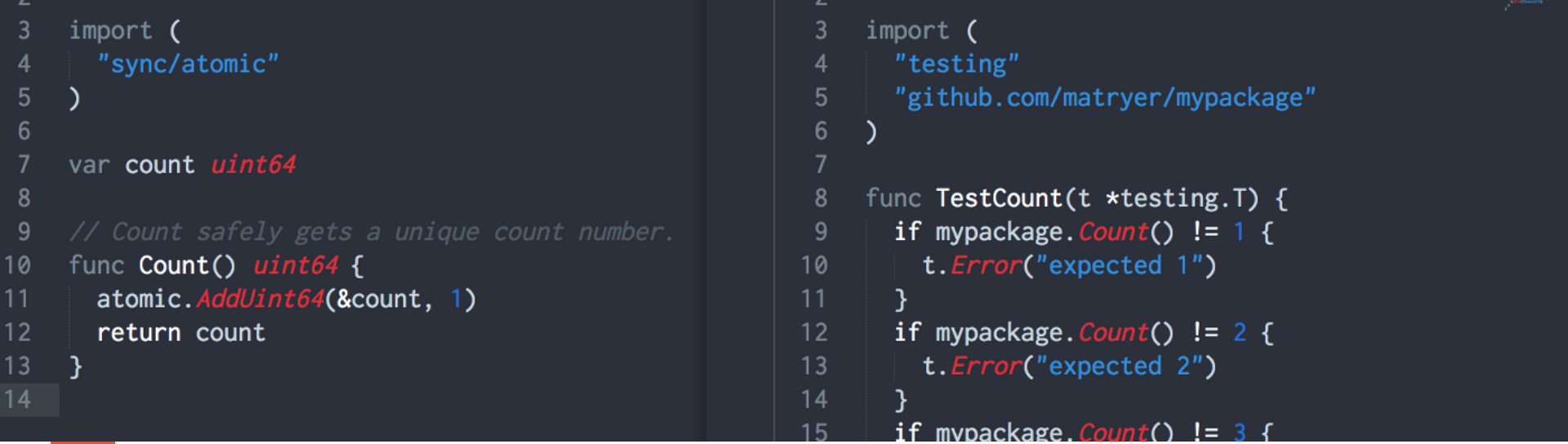


Software Testing for Continuous Delivery

Seminar 6.5: Specification-based Testing (theory) - CON'T

Dr. Byron J. Williams
September 13, 2019

UF Herbert Wertheim College of Engineering
Department of Computer & Information Science & Engineering
UNIVERSITY *of* FLORIDA

Content adapted from “Developer Testing” by Alexander Tarlinder 1st Edition (2016)
Images courtesy Google Image Search

FICS Research

UF Herbert Wertheim College of Engineering
UNIVERSITY of FLORIDA

POWERING THE NEW ENGINEER TO TRANSFORM THE FUTURE

DEPARTMENT OF COMPUTER & INFORMATION SCIENCE & ENGINEERING
FLORIDA INSTITUTE FOR CYBERSECURITY RESEARCH (FICS)
Dr. Byron J. Williams

Specification-based Testing

- Okay, I know how to write a unit test, but what should I test?
- How many tests should I write?
- Have I thoroughly tested this unit? How can I show this?
- How can a developer think like a tester (practically vs. theoretically)?

Writing Tests

- **Order of Tests**
  1. **Degenerate case**—Start with a test that operates on an "empty" value like zero, null, the empty string, or the like.
  2. **Happy path tests**—lay the foundation for the implementation while remaining focused on the core functionality
  3. **Provide new information or knowledge**—Try approaching the solution from different angles and exercising different parts
  4. **Error handling and negative tests**—These tests are crucial to correctness - in many cases, they can safely be written last.

- **Red to Green Strategies**
  - **Faking**—This is the simplest way to make a test pass. Just return or do whatever the particular test expects. If the test expects a specific value, then just hand it over
  - **The obvious implementation**—Sometimes beating about the bush just isn't worth it. If we know what to type, then we should type it.
  - **Triangulation** - Some algorithms can be teased out by providing a number of examples and then generalizing

RECAP

Equivalence Partitioning

- Concept from partition-based testing - basic ideas:
  - Members in equivalence class are treated as **“equivalent”**
  - Defining meaningful partitions — sampling from partitioned subsets for different types of partitions
  - Coverage of partitions: uniform
- Ex: Integer-based calculator - all integers are in the same equivalence partition or ***equivalence class***, are subsets of data in which all values are equivalent to each other

Testing for Partition Coverage

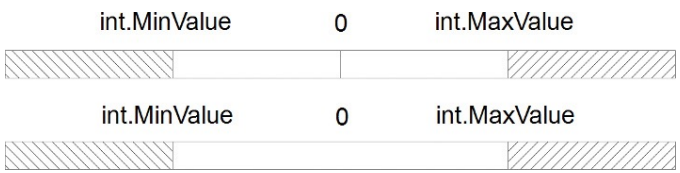
Sensitize test cases

- i.e., defining specific input variables and associated values to exercise certain parts of the program in the white-box view or to perform certain functions in the black-box view
- e.g., function `add(int a, int b)`
- considering valid/invalid input values of *a* and *b*
- How many cases are in exhaustive test?

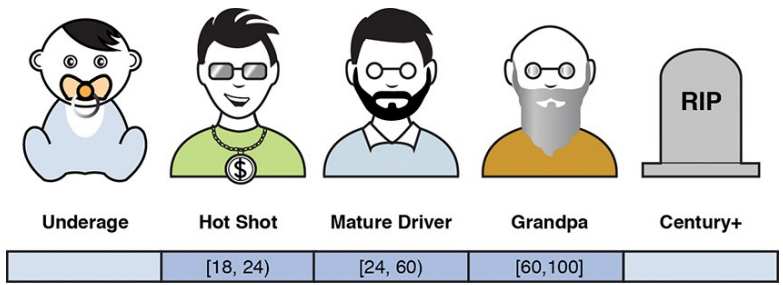
Test Case	Condition		Input	
	int a	***int b***	***a***	***b***
1	False	False	3.2	-0.4
2	False	True	"MSU"	2
3	True	False	7	3/4
4	True	True	-9	-2

Domain Equivalence

- A **developer** would know the **range** of a data type and base the partitioning on that…
- Whereas the **tester** would probably think more about the **domain** and partition from that viewpoint. This could lead to different partitioning
- Near endless possibilities, and it's the **specification** and **test scenario** that should guide the choice of relevant equivalence partitions




```
public double getPremiumFactor(int age) {
   if (age >= 18 && age < 24) {
      return 1.75;
   } else if (age >= 24 && age < 60) {
      return 1;
   } else if (age >= 60) {
      return 1.35;
   }
}
```



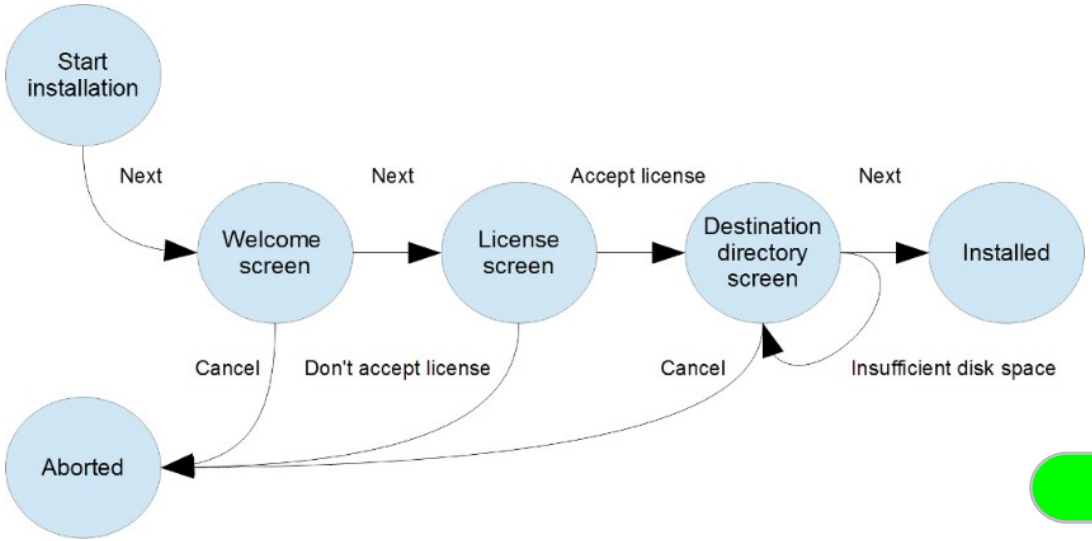
RECAP

Boundary Value Analysis

- ***Values that occur at the boundary of an equivalence partition***
- If no equivalence partitions have been identified, think of boundary values as occurring at the edges of **the domain of *allowed* input**
  - often called **edge cases**
- **Derived**: from the spec; the size of a data type; and/or common sense
- Developers have both access to a **specification** and **knowledge** about the **ranges of the data types** they are using.
  - no excuses for not checking potential edge cases

State Transition Testing

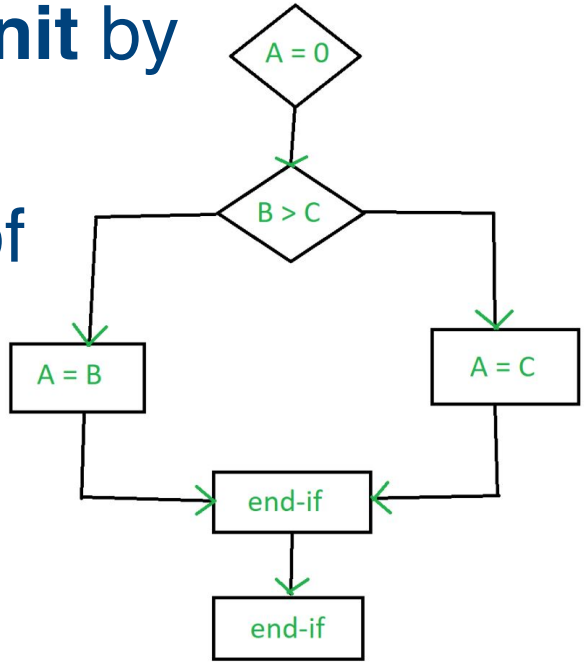
- Certain applications or components of a system can be modeled as state machines (e.g., installation wizards, page navigation, control systems, etc)



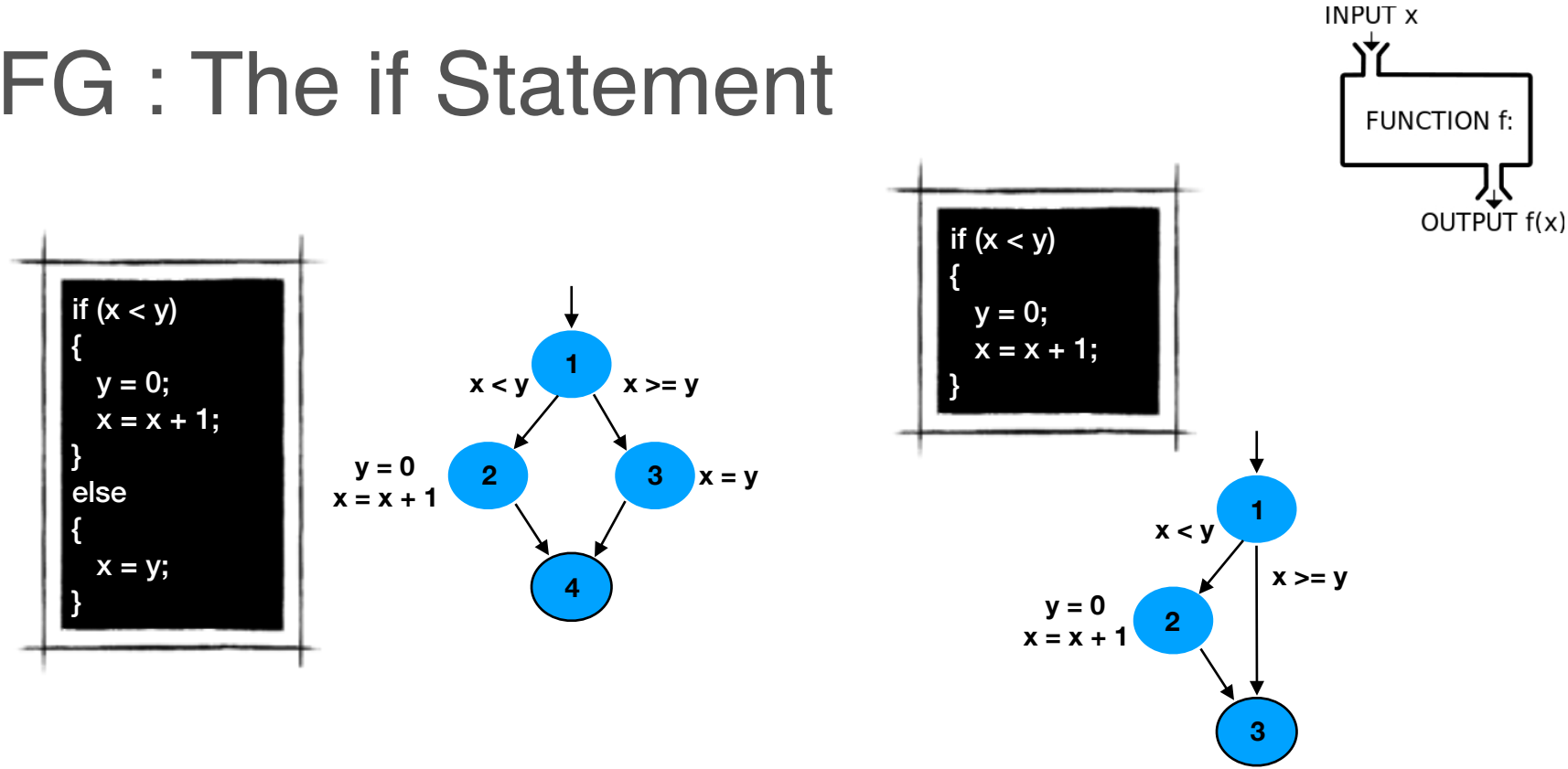
State diagrams can be drawn at **different levels of abstraction.** That's probably the greatest strength of this technique.

Control Flow Graphs / Testing

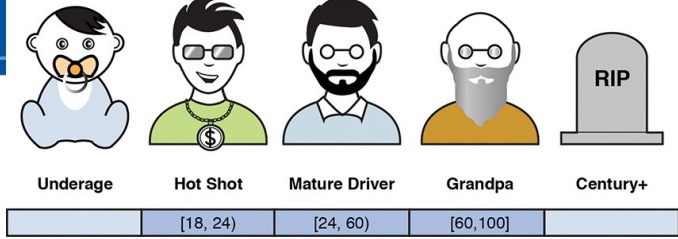
- A CFG models all executions of a **unit** by describing **control structures**
- Nodes : Statements or sequences of statements (basic blocks)
- Edges : Transfers of control



CFG : The if Statement



Decision Tables



- Premium is also affected by the driver's gender — statistics show women to be safer drivers
- Can indicate ***business rules*** using decision tables:
  - capture all combinations of variables and possible outcomes
- Often inputs to automated acceptance testing tools (e.g., FitNesse, Concordion, or Cucumber) for parameterized tests

Age	18–23	18–23	24–59	24–59	60+	60+
Gender	Male	Female	Male	Female	Male	Female
Premium factor 1	N	N	N	Y	N	N
Premium factor 1.05	N	N	Y	N	N	N
Premium factor 1.25	N	N	N	N	N	Y
Premium factor 1.35	N	N	N	N	Y	N
Premium factor 1.65	N	Y	N	N	N	N
Premium factor 1.75	Y	N	N	N	N	N
Fraud investigation	N	N	Y	Y	Y	N

Age	18–23	18–23	24–59	24–59	60+	60+
Gender	Male	Female	Male	Female	Male	Female
Premium factor	1.75	1.65	1.05	1	1.35	1.25
Fraud investigation	N	N	Y	Y	Y	N