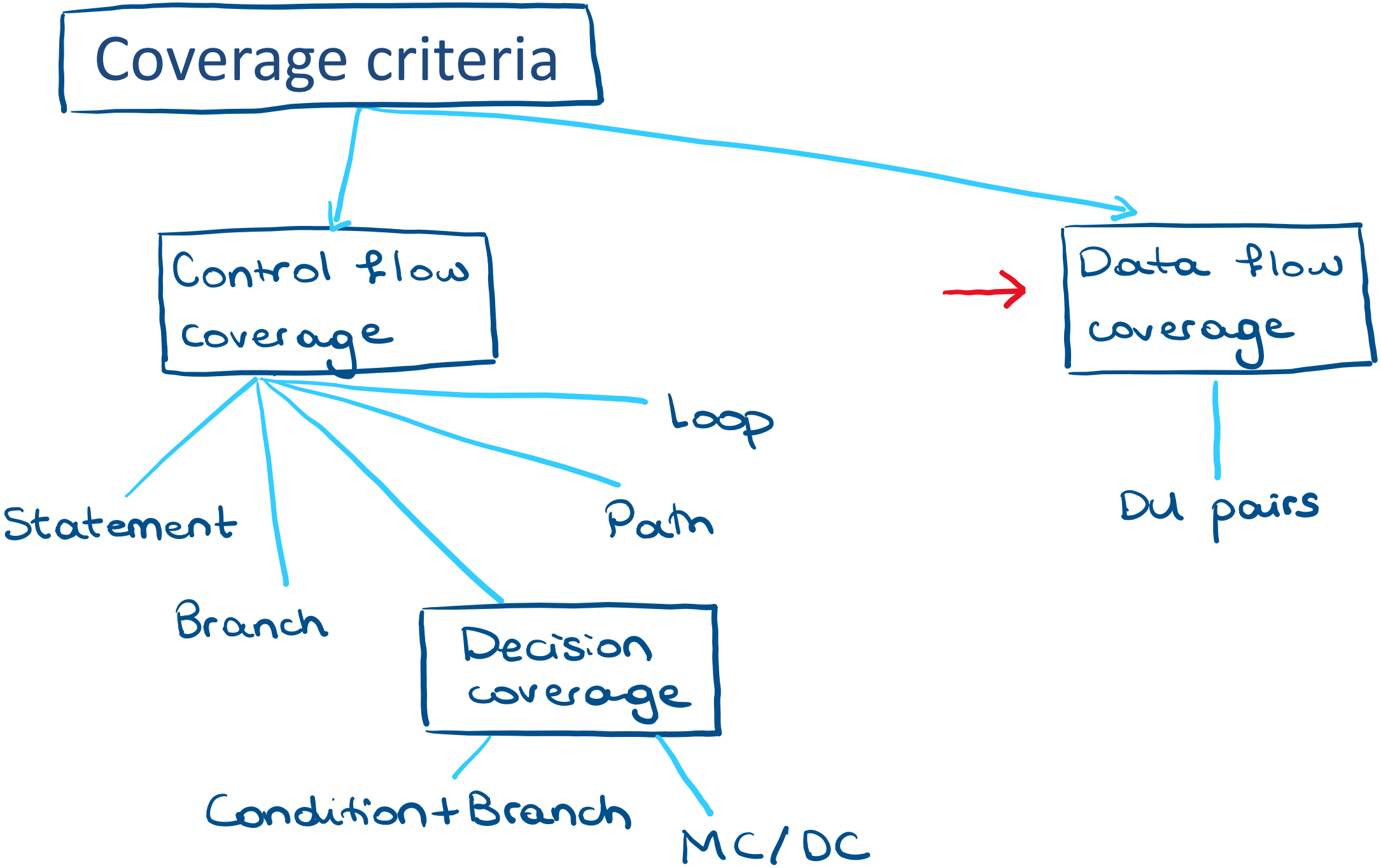
Summary

Effective and systematic testing



Testing

- Structural testing
  - Data flow coverage
- Property-based testing
- Test doubles and mocks


Coverage criteria
Control flow coverage
Data flow coverage
Loop
Statement
Path
DU pairs
Branch
Decision coverage
Condition+Branch
MC/DC

Data flow coverage

- Covering all program paths is infeasible
  - Their number grows exponentially in the number of branches
  - Loops
- Idea: Consider those paths where a computation in one part of the path affects the computation of another

Variable definition and use

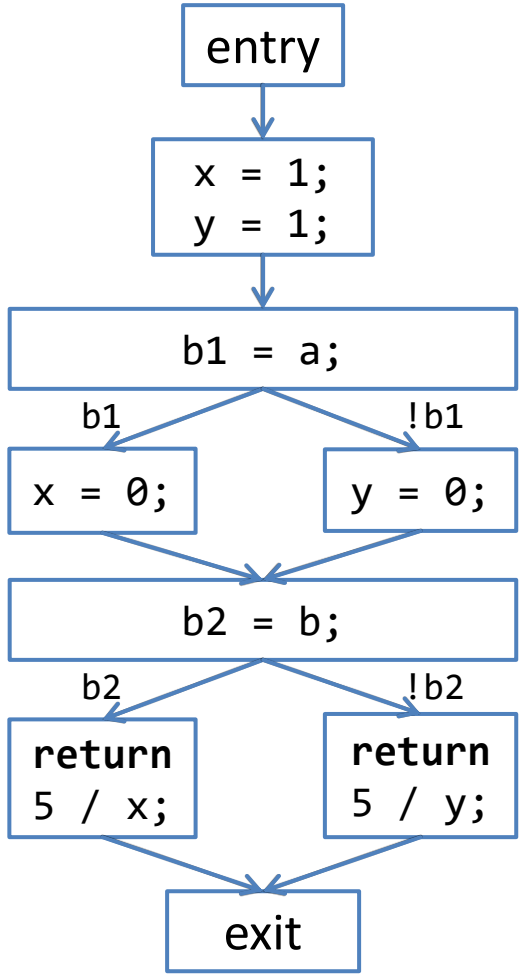
- A variable definition for a variable v is a basic block that assigns to v
  - v can be a local variable, formal parameter, object field, or array element
- A variable use for a variable v is a basic block that reads the value from v
  - In conditions, computations, output, etc.

Definition-clear paths

- A definition-clear path for a variable v is a path n_1, …, n_k in the control flow graph such that
  - n_1 is a variable definition for v
  - n_k is a variable use for v
  - No n_i, where $1 < i \leq k$, is a variable definition for v
- Note: Definition-clear paths do not necessarily go from entry to exit, in contrast to our earlier definition of path

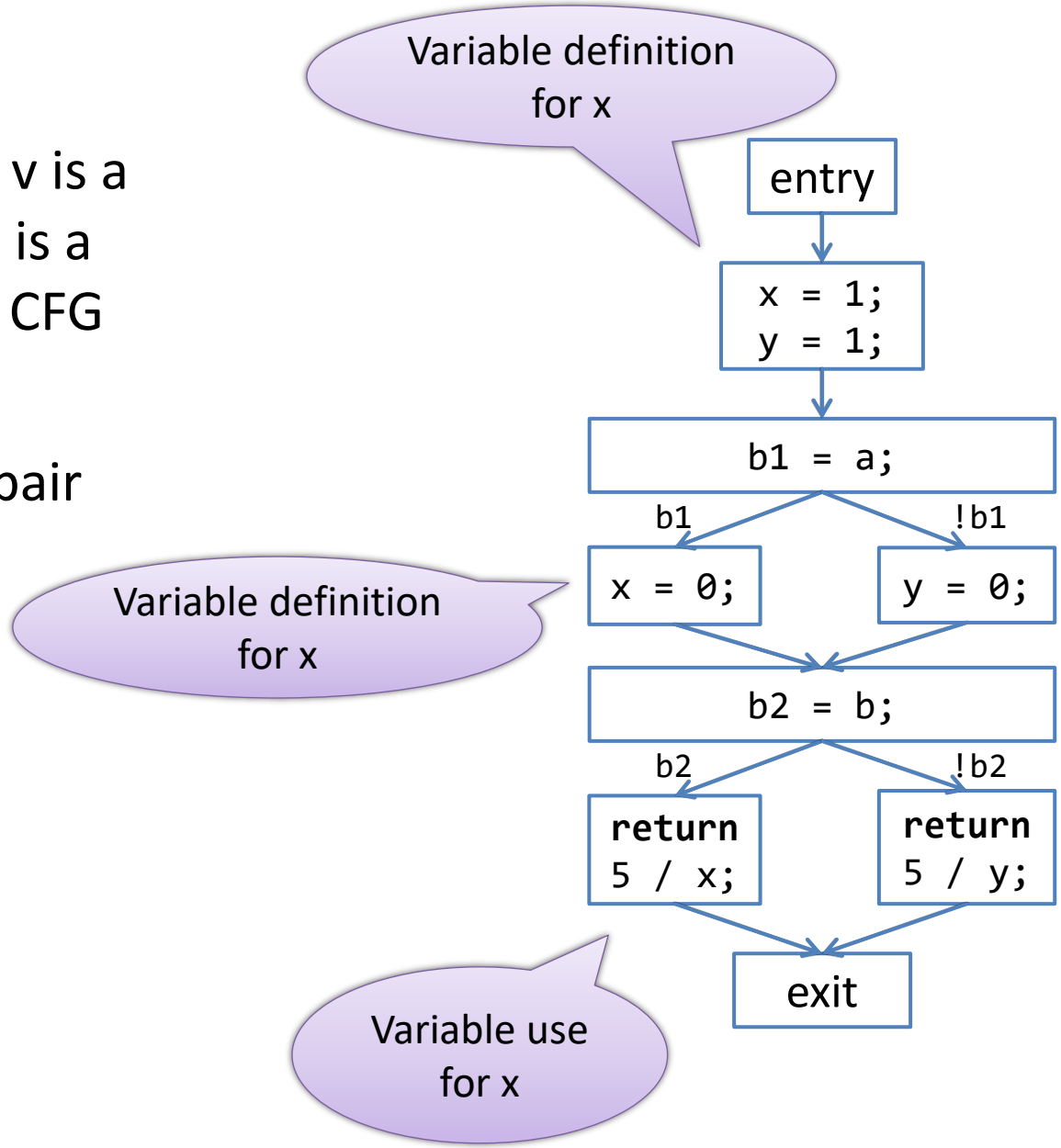
Example revisited

```
int Foo(boolean a, boolean b) {
  int x = 1;
  int y = 1;
  if (a)
    x = 0;
  else
    y = 0;
  if (b)
    return 5 / x;
  else
    return 5 / y;
}
```

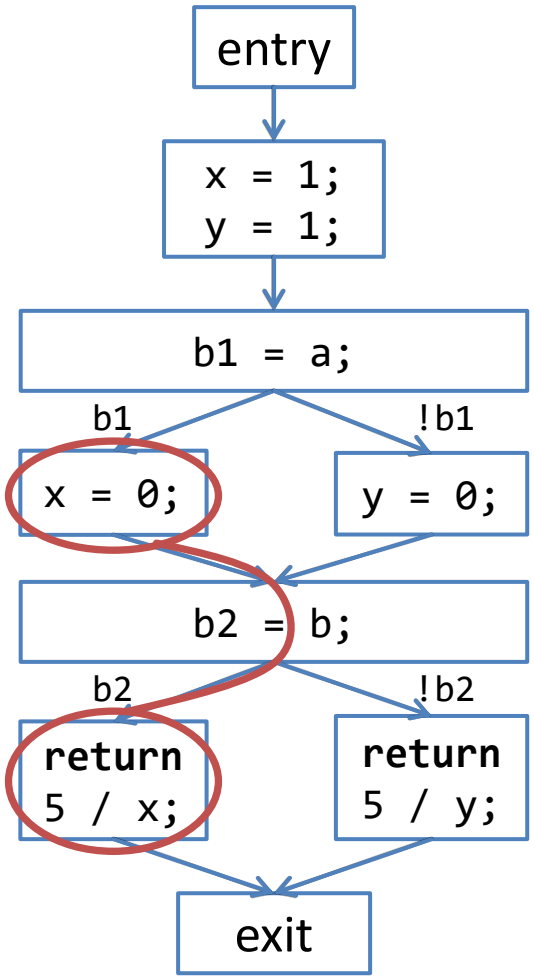


Definition-use pairs

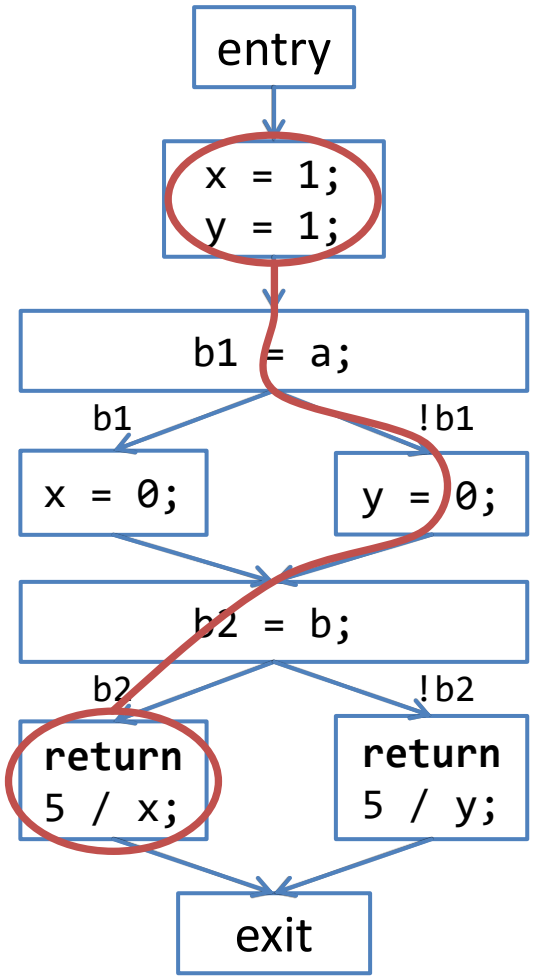
- A definition-use pair for a variable v is a pair of nodes (d,u) such that there is a definition-clear path d, …, u in the CFG
- We say DU pair for definition-use pair



Definition-use pairs – Example



Definition-use pairs – Example



DU-pairs coverage

$$\text{DU–pairs coverage} = \frac{\text{Number of executed DU pairs}}{\text{Total number of DU pairs}}$$

DU-pairs coverage – Example

- Consider the inputs

a = **true**, b = **false**

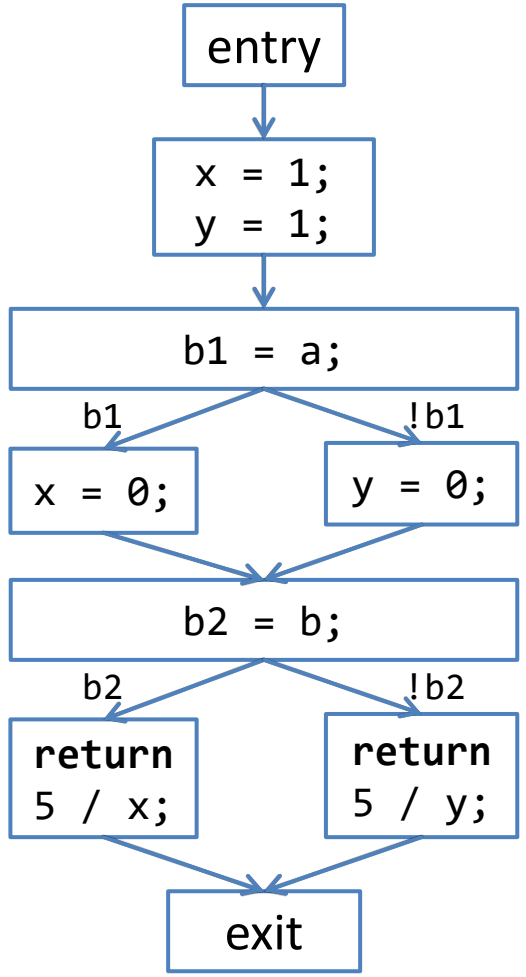
a = **false**, b = **true**

- What is the DU-pairs coverage?

a) 50% b) 70% c) 100%

(but 100% branch coverage)

- In this example, DU-pairs coverage is equivalent to path coverage



Determining all DU pairs

- DU pairs are computed with a reaching-definitions static analysis

- Algorithm: For each node n and for each variable v, compute all variable definitions for v that possibly reach n via a definition-clear path

- The reaching definitions at a node n are:
  - The reaching definitions of n's predecessors in the CFG
  - Minus the definitions killed by one of n's predecessors
  - Plus the definitions made by one of n's predecessors

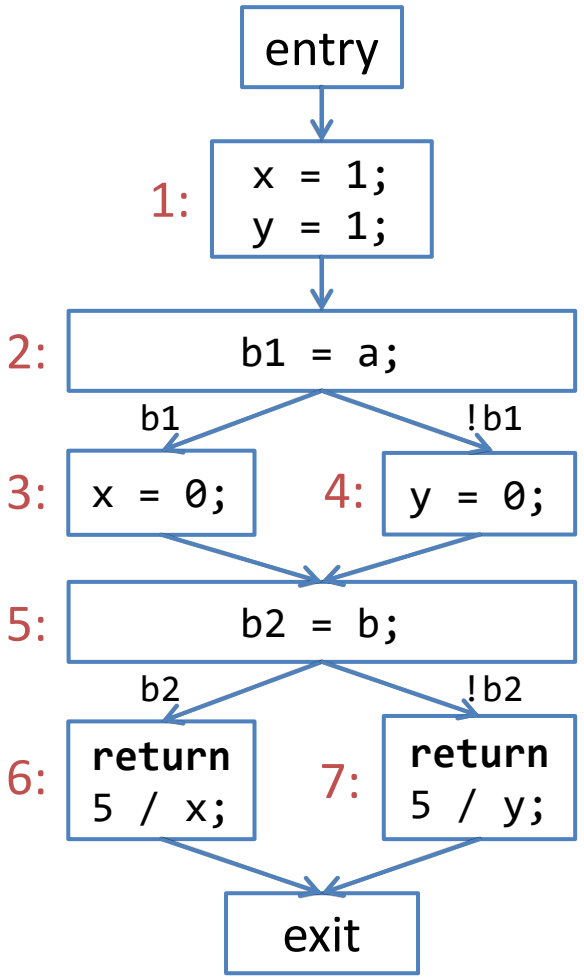
Reaching definitions – Algorithm

- We use the sets
  - pred(n) = {m | (m,n,c) is an edge in the CFG}
  - succ(m) = {n | (m,n,c) is an edge in the CFG}
  - gen(n) = {v_n | n is a variable definition for v}
  - kill(n) = {v_m | n is a variable definition for v and m ≠ n}
- We compute via fixpoint iteration
  - Reach(n): The reaching definitions at the beginning of n
  - ReachOut(n): The reaching definitions at the end of n

Reaching definitions – Algorithm

```
foreach node n do ReachOut(n) := ∅ end
worklist := nodes
while worklist ≠ ∅ do
  n := any(worklist)
  remove n from worklist
  Reach(n) := ⋃_{m ∈ pred(n)} ReachOut(m)
  ReachOut(n) := Reach(n) \ kill(n) ∪ gen(n)
  if ReachOut(n) has changed then
    worklist := worklist ∪ succ(n)
  end
end
```

Reaching definitions – Example




```
foreach node n do ReachOut(n) := ⦰ end
worklist := nodes
while worklist ≠ ⦰ do
  n := any(worklist)
  remove n from worklist
  Reach(n) := U_{m ∈ pred(n)} ReachOut(m)
  ReachOut(n) := Reach(n) \ kill(n) U gen(n)
  if ReachOut(n) has changed then
    worklist := worklist U succ(n)
  end
end
```

n	Reach(n)	ReachOut(n)
1	⦰	x_1, y_1
2	x_1, y_1	x_1, y_1
3	x_1, y_1	x_3, y_1
4	x_1, y_1	x_1, y_4
5	x_1, x_3, y_1, y_4	x_1, x_3, y_1, y_4
6	x_1, x_3, y_1, y_4	x_1, x_3, y_1, y_4
7	x_1, x_3, y_1, y_4	x_1, x_3, y_1, y_4

Reaching definitions to DU pairs

- The set of DU pairs is easily determined as

{(d,u) | u is a variable use for v and $v_d \in$ Reach

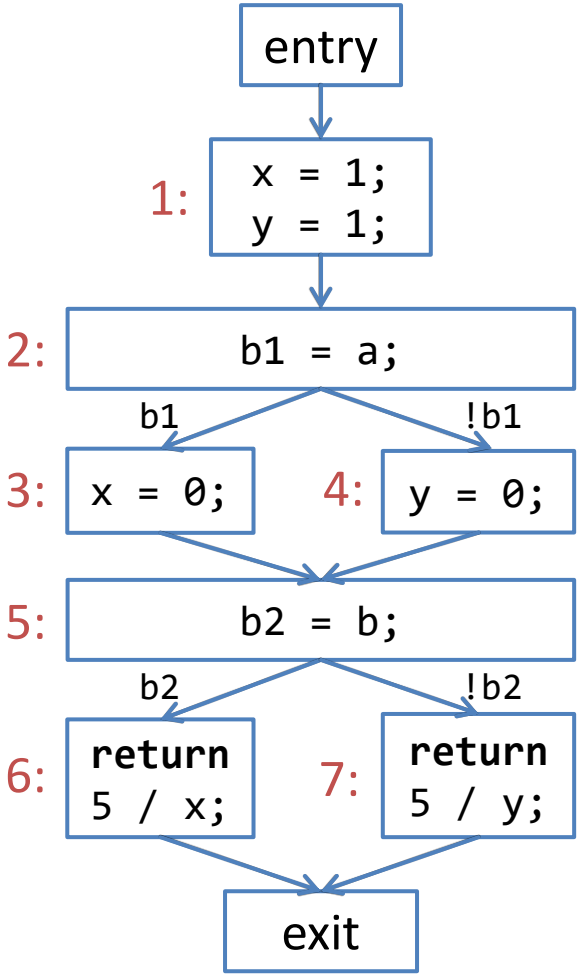
n	Reach(n)
1	$\varnothing$
2	x_1, y_1
3	x_1, y_1
4	x_1, y_1
5	x_1, x_3, y_1, y_4
6	x_1, x_3, y_1, y_4
7	x_1, x_3, y_1, y_4

- DU pairs of x:

(1,6), (3,6)

- DU pairs of y:

(1,7), (4,7)



DU-pairs coverage – Discussion

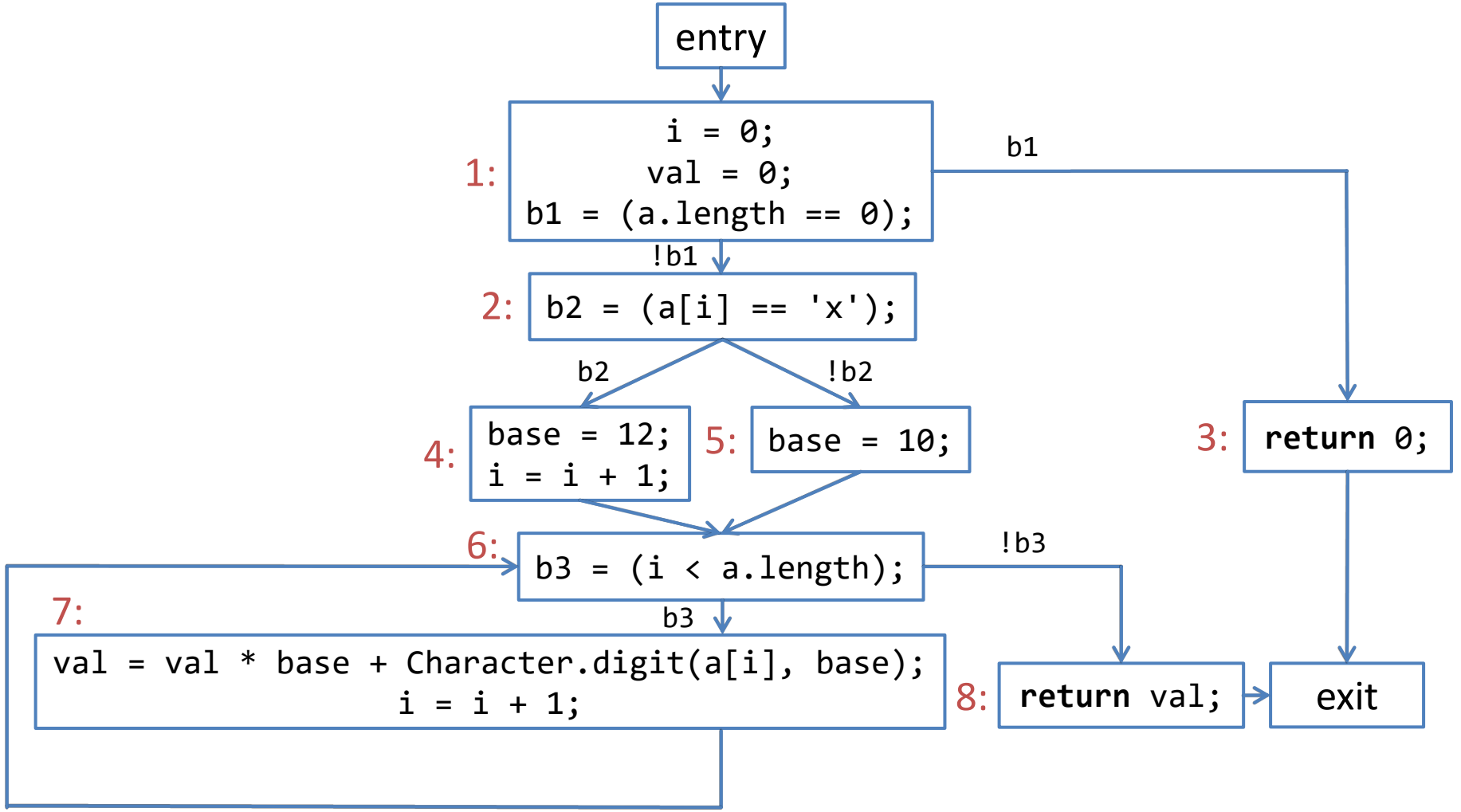
- Convert character sequence to integer
  - Input format: d_{dec}* | ‘x’(d_{hex}*)

```
static int Convert(char[] a) {
  int base; int i = 0; int val = 0;
  if (a.length == 0) return 0;
  if (a[i] == 'x') { base = 12; i = i + 1; }
  else { base = 10; }
  while (i < a.length) {
    val = val * base + Character.digit(a[i], base);
    i = i + 1;
  }
  return val;
}
```

We assume that all inputs are of the right format

Is this method correct?

DU-pairs coverage – Discussion



DU-pairs coverage – Discussion

n	Reach(n)	ReachOut(n)
1	⦰	i_1, val_1
2	i_1, val_1	i_1, val_1
3	i_1, val_1	i_1, val_1
4	i_1, val_1	i_4, val_1, $base_4$
5	i_1, val_1	i_1, val_1, $base_5$
6	i_1, i_4, i_7, val_1, val_7, $base_4$, $base_5$	i_1, i_4, i_7, val_1, val_7, $base_4$, $base_5$
7	i_1, i_4, i_7, val_1, val_7, $base_4$, $base_5$	i_7, val_7, $base_4$, $base_5$
8	i_1, i_4, i_7, val_1, val_7, $base_4$, $base_5$	i_1, i_4, i_7, val_1, val_7, $base_4$, $base_5$

- We get 14 DU pairs
- DU pairs for i:

  (1,2), (1,4), (1,6), (1,7), (4,6), (4,7), (7,6), (7,7)
- DU pairs for val:

  (1,7), (1,8), (7,7), (7,8)
- DU pairs for base:

  (4,7), (5,7)

DU-pairs coverage – Discussion

- Consider the inputs

a = {}

a = {‘x’}

a = {‘1’}

a = {‘1’,‘2’}

```
static int Convert(char[] a) {
  int base; int i = 0; int val = 0;
  if (a.length == 0) return 0;
  if (a[i] == 'x') { base = 12; i = i + 1; }
  else { base = 10; }
  while (i < a.length) {
    val = val * base + Character.digit(a[i], base);
    i = i + 1;
  }
  return val;
}
```

- The bug is missed
- Branch and loop coverage: 100%
- DU pairs missed: (4,7) for i, base (coverage 86%)

DU-pairs coverage – Discussion

- DU pairs for i and val include (7,7)

- Complete DU-pairs coverage requires more than one loop iteration

```
static int Convert(char[] a) {
  int base; int i = 0; int val = 0;
  if (a.length == 0) return 0;
  if (a[i] == 'x') { base = 12; i = i + 1; }
  else { base = 10; }
  while (i < a.length) {
    val = val * base + Character.digit(a[i], base);
    i = i + 1;
  }
  return val;
}
```

Determining all DU pairs

- A static analysis would need arithmetic and aliasing information to determine whether a definition and use refer to the same heap location
- Static analysis has to over-approximate

```
static void Repeat(int[] from, int[] to) {
  int i = 0;
  if (from.length == 0) return;
  while (i < to.length) {
    to[i] = to[i] + from[i % from.length];
    i = i + 1;
  }
}
```

Measuring DU-pairs coverage

- Keep track of currently active definitions
  - defCover: Variable → Block
- Keep track of executed DU pairs
  - useCover: Variable × $\text{Block}_{\text{def}}$ × $\text{Block}_{\text{use}}$ → $\mathbb{N}$
- Maps can be encoded as arrays, indexed by identifiers for variables and basic blocks

Measuring DU-pairs coverage

```
int Foo(boolean a, boolean b) {
  int x = 1; defCover["x"] = 1;
  int y = 1; defCover["y"] = 1;
  if (a)
    x = 0; defCover["x"] = 3;
  else
    y = 0; defCover["y"] = 4;
  if (b)
    useCover["x", defCover["x"], 6]++;
    return 5 / x;
  else
    useCover["y", defCover["y"], 7]++;
    return 5 / y;
}
```

Current variable definition for x is basic block 1

Current variable definition for x is basic block 3

DU pair for variable x with current definition and use-block 6 has been executed

Data flow coverage – Discussion

- Data flow coverage complements control flow coverage
  - For example, choose tests that maximize branch and DU-pairs coverage
- Not all DU pairs are always feasible
  - Static analysis over-approximates data flow
- DU-pair “anomalies” may point to errors
  - Double definition without use, or termination after definition without use, etc.

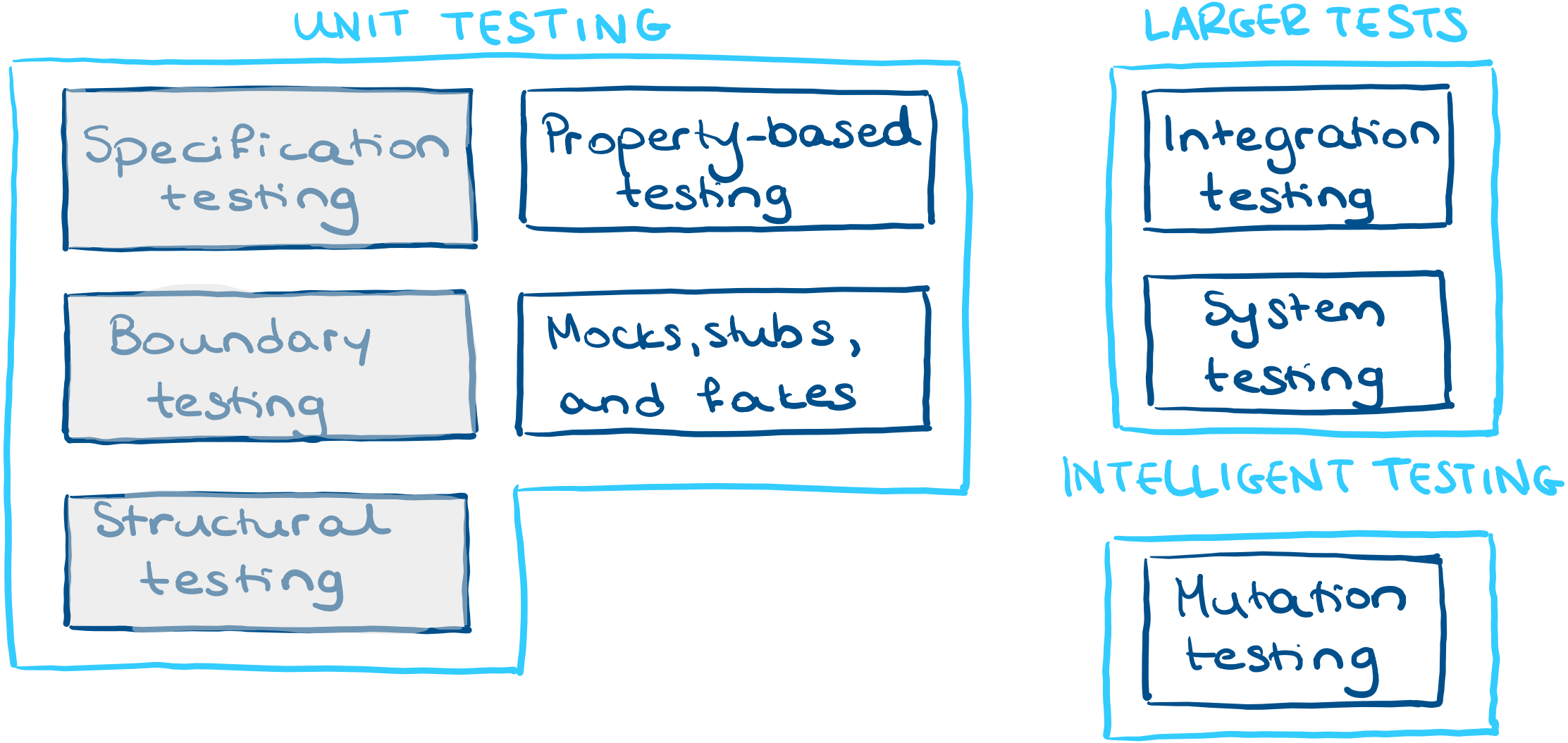
Test coverage – Discussion

- High coverage does not mean that code is well tested
  - But low coverage means that code is not well tested
- How well tested the code is depends on the coverage criterion
- Full coverage for stronger criteria implies larger test suites

Testing

- Structural testing
  - Data flow coverage
- Property-based testing
- Test doubles and mocks

Effective and systematic testing



Example- vs property-based testing

- Example-based testing is picking one concrete example from all the possible ones and writing a test case
  - E.g., specification-based testing is example-based testing
- Property-based testing is defining a property (or set of properties) that the program should satisfy and letting the test framework choose several examples
  - The goal of the test framework is to find a counterexample that causes the program to violate the properties

Example: The passing grade program

A student passes an exam if they get a grade ≥ 5. Grades below that are a fail. Grades fall in the range [1.0, 10.0].

Jqwik: A property-based testing framework

- Pronounce it “jay quick”

- Check it out here: https://jqwik.net/

Property-based testing: Pros and cons

\+ Explores the input domain much better

\- Is more complex than example-based testing

\- Requires more creativity and practice to automate

Common issues in property-based tests

1. Generating data may be expensive or impossible

- E.g., generating an array of 100 elements in which the numbers must be unique and multiples of 2, 3, 5, and 15

- Or generating an array of 10 unique elements, from a range of 2 to 8

Common issues in property-based tests

2. Failing to express the boundaries of a property

- Property testing frameworks mix edge cases with random data points, only if properties are expressed correctly

- E.g., `Arbitraries.floats().lessThan(1f)`

Common issues in property-based tests

3. Ensuring the input data passed to the method under test is fairly distributed among all the possible options

- Property testing frameworks do their best to generate well distributed inputs, e.g., when asking for an integer between 0 and 10, all the numbers have the same probability of being generated

- But tests that manipulate the generated data can harm this property

Common issues in property-based tests

3. Ensuring the input data passed to the method under test is fairly distributed among all the possible options

- Property testing frameworks do their best to generate well distributed inputs, e.g., when asking for an integer between 0 and 10, all the numbers have the same probability of being generated

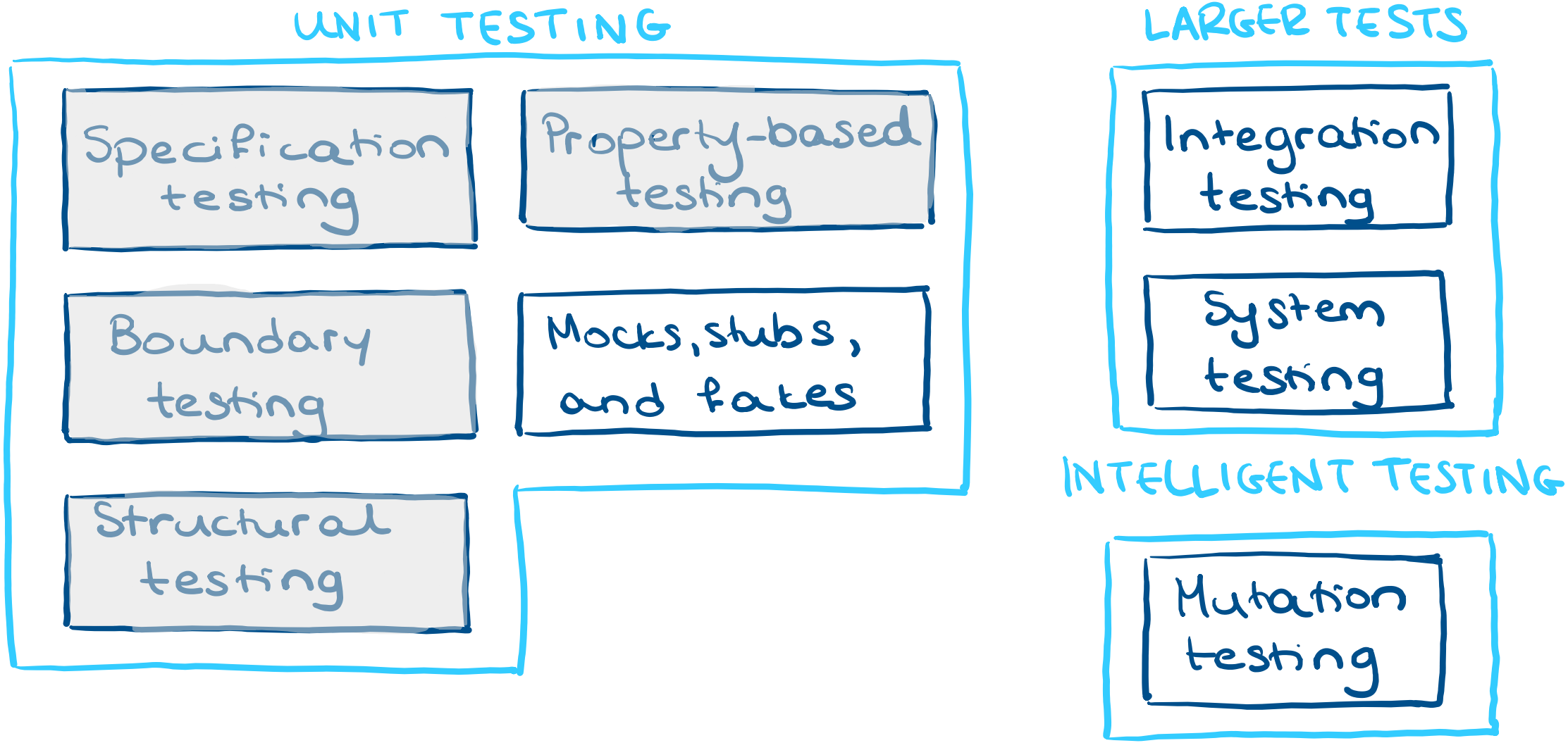
- But tests that manipulate the generated data can harm this property

```
@Property
void gradesBadTest(@ForAll @FloatRange(max = 100f) float grade) {
  // ... test here ...
}
```

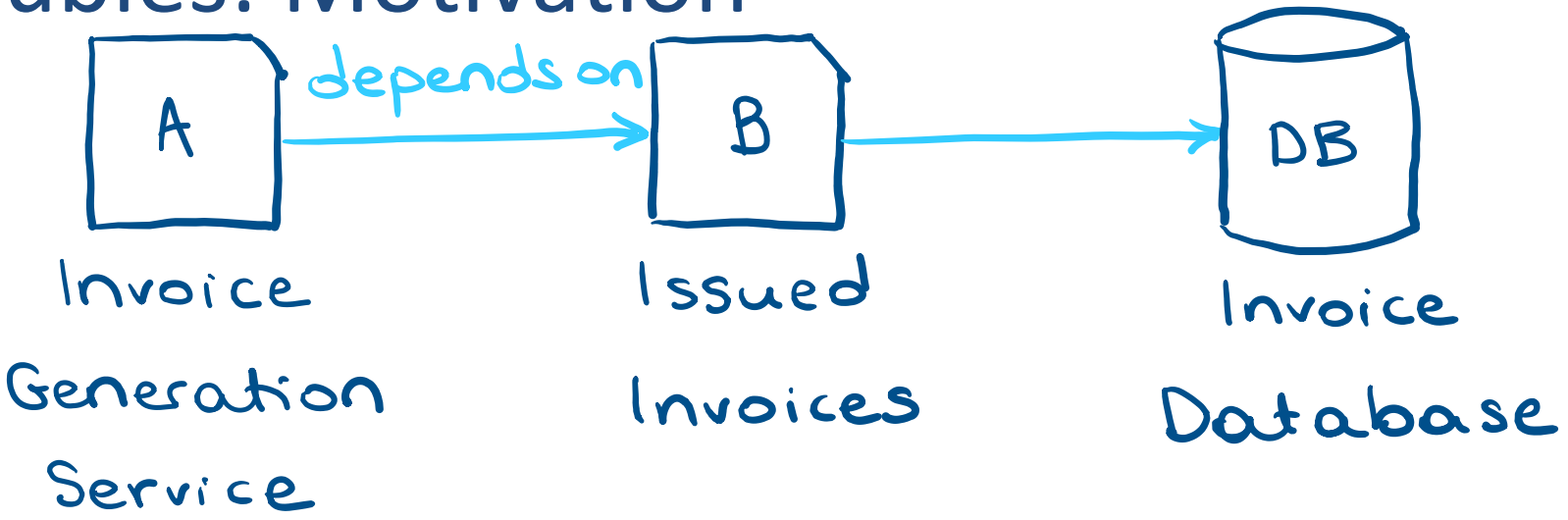
Testing

- Property-based testing
- Test doubles and mocks

Effective and systematic testing



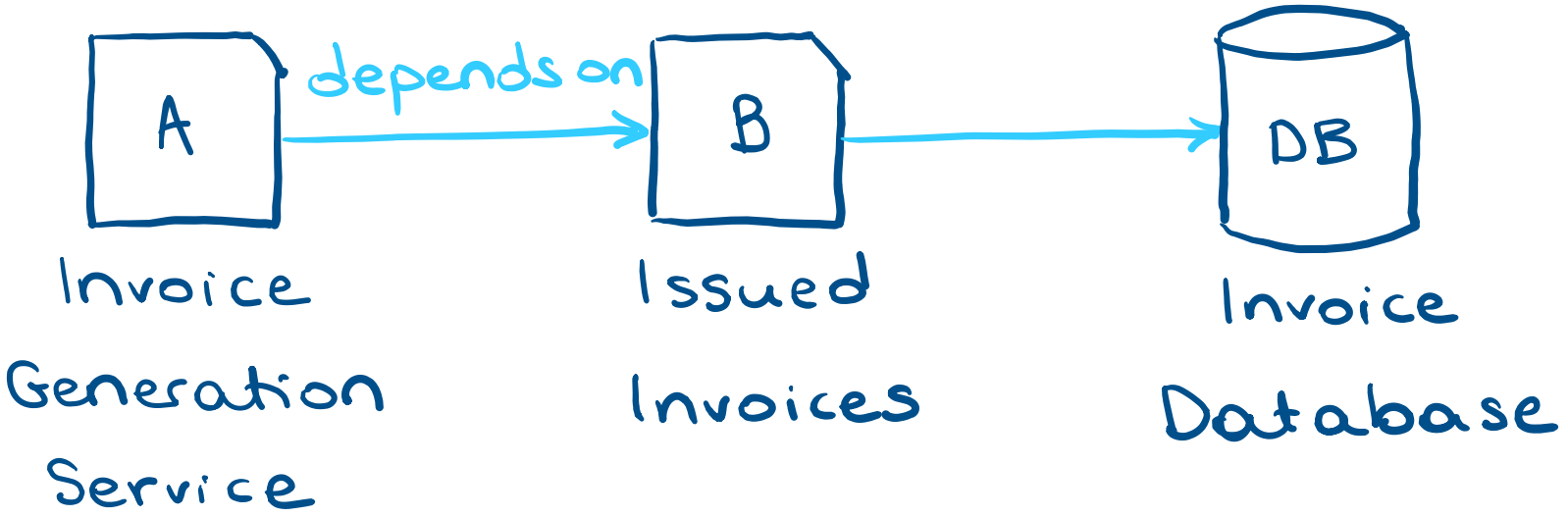
Test doubles: Motivation



- Testing class A together with its dependencies might be too slow, too hard, or too much work
  - E.g., when testing the Invoice Generation Service, maybe we do not want to test whether the SQL query in the Issued Invoices is correct
  - We only want to ensure the invoice is generated correctly
  - Testing the SQL queries is the responsibility of the Issued Invoices test suite

Test doubles: Definition

- A test double is an object that mimics the behavior of a software component for testing purposes
  - Within a test, we have full control over what the double does, and thus cut the dependency on the real object
  - E.g., we can implement a fake Issued Invoices class that returns a hard-coded list of values rather than retrieving them from an external database



Test doubles: Pros

- We have more control
  - We can easily tell these doubles what to do, e.g., to throw an exception or return a fake date
- Simulations are faster
  - Imagine dependencies with a web service or a database, where each query may take a few seconds to process
- Doubles enable developers to think about how classes should interact

Test doubles: 5 types

- Dummy objects
- Fake objects
- Stubs
- Mocks
- Spies

Dummy objects

- Dummy objects are passed to the class under test but never used
  - Common when you need to fill a long list of parameters, but the test exercises only a few of them
  - E.g., think of a Customer class that depends on Address, Email, and so on:

  When exercising a behavior that does not care about a customer's address, we can set up a dummy Address object and pass it to the Customer class

Fake objects

- Fake objects are real working implementations of the class they simulate, but they usually do the same task in a much simpler way
  - E.g., think of a fake database class that uses an array list instead of a real database

Stubs

- Stubs provide hard-coded answers to the calls performed during the test

  - Unlike fake objects, stubs do not have a working implementation:

  If the code calls a stubbed method to get all invoices, the stub will return a hard-coded list of invoices, e.g., we could create a stub that returns an empty list, one that returns a list with one invoice, and another that returns a list with many invoices

  - Stubs are the most popular type of simulation

Mocks

- Mocks act like stubs and also save all interactions with the mocked object
  - E.g., imagine that we want to check the calls to a method that gets all invoices:

  Mocks allow asserting that the method is only called once, that it is never called with a specific parameter, or that it is called twice with parameter A and once with B

  - Mocks are also very popular

Spies

- Spies do not simulate the underlying object but only record all the interactions with it
  - Spies wrap themselves around the real object
  - They are used in very specific contexts, e.g., when it is much easier to use the real implementation than a mock, but we still want to check the interactions with the underlying object

Example: The invoice filter program

The program must return all the issued invoices with values smaller than 100. The collection of invoices can be found in the database. The class `IssuedInvoices` already contains a method that retrieves all the invoices.

We will stub `IssuedInvoices`.

Mockito

- Check it out here: https://site.mockito.org/

Stub demo: Observations

- The test was easier to write
- The test class is less likely to change if something other than `InvoiceFilter` changes
  - We are not testing `IssuedInvoices`
  - If the contracts of `IssuedInvoices` change, then we may have to propagate the changes to our stub
- The test can only fail because of a bug in `InvoiceFilter`

Example: The SAP invoice sender program

Our current system has an additional requirement:

All low-valued invoices should be sent to our SAP system (a software that manages business operations). SAP offers a send web service for invoices.

We will stub `InvoiceFilter` and mock SAP.

Disadvantages of stubs/mocks

- They make the code less realistic: in production, the code uses the concrete implementation of the stubbed/mocked class
  - E.g., when changing the implementation of `issuedInvoices.all()` or `sap.send()`, the developer will update the tests of the `IssuedInvoices` or `SAP` class
  - But it is easy to forget to update the tests of `InvoiceFilter` or `SAPInvoiceSender` especially in large-scale software
  - For stubs/mocks to work well on a large scale, developers must design stable contracts
  - When contracts do change, it is part of the developer's job to find all dependencies

Disadvantages of stubs/mocks

Key point: You need to keep them up to date

- The tests are more coupled with the code they test
  - Tests without stubs/mocks typically call a method and assert the output – they don't know anything about the method's implementation
  - The test we wrote for `SAPInvoiceSender` knows about both `filter.lowValueInvoices()` and `sap.send()`
  - When tests know so much, they are harder to change
  - So, although stubs/mocks simplify tests, they increase coupling between test and production code, which may force us to change them whenever we change the production code

What to stub/mock

- Dependencies that are too slow, e.g., web services or databases
  - We don’t want slow test suites
- Dependencies that depend on external infrastructure
  - External infrastructure might be too slow or too complex to set up, e.g., think of the `IssuedInvoices` class
- Cases that are hard to simulate
  - E.g., when we would like the dependency to throw an exception
- Dependencies that have non-deterministic behavior
  - We don’t want flaky tests suites – a flaky test yields both passing and failing results without any changes to the code

Suggested reading

From Book: "Effective Software Testing A Developer's Guide" by M. Aniche

- Introduction of Chapter 5
- Sections 5.1, 5.6
- Introduction of Chapter 6
- Sections 6.1, 6.2, 6.2.1, 6.2.2, 6.3.1, 6.3.2