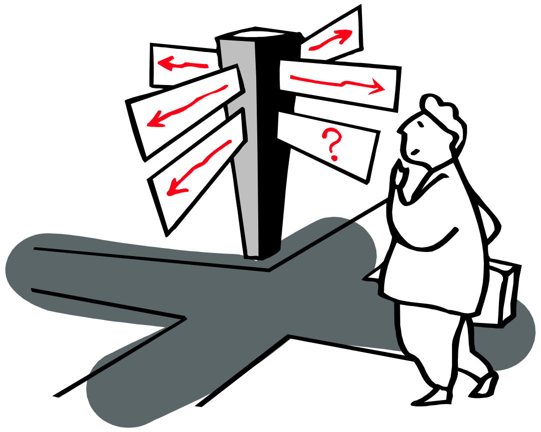
Software Testing

3. Test Design - part 2
Decision Table Testing
State Transition Testing
Scenario-Based Testing (Use Cases)

Universiteit Antwerpen

3. Test Design - Part 2



(Loosely based on "Chapter 6 - 7 - 8" of Practical Test Design
+ "Chapter 7" of Software Testing)

- Models in Testing
- Decision Tables
  + Decision variables and conditions
    - don't care, can't happen, don't know
- State Machines
  + What? (variants: Mealy & Moore)
  + State Transition Tables (State-to-state, Event-to-state)
- Scenario Based Testing
  + Use Cases
  + User Stories

Why Models? (in Testing)

Models are what distinguishes craft from engineering!

Testing is a search problem

- trillions and trillions of possible input and state combinations
- only a few will reach, trigger, propagate and reveal faults

Solutions?

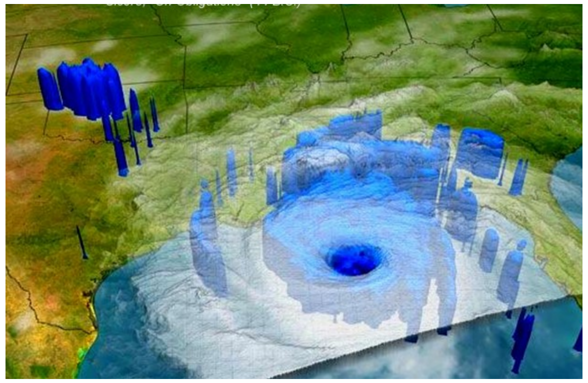
Brute force …
does not work at this scale

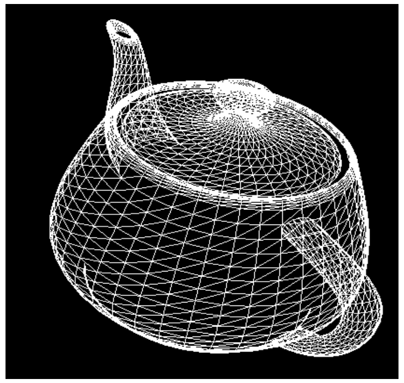
(Smart) Poking around …
inefficient +
false feeling of confidence

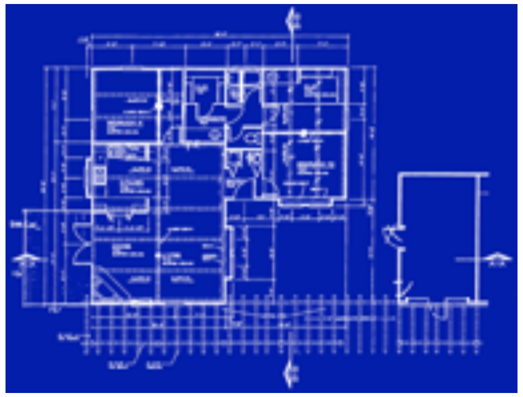
Model based = systematic, focused and automated

- Systematic: we try every target combination (target ≈ model)
- Focused: where are bugs likely? (likely ≈ fault model)
- Automated: more tests; more repeatable test (tools ≈ model)

What is a Model? (in Testing)







ENGINEERING MODELS

- subject
  + e.g. airplanes, buildings, weather, economics
- point of view/theory
  + e.g. gravitation, air-pressure, law of demand and offer
- representation
  + wire frame image, blueprint
  + mathematical equations
- technique
  + skill and expertise of modeller matters

TEST MODELS

- subject
  + e.g. implementation, component, … under test
- point of view/theory
  + required behaviour, where are bugs likely?
- representation
  + graphs (nodes & edges),
  + checklists
- technique
  + experience with requirements, design, …

coverage

Cartoons vs. Models

CARTOONS

- sketching, refining, documentation
  + initial analysis, design, …

potentially

- incomplete
- ambiguous
- inconsistent
- not integrated

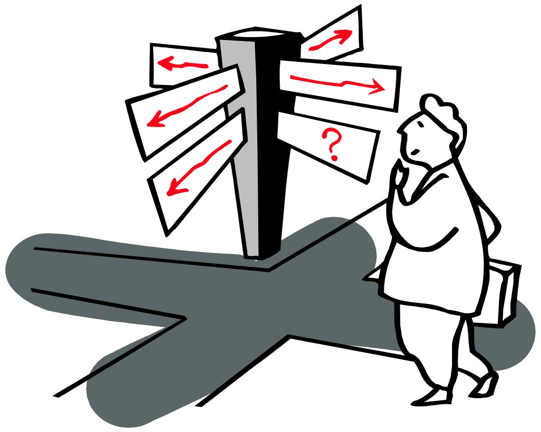
(TEST) MODELS

- Specify precisely
- what is allowed …
  what is NOT allowed

Different models representing various perspectives

- necessarily
  + complete
  + unambiguous
  + consistent
  + coherent

3. Test Design - Part 2



(Loosely based on “Chapter 6 - 7 - 8” of Practical Test Design
+ “Chapter 7” of Software Testing)

- Models in Testing
- Decision Tables
  + Decision variables and conditions
    - don’t care, can’t happen, don’t know
- State Machines
  + What? (variants: Mealy & Moore)
  + State Transition Tables (State-to-state, Event-to-state)
- Scenario Based Testing
  + Use Cases
  + User Stories

Decision Table: example

- Annual renewal of a hypothetical car insurance policy

	Condition Section		Action Section		
Variant	# Claims	# Insured Age	Premium Increase	Send Warning	Cancel
1	0	<= 25	50	No	No
2		>= 26	25	No	No
3	1	<= 25	100	Yes	No
4		>= 26	50	No	No
5	2 to 4	<= 25	400	Yes	No
6		>= 26	200	Yes	No
7	5 or more	Any	0	No	Yes

age is a “don’t care condition” when # Claims >= 5

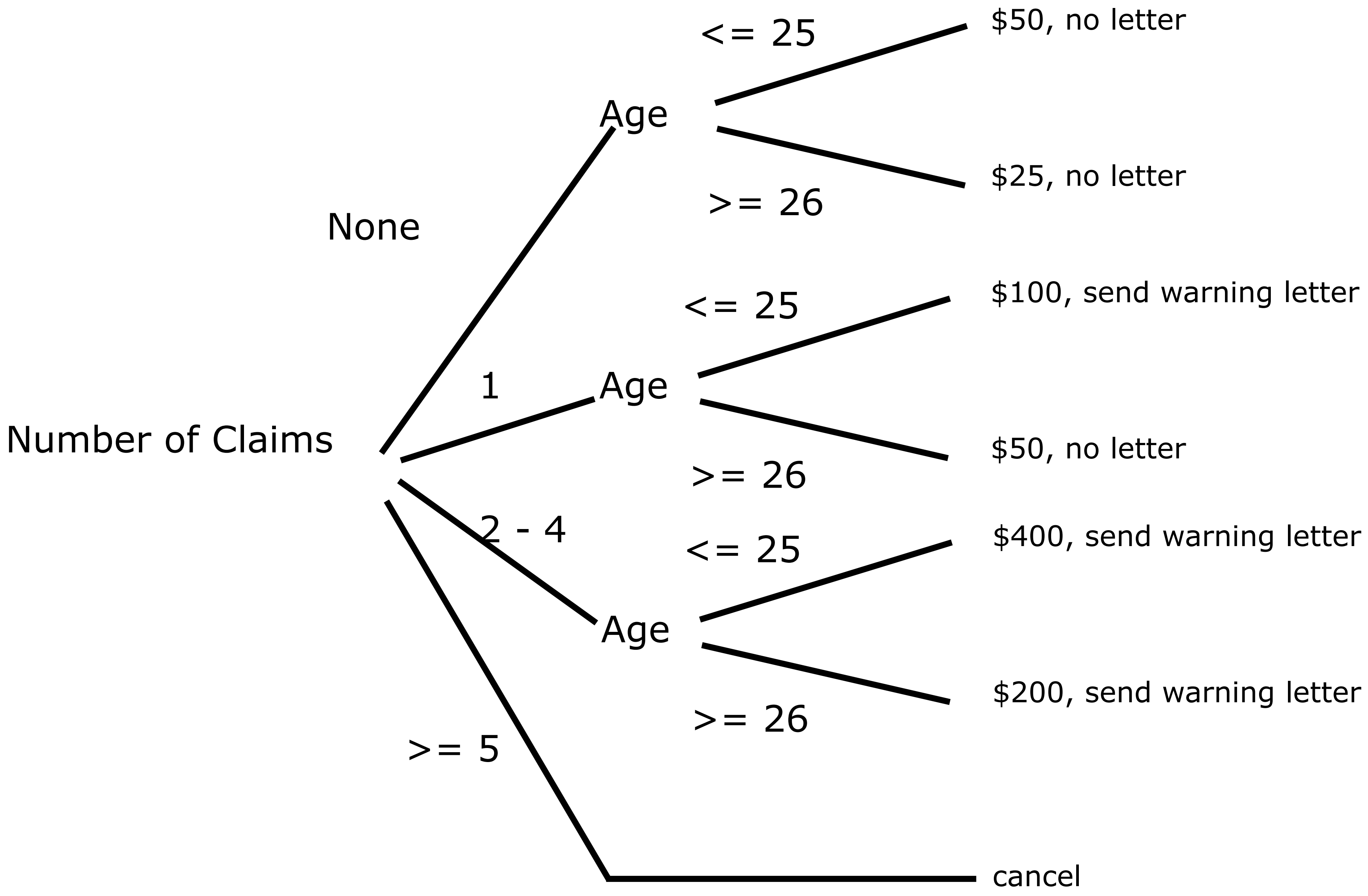
Implicit Assumptions

- Logical operator to interpret a row? (AND)
- Can a person of 15 really be insured? And what about 99?

Decision Table: column wise view

		Variant						
		1	2	3	4	5	6	7
Condition Section	Number of Claims	0	0	1	1	2-4	2-4	>= 5
	Insured Age	<= 25	>= 26	<= 25	>= 26	<= 25	>= 26	Any
Action Section	Premium increase	50	25	100	50	400	200	0
	Send warning	No	No	Yes	No	Yes	Yes	No
	Cancel	No	No	No	No	No	No	Yes

Decision Table: decision tree view



Number of Claims

None

Age

<= 25

$50, no letter

>= 26

$25, no letter

1

Age

<= 25

$100, send warning letter

>= 26

$50, no letter

2 - 4

Age

<= 25

$400, send warning letter

>= 26

$200, send warning letter

>= 5

cancel

- easier to understand (and program)
- possibility to overlook a case

Decision Tables: When

- One of several distinct responses is to be selected
- … according to distinct cases of input variables (finite!)
  + input cases are mutually exclusive
- response is independent of
  + order of input variables
  + prior input or output

distinction with state models

- typical for business rules, simple protocols
- prefer small tables
  + (combinational explosion)
    - n conditions ⇒ 2^n variants

Identify decision variable and conditions

decision table with n conditions ⇒ 2^n variants

usually fewer variants, only explicit variants are listed
Implicit variants?

- **don’t care:** condition true or false, doesn’t change the action
  + e.g.: when # claims >= 5, then age is “don’t care” condition
- **can’t happen:** condition which is assumed never to become true
  + e.g.: insured age < 16 or 18 (age when driving is allowed)
  + can sometimes occur when context changes (Ariane crash)
  + must be tested explicitly (sometimes is a “surprise”)
- **don’t know:** is an incomplete model
  + e.g. what happens with an age > 300?

can’t happen + don’t know (+ don’t care): typical source for bugs

- specify resulting action anyway (defaults)
- test for the result

Testcases for Decision Tables

- Boundary Value Analysis on the conditions
  + explicit tests for don’t care / can’t happen
- Oracle = Actions Section
  + Actions need to be verifiable by test framework

Condition Section	Number of Claims	0	0	1	1	2-4	2-4	>= 5
		select in, on, off, out points here: 0, 1, 2, 3, 4, 5, 7 can’t happen: 1000 claims						
	Insured Age	<= 25	>= 26	<= 25	>= 26	<= 25	>= 26	don’t care
		select in, on, off, out points here: 20, 25, 26, 30 can’t happen: 13 (extremely young) + 17, 18 (borderline driver licence) + 105 (extremely old)						
Action Section	Premium increase	50	25	100	50	400	200	0
	Send warning	No	No	Yes	No	Yes	Yes	No
	Cancel	No	No	No	No	No	No	Yes

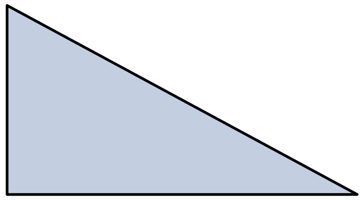
Triangle Problem (revisited)

A valid triangle must meet two conditions

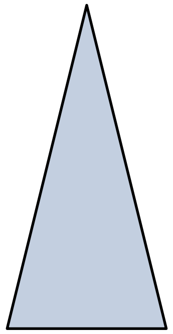
- No sides may have a length of zero
- each side must be shorter than the sum of all sides divided by 2

A triangle is

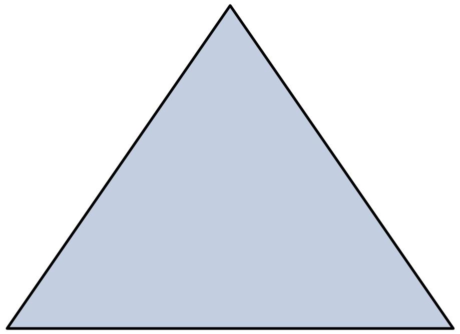
- scalene: no sides are equal in length



- isosceles: there exist two sides which are equal in length



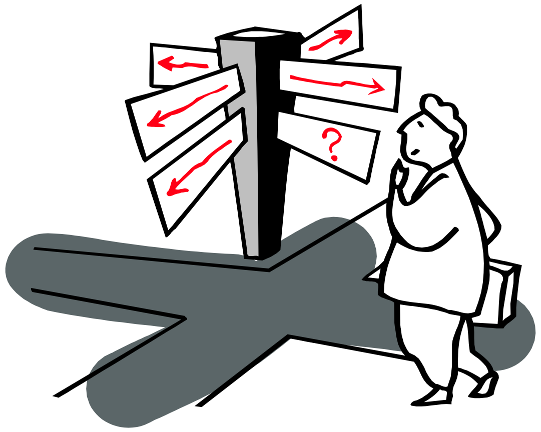
- equilateral: all sides are equal in length



Break Out Groups

Create a decision table for 3 strictly positive integers
(ignore zero, negative values, alphanumeric, missing values, …)

3. Test Design - Part 2



(Loosely based on “Chapter 6 - 7 - 8” of Practical Test Design
+ “Chapter 7” of Software Testing)

- Models in Testing
- Decision Tables
  + Decision variables and conditions
    - don’t care, can’t happen, don’t know
- State Machines
  + What? (variants: Mealy & Moore)
  + State Transition Tables (State-to-state, Event-to-state)
- Scenario Based Testing
  + Use Cases
  + User Stories

What is a state machine?

= system whose output is determined by both current and past input

(with Decision tables, only current input influences result)

- identical inputs are not always accepted
- identical inputs may produce different outputs

State machine consists of 4 building blocks

- **State:** an abstraction that summarizes the past inputs
- **Transition:** an allowable two-state sequence
  + "accepting" and "resulting" state
  + Caused by an event; may result in an action
    - **Guarded transition:** event + guard predicate
- **Event:** an input (or an interval of time)
- **Action:** the result or output that follows an event

Special states: initial state, current state, final state

Variants: Mealy and Moore

MEALY State Machine

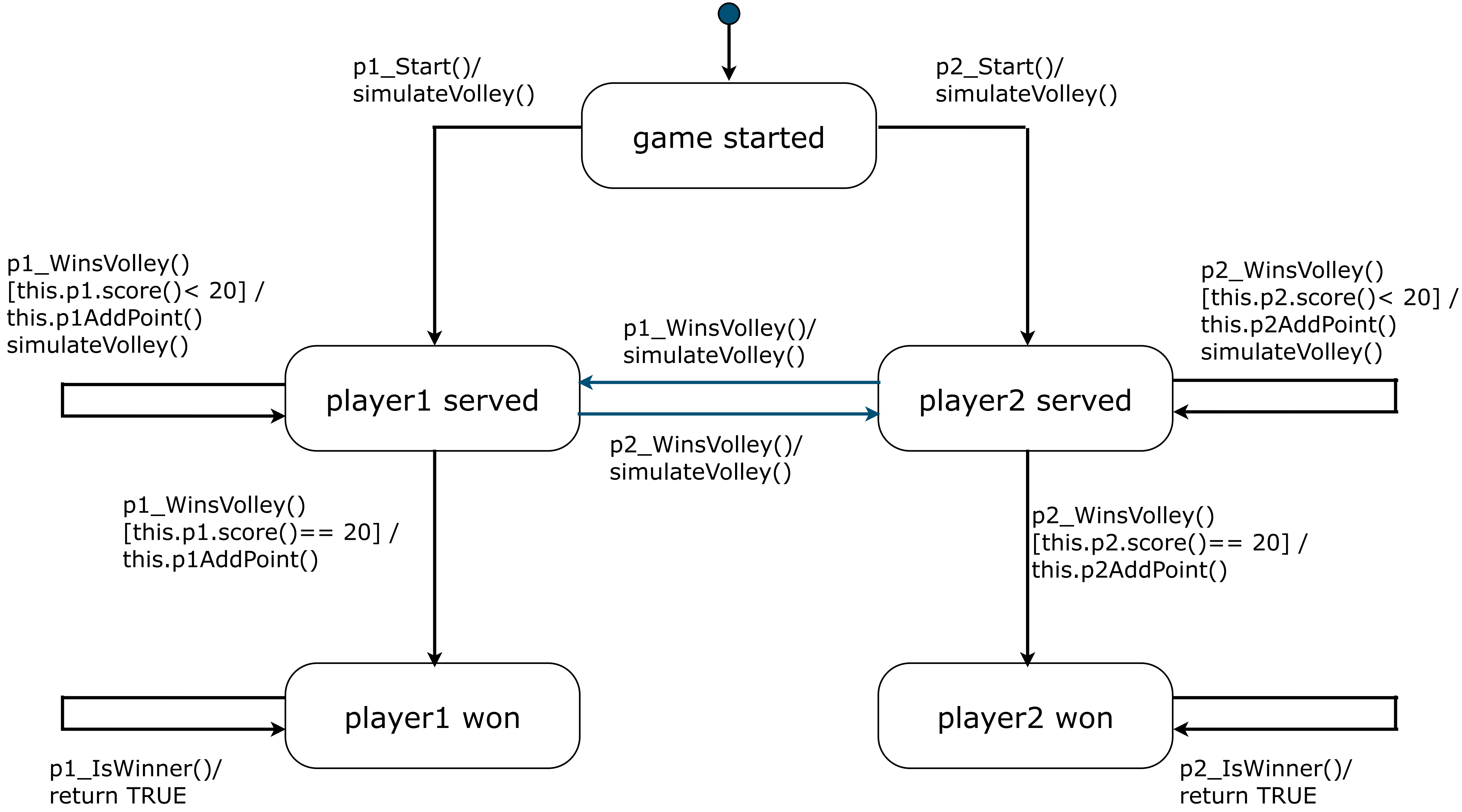
- transitions are *active*
  + may have output action
  + any output action may be used in more than one transition
- states are *passive*

MOORE State Machine

- transitions are *passive*
- states are *active*
  + may have output action
  + every output action has at least one state

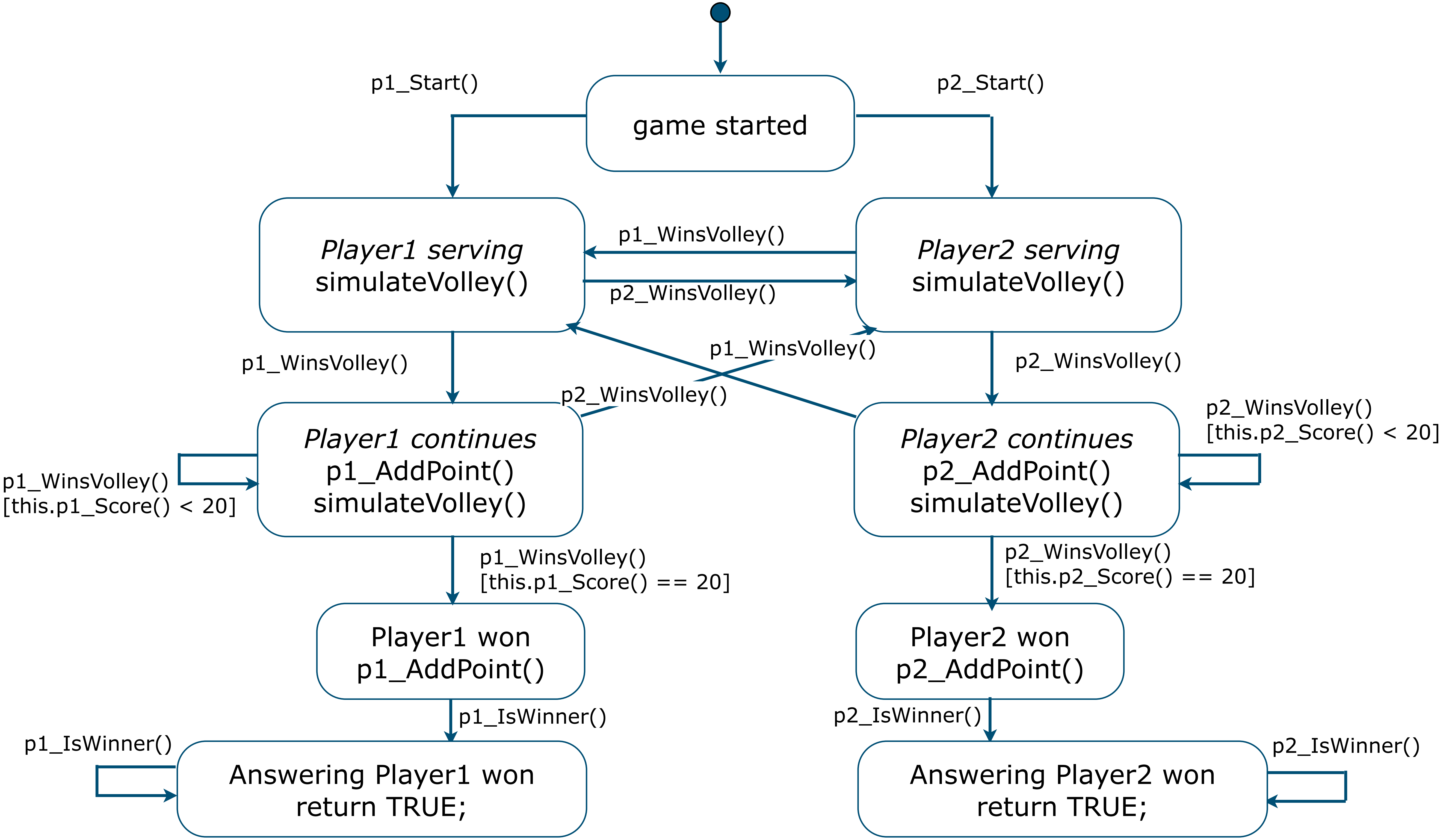
- Mathematically equivalent
- MEALY is preferred for engineering design
- UML allows both!
  - Hybrids are bad for test design!

Two-player Squash: Mealy State Machine



game started
p1_Start()/ simulateVolley()
p2_Start()/ simulateVolley()
player1 served
player2 served
p1_WinsVolley() [this.p1.score()< 20] / this.p1AddPoint() simulateVolley()
p2_WinsVolley() [this.p2.score()< 20] / this.p2AddPoint() simulateVolley()
p1_WinsVolley()/ simulateVolley()
p2_WinsVolley()/ simulateVolley()
p1_WinsVolley() [this.p1.score()== 20] / this.p1AddPoint()
p2_WinsVolley() [this.p2.score()== 20] / this.p2AddPoint()
player1 won
player2 won
p1_IsWinner()/ return TRUE
p2_IsWinner()/ return TRUE

Two-player Squash: Moore State Machine



State Transition Tables

- **state-to-state** format
  + row = accepting state; column = resulting state
  + cell = transition = event (with guard) + action

- **event-to-state** format
  + row = event (with guard); column = accepting state
  + cell = transition = action + resultant state

- **expanded state-to-state** format
  + 2 state-to-state tables: 1 for events and 1 for actions
  + separates the transition function and output function

Example: State-to-state

Current State	Resultant State/Event/Action				
	Game Started	**Player 1 Served**	**Player 2 Served**	**Player 1 Won**	**Player 2 Won**
Game Started		p1_start()	p2_start()		
		simulateVolley()	*simulateVolley()*		
Player 1 served		p1_winsVolley() [p1 score < 20]	p2_winsVolley()	p1_winsVolley() [p1 score == 20]	
		this.p1AddPoint(), simulateVolley()	*simulateVolley()*		
Player 2 served		p1_winsVolley()	p2_winsVolley() [p2 score < 20]		p2_winsVolley() [p2 score == 20]
		simulateVolley()	*this.p2AddPoint(), simulateVolley()*		
Player 1 won				p1_IsWinner()	
				return TRUE	
Player 2 won					p2_IsWinner()
					return TRUE

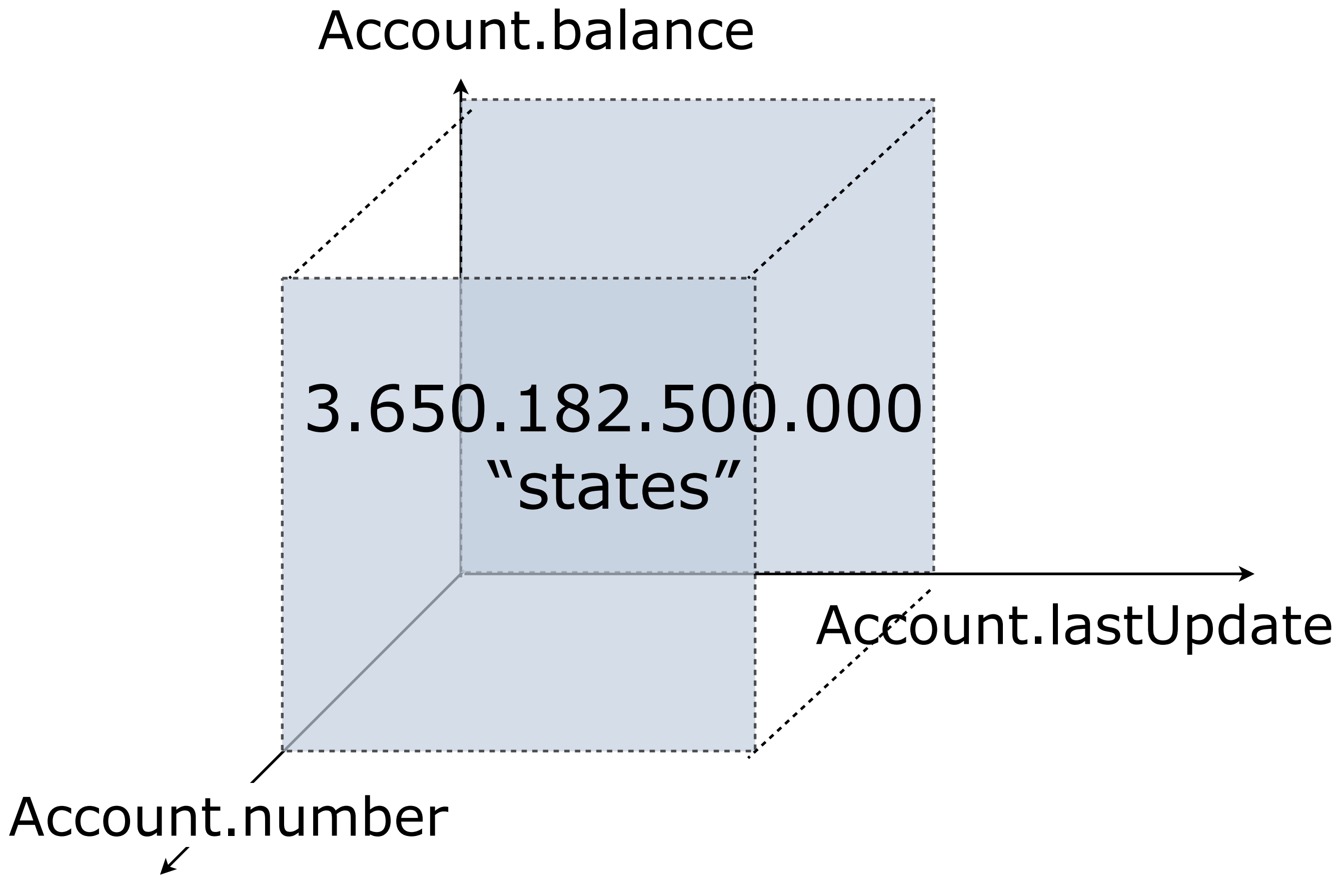
Example: Event-to-state

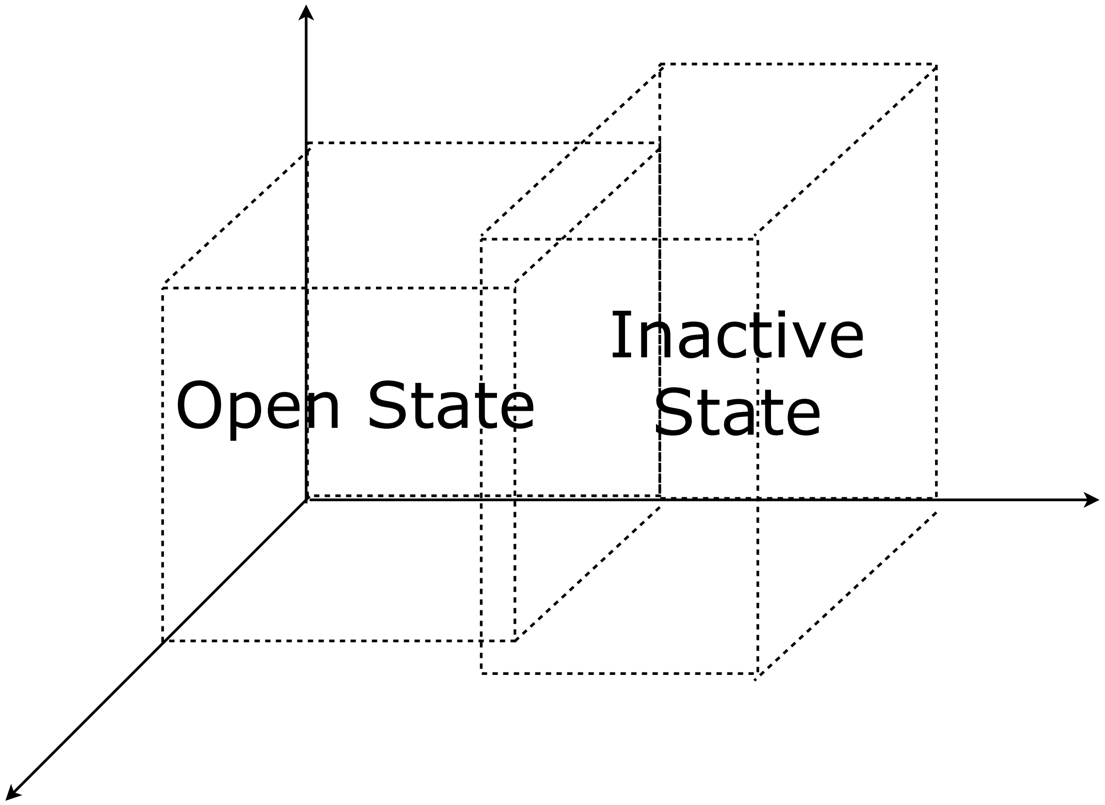
		Current State/Action/Next State				
Event	Guard	**Game Started**	**Player 1 Served**	**Player 2 Served**	**Player 1 Won**	**Player 2 Won**
p1_start()		*simulateVolley()*				
		Player 1 Served				
p2_start()		*simulateVolley()*				
		Player 2 Served				
p2_WinsVolley()	DC (don’t care)		*simulateVolley()*			
			Player 2 Served			
	p2_Score < 20			*this.p2AddPoint() ,simulateVolley()*		
				Player 2 Served		
	p2_Score == 20			*this.p2AddPoint()*		
				Player 2 Served		
p1_WinsVolley()	…		…			
p1_IsWinner()					*return TRUE*	
					Player 1 Won	
…						…

What is State?

- A particular subset of a class combinational value set
  + Represented by a State Invariant (predicate)

```
Class Account {
     private: AccountNumber   number;
      Money   balance;
      Date lastUpdate}
```





Advice / Best Practices

state invariant defines a subset of legal values for a class

- define a predicate for each state invariant
  + allows for explicit verification whether system is in a given state
- state invariant is same or stronger than class invariant
  + method postconditions + state invariants = all states

don’t use hybrids (i.e. no mixing of Mealy and Moore)

Beware

- UML initial state (solid circle)
- UML final state (bulls-eye)
  + Typically allow for silent transitions without any events
  + What about constructor / destructor?
    - Create alpha-state: object after constructor call
    - Create omega-state: object right before destructor call

What is a Transition?

Transition =

- 1 state invariant for accepting state
- 1 state invariant for resulting state
- 1 associated event
  + message sent to the class under test
  + interrupt or similar external control (timer, …)
- [optional guard expression]
  + predicate expression evaluated in the context of class under test
- 0 or more actions
  + message sent to a server object of the class under test
  + response provided by an object of the class under test

Unspecified Event/State Pairs

What should happen when a state machine in a given state receives an event not specified for this state?

- ignore
  - + standard semantics for state machines
  - + unacceptable for testing purposes
- omitted
  - + incomplete specification: extend the state machine
- illegal (or “impossible”)
  - + illegal event = valid event, not acceptable for the current state (e.g. “pop()” on an empty stack)
  - + if accepted results in an illegal transition
    - Sneak path

- beware: guarded transitions, what if the guard is FALSE?
  - + transition to an “Illegal event exception” state

Response Matrix

Response Matrix = Modified event-to-state table

- ROWS: list events + guards
  + unguarded event: 1 row
  + guarded event: 1 row for each unique event/guard combination
    - one row for each truth combination of subexpressions
      + additional column for all sub expressions
  + if event is sometimes guarded, then include a “don’t care” row
- COLUMNS: list Accepting states
- CELLS: list responses
  + error codes for possible responses

0. Accept (perform specified transition)
1. Queue (queue event for subsequent evaluation and ignore)
2. Ignore
3. Flag (return a non zero error code)
4. Reject (raise an exception)
5. Mute (disable the source of events and ignore)

State Based Testing: Fault Model

s states, e events, a actions gives

$\boxed{(s \times a)^{es}}$ possible implementations

- 5 states, 2 events and 2 actions: 10 billion possibilities
- only one correct implementation

Possible faults

- Specification
  + missing or incorrect transition
  + missing or incorrect event
  + missing or incorrect action
- Implementation
  + an extra, missing or corrupt state
  + a sneak path (a message is accepted when it shouldn't be)
  + an illegal message failure (unexpected message causes a failure)
  + trap door (implementation accepts undefined messages)

No empirical models about distribution of faults!

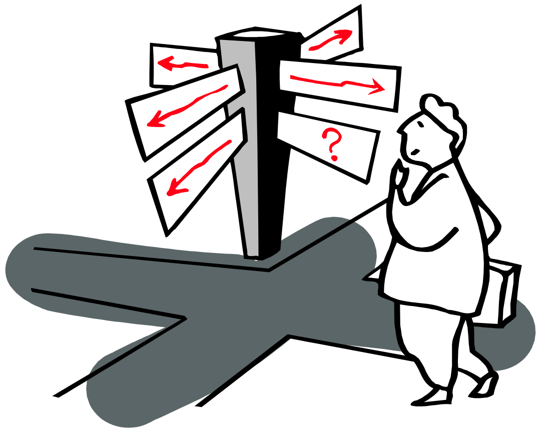
State Machine Coverage

- All States
  - + each state is visited at least once
- All Events
  - + each event is triggered at least once
- All Transitions
  - + every specified transition is traversed at least once
- All n-Transition sequences
  - + every specified transition sequence of n events
  - + special case: all-transition-**pairs** criterion
    - every pair of adjacent transitions are traversed at least once

- Paths
  - + All loop-free Paths
  - + All-one-loop Paths criterion
    - loops are traversed exactly once
  - + All round-trip Paths
    - every sequence of specified transitions beginning and ending in the same state is traversed at least once
    - subtle variation: when one path has multiple possibilities for loops

Based on years of experience with state-based testing of hardware and telecommunications infrastructure.

3. Test Design - Part 2



(Loosely based on “Chapter 6 - 7 - 8” of Practical Test Design
+ “Chapter 7” of Software Testing)

- Models in Testing
- Decision Tables
  + Decision variables and conditions
    - don’t care, can’t happen, don’t know
- State Machines
  + What? (variants: Mealy & Moore)
  + State Transition Tables (State-to-state, Event-to-state)
- Scenario Based Testing
  + Use Cases
  + User Stories

What are Use Cases?

- **Use Case**
  + A use case describes outwardly visible requirements of the system
  + A use-case is a generic description of an entire transaction executed to achieve a *goal* (= the use case goal) and involving several *actors*.

- **Actors**
  + Actors have responsibilities
  + To carry out responsibilities, an actor sets goals
  + Primary actor (= stakeholder) has unsatisfied goal and needs system assistance
  + Secondary actor provides assistance to satisfy the goal

- **Scenario**
  + Scenario = an instance of a use-case, showing a typical example of its execution
    - Use case = Primary “success” & secondary “alternative” scenarios
    - Scenario shows how objects interact to achieve the use case goal

Primary & Secondary Scenarios

- Scenario is one way to realize the use case
  From the actors point of view!
- = List of steps to accomplish the use case goal

- **Primary "success" scenario**
  + = Happy day scenario
  + Scenario assuming everything goes right
  (i.e., all input is correct, no exceptional conditions, …)

- **Secondary "alternative" scenarios**
  + Scenario detailing what happens during special cases
  (i.e., error conditions, alternate paths, …)

Example: Place Order Scenario (1/2)

USE CASE 5	Place Order
Goal in Context	Customer issues request by phone to National Widgets; expects goods shipped and to be billed.
Scope & Level	Company, Summary
Preconditions	National Widgets has catalogue of goods
Success End Condition	Customer has goods, we have money for the goods.
Failed End Condition	We have not sent the goods, Customer has not spent the money.
Primary Actors	Customer, Customer Rep, Shipping Company
Secondary Actors	Accounting System, Shipping Company
Trigger	Purchase request comes in.

DESCRIPTION	
Step	Action
1.	Customer calls in with a purchase request.
2.	Customer Rep captures customer info.
3.	WHILE Customer wants to order goods.
3.1.	Customer Rep gives Customer info on goods, prices, etc.

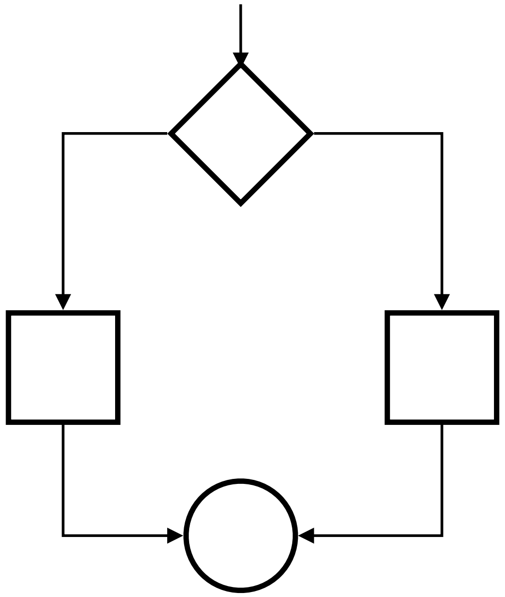
Example: Place Order Scenario (2/2)

3.2.	Customer selects good to add to order list.
4.	Customer approves order list.
5.	Customer supplies payment details.
6.	Customer Rep creates order.
7.	Customer Rep requests Accounting System to Charge Account.
8.	Customer Rep requests Shipping Company to Deliver Product.
9.	Customer pays goods.
Branch	**SUBVARIATIONS**
1.	Customer may use: (a) phone in, (b) fax in, (c) use web order form.
4.	Customer may pay via: (a) credit card; (b) cheque; (c) cash.
Branch	**ALTERNATIVE PATHS**
any	Customer may cancel transaction.
Branch	**EXTENSIONS**
After 3.2	Out of selected good: 3.2.a. Renegotiate Order (Use case 44).
Before 9	Customer returns goods: 9a. Handle returned goods (Use case 45).

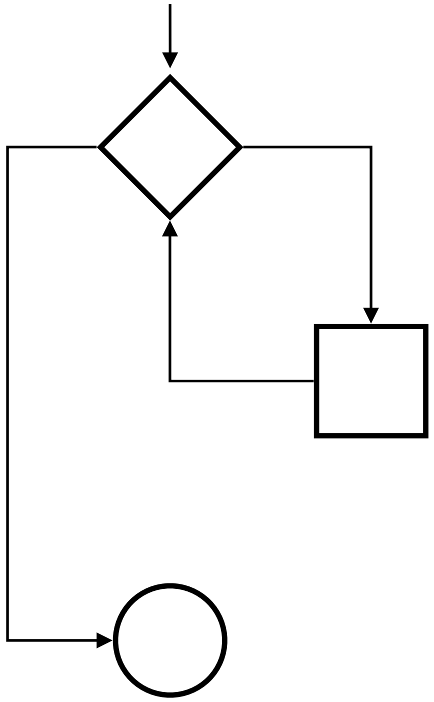
Test Use Case Scenario

Consider the “steps” in the use case as a control flow graph

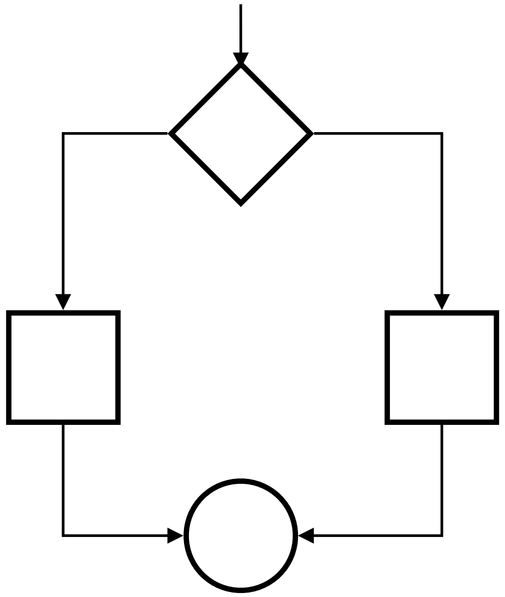
if-then-else



while

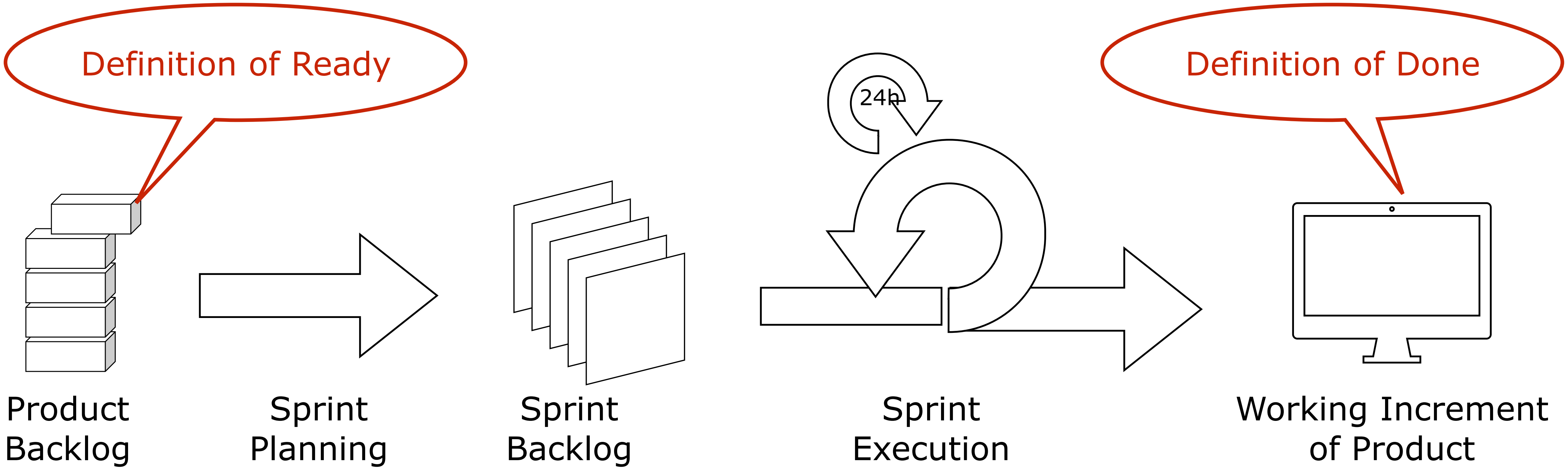


branch
(essentially an if-then-else or a go-to)

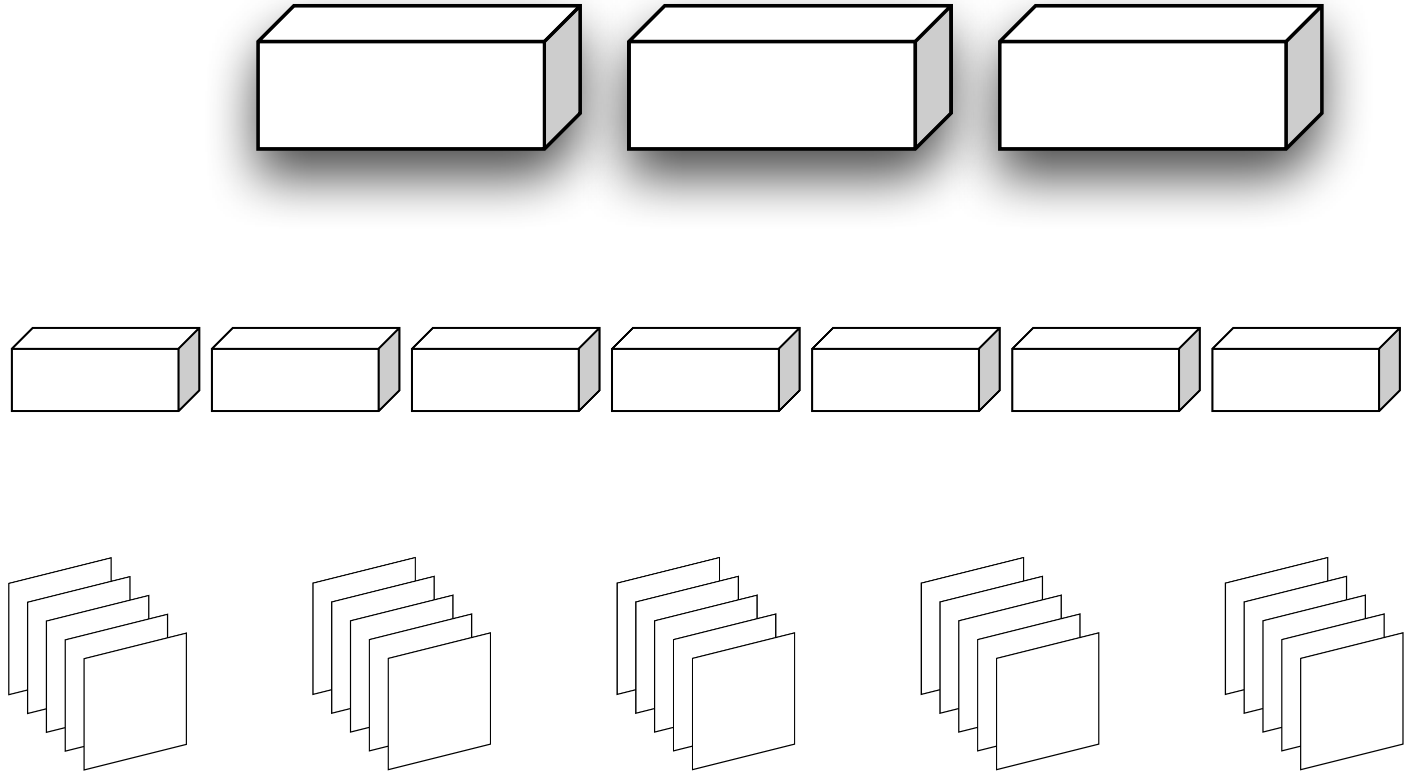


- Condition Coverage
  + Possible outcomes for each condition (“true” or “false”) at least once
- Branch coverage (a.k.a. decision coverage)
  + Every arrow leaving a decision is executed at least once
- Path coverage
  + Cover entry-exit paths

User Stories in scrum



Product Backlog — Level of Detail



Epic	Months	Bigger than a release
Features	Weeks	Bigger than a sprint
Sprintable Stories	Days	Sprint Ready

INVEST Criteria

I	Stories should be *independent* of another and should not have dependencies on other stories
N	*Negotiable*: Too much detail on story limits conversation with the customer
V	Each story has to be of *value* to the customer
E	Stories should be small enough to *estimate*
S	Stories should be *small* enough to be completed in one iteration
T	*Testable*: Acceptance criteria should be available

User Stories: Gherkin Format

Template

As a <user role>
I want to <goal>
so that <benefit>.

- …
- …
- …

Conditions of Satisfaction

Example

As a *clerk*
I want to *calculate stampage*
so that *goods get shipped fast*.

- Verify with nearby address
- Verify with overseas address
- Verify with parcels <= 1kg
- Verify with fragile parcel

Gherkin Syntax

```
Scenario: Eric wants to withdraw money from his bank account
at an ATM
    Given Eric has a valid Credit or Debit card
    And his account balance is $100
    When he inserts his card
    And withdraws $45
    Then the ATM should return $45
    And his account balance is $55
```

```
Scenario Outline: A user withdraws money from an ATM
    Given <Name> has a valid Credit or Debit card
    And their account balance is <OriginalBalance>
    When they insert their card
    And withdraw <WithdrawalAmount>
    Then the ATM should return <WithdrawalAmount>
    And their account balance is <NewBalance>

    Examples:
      | Name   | OriginalBalance | WithdrawalAmount | NewBalance |
      | Eric   | 100             | 45               | 55         |
      | Gaurav | 100             | 40               | 60         |
      | Ed     | 1000            | 200              | 800        |
```

Testing a 3-Tiered Architecture

```
Scenario: Eric wants to withdraw money
          from his bank account at an ATM
    Given Eric has a valid Credit or Debit card
    And his account balance is $100
    When he inserts his card
    And withdraws $45
    Then the ATM should return $45
    And his account balance is $55
```

Application Layer

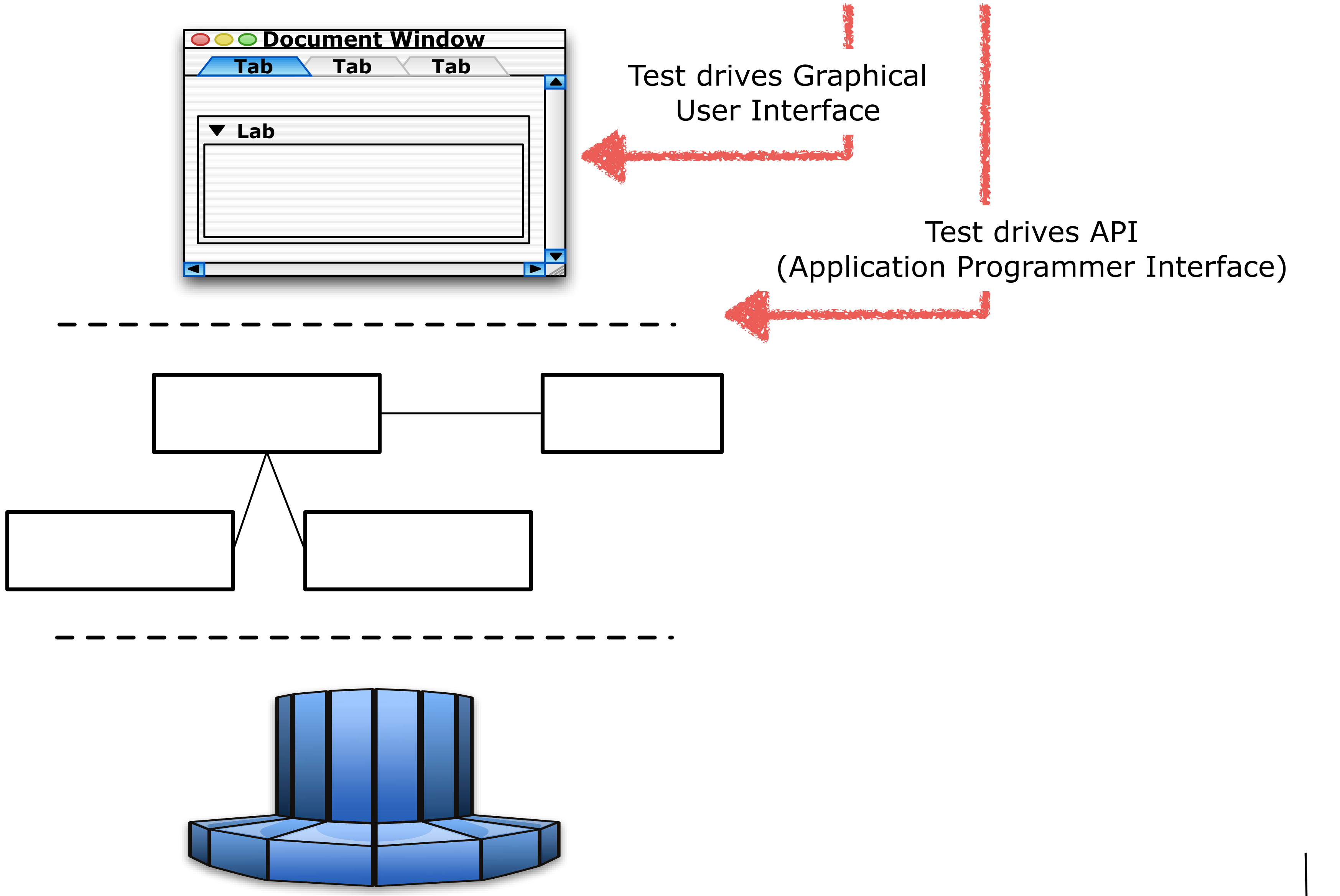
- Models the UI and application logic

Domain Layer

- Models the problem domain (usually a set of classes)

Database Layer

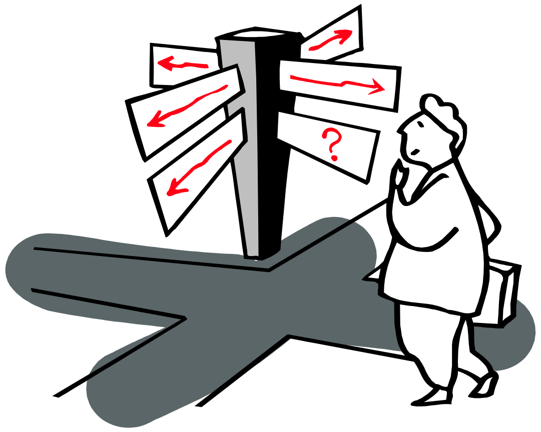
- Provides data according to a certain database paradigm (usually relational database)



Test drives Graphical User Interface

Test drives API (Application Programmer Interface)

3. Test Design - Part 2



(Loosely based on “Chapter 6 - 7 - 8” of Practical Test Design + “Chapter 7” of Software Testing)

- Models in Testing
- Decision Tables
  - + Decision variables and conditions
    - don’t care, can’t happen, don’t know
- State Machines
  - + What? (variants: Mealy & Moore)
  - + State Transition Tables (State-to-state, Event-to-state)
- Scenario Based Testing
  - + Use Cases
  - + User Stories