

Computational Systems

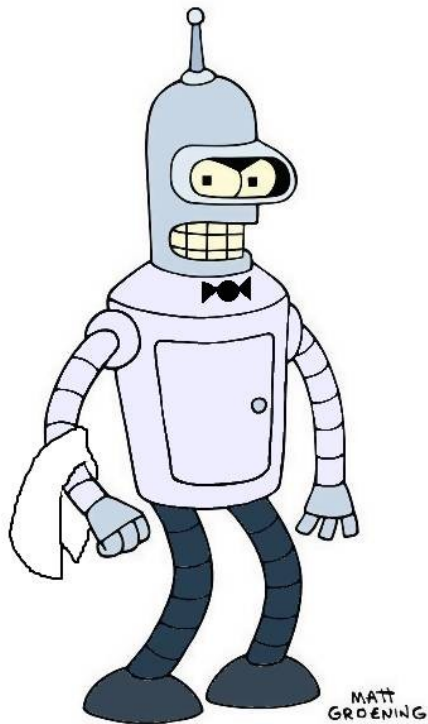
COMP1209

Testing

Yvonne Howard ymh@ecs.soton.ac.uk

A Problem

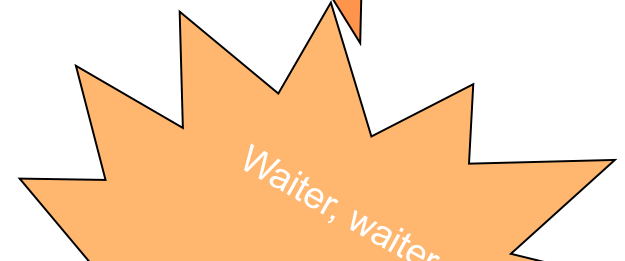
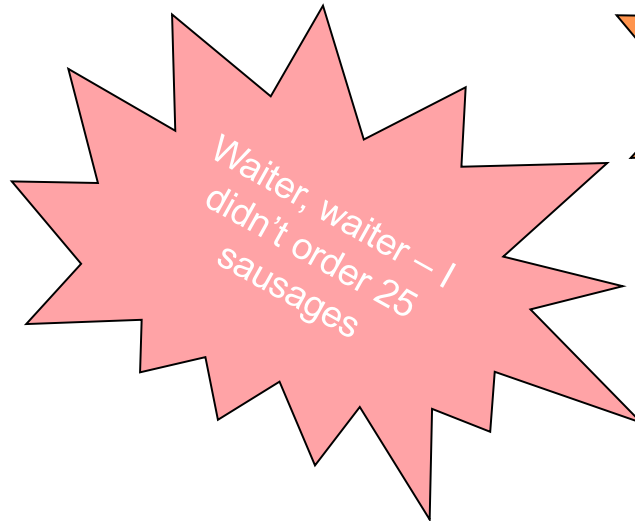
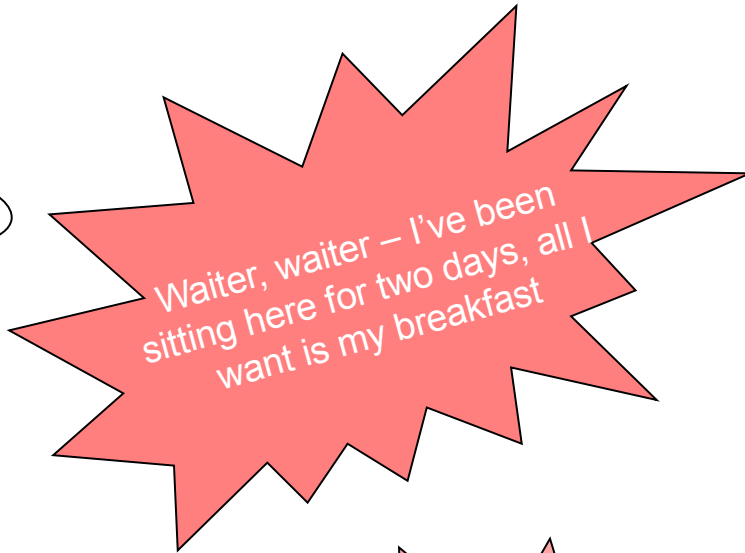
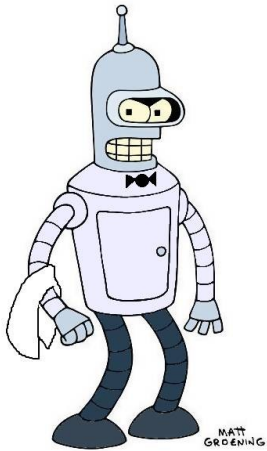
A café wants to build an automated system to provide breakfasts. The robot waiter greets people before taking their order by name.



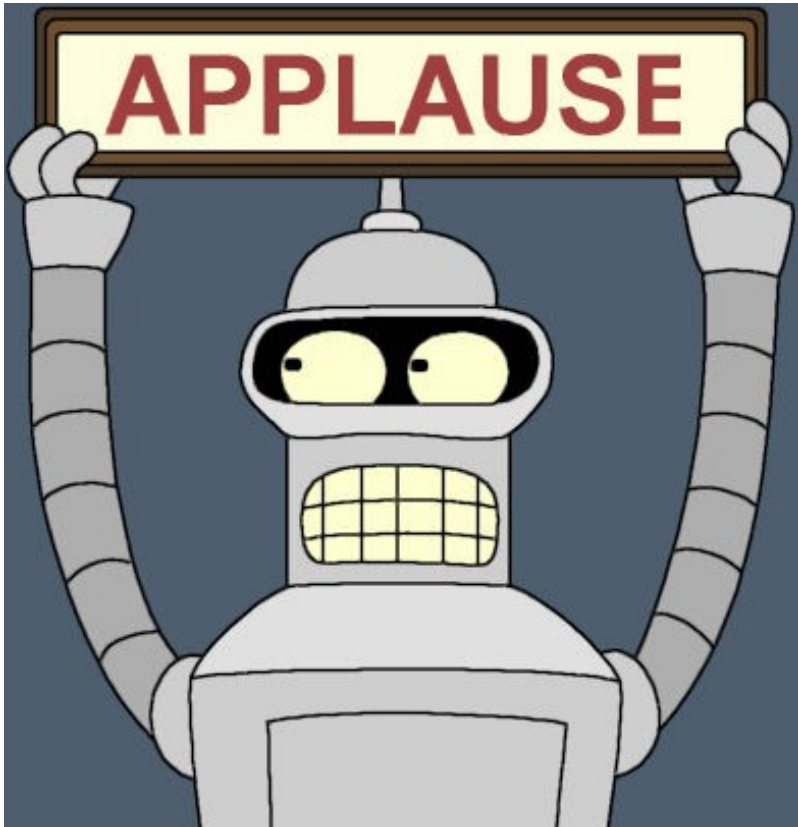
Customers can order different combinations of ingredients for their meal, and also ask for one drink. The system then cooks the breakfast. It must be able to fry sausages, bacon, eggs and mushrooms; toast bread, waffles and muffins; and pour their orange juice or coffee.

The waiter then serves the breakfast.

The all new RoboCafe



So what went wrong and how can we have happiness café?



Software Engineering: Big Picture



This is rarely a straightforward progression – in reality there are lots of iterations and points of feedback

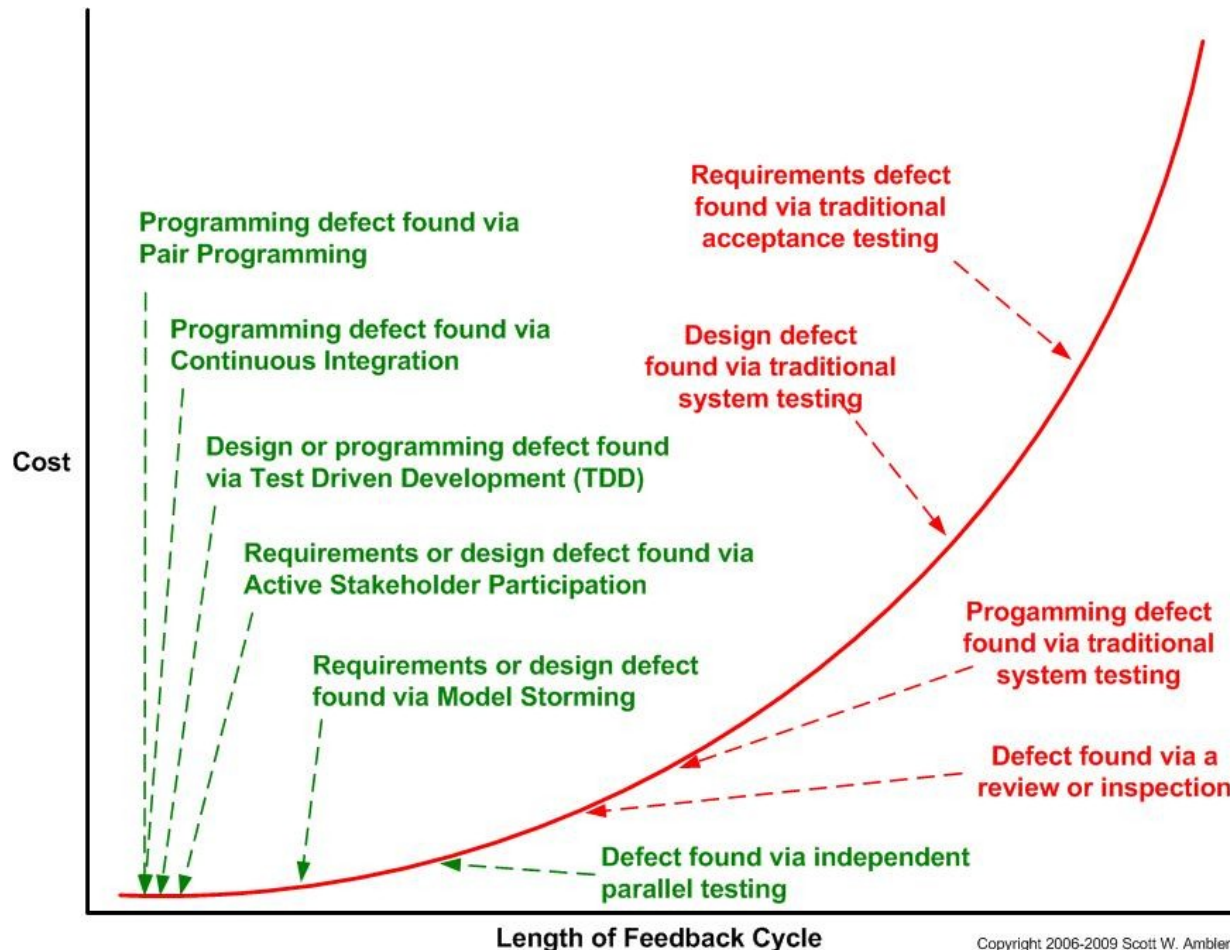
We can do some software engineering

- Testing
 - *Used for most software systems*
- Or
 - Formal verification
 - Build a model of the software
 - Use a prover to prove that the model is correct
 - *Only a very small proportion of systems are verified this way, mostly safety critical systems*
- *We're going to talk about testing*

Cost of Change Curve (Barry Boehm 1981)

- The cost to address a defect rises exponentially the longer it takes you to find it.
- if you inject a defect into your system and find it a few minutes later and fix it, the cost is almost negligible.
- if you find it three months later the cost of fixing rises exponentially
 - to fix the original problem
 - to fix any work based on the defect


The Cost of Change Curve: Traditional versus Agile



- Agile techniques have feedback cycles on the order of minutes or days,
- Traditional techniques have long feedback cycles (end of development)
- Traditional strategies can be effective at finding defects but the cost of fixing them can be much higher

What are we testing for?

Which of these might be valid?

1. Find bugs?
2. Try to break the software?
3. Reduce risk?
4. Check that performance is OK?
5. Prove the software is defect free? 

“Program testing can be used to show the presence of defects, but never their absence”

Edgar Dijkstra

The goal of testing

To increase to an acceptable level the user's confidence that the system under test will behave correctly under all circumstances of interest

- We have to define
 - Correct behaviour
 - Level of confidence
 - Domain of concern

USER FOCUS

Correct behaviour

- Bender takes Dave's order for:
 - 2 sausages, 1 fried egg, 2 slices of toast and a cup of coffee
- Bender serves Dave his breakfast of:
 - 2 sausages 1 fried egg, 2 slices of toast and a cup of coffee
- Other issues (known as Non Functional requirements NFRs, 'ilities) which describe Quality

Correct Behaviour

- Definition of ‘correct behaviour’ required
- Provided by a “Baseline” or “Blueprint”
- Depends on level of testing

Whole system – Requirements specification

Increment – user stories

One module – Program specification

... and so on

- Compare results of the test with what was supposed to

Level of Confidence

- Usually specified as ‘residual defect discovery rate’
 - Number of defects found in a given test or series of tests, or
- Number of defects found in a given time
 - *“Less than 10 non-critical defects discovered in last 7 days”*
- An alternative – Reliability specification

“Mean time between failures shall be not less than 7000 hours”

Tests are supposed to find errors
A good test is focused to find errors
A successful test finds new classes of errors

- So how do we do good, successful, systematic testing?
 - Choose the right test method
 - Make test cases
 - A set of input values and expected outcomes for a software feature
 - Make a test plan
 - All the test cases necessary to test the software thoroughly
 - Follow good testing practice
 - Plan testing early (agile testers, write tests before code)
 - User focus, not designer or programmer
 - Use independent testing staff, not the people who designed or built the software (except agile teams where testers and developers are interchangeable)

When do we test?

- During implementation
 - Unit tests - For each module or submodule
 - Integration tests - When you put modules together
 -
 - Regression tests - When you change or add a component, to make sure that everything still works together
- when you think you are ready to deliver
 - Alpha – in house

Two main types of testing

- White box
- Black Box

Structural (White Box) Testing

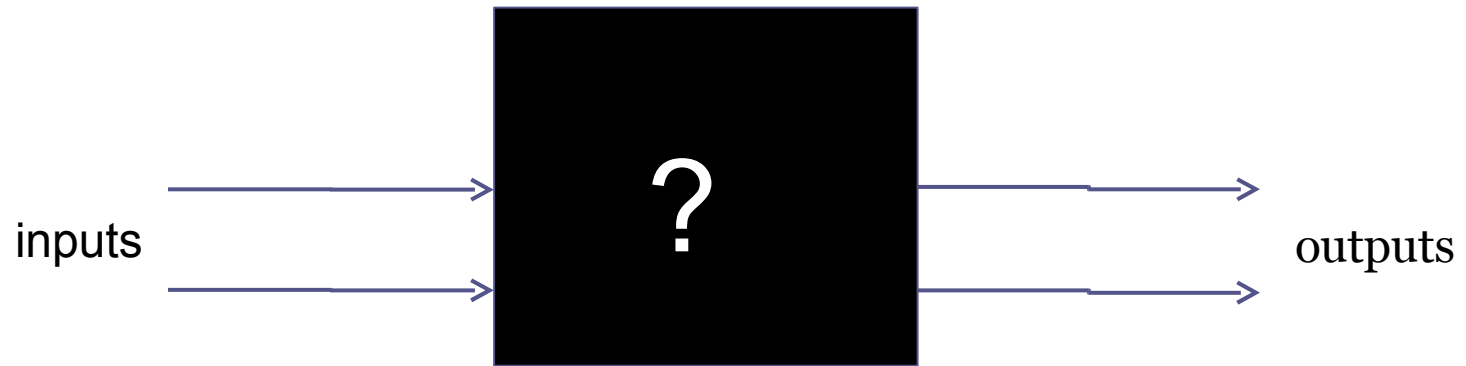
- Uses knowledge of the program structure and algorithms
- Construct test cases that will follow every path through the system
 - Every statement in a method is executed at least once
 - Every branch has been exercised for true/false conditions
- This is very expensive and is usually done only for a small, critical part of the system

Black box testing



Tests without reference to internal processing

- Considers the correct behaviour of inputs and expected outputs
- Uses techniques to reduce the amount of testing needed to satisfy correctness
 - Equivalence partitioning
 - Boundary value analysis



We are going to concentrate on black box testing

Strengths and weaknesses of Black Box Testing

- **Strengths**

- More effective for larger units of code than glass box testing
- Tester needs no knowledge of implementation
- Tester and developer can be independent
- Test's are from the user's point of view
- Exposes ambiguities in the spec (but it's a bit late in the game for that)
- Test cases can be designed early, from the specification

- **Weaknesses**

- Only a small number of possible inputs can reasonably be tested
- Good test cases need clear specifications
- Some program paths will be untested
- Cannot target code directly

What is meant by a ‘test case’?

- “A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement” (IEEE standard)
- “Input – processing – output”
- Example (informal):

Insert a card into an ATM (cash machine). The ATM reads the card and asks for the PIN

- Inputs may include data and/or controls
 - E.g. Numbers entered into a program
 - E.g. Bank card inserted into ATM

Test Cases Problem

- A user ID consists of two characters < 1 alpha char> < 1 digit>
 - The < 1 alpha char> consists of the characters A..Z or a..z
 - < 1 digit> consists of the characters 0..9
- What test cases do you think that you might need to ensure that software processes a user ID correctly?

Test case	System action	Test data input	Pass criteria	Pass/fail
1	When the system asks for a userID	R6	Accept user ID and continue	
2	When the system asks for a userID	xyz	Reject userID with error message 'only 2 chars'	
3	When the system asks for a userID	6R	Reject userID with error	
4	When the system asks for a userID	R%	Reject userID	
5...	

For a 2 char userID! – How can we stop the test case explosion?

Test Scenarios

- Scenarios are sequences of test cases which represent a typical use of the system
 - Example of an ATM Scenario

No.	Input	Initial state	Output	Final state
1	User enters valid card	Idle	ATM reads it successfully, displays "Enter PIN" prompt	Await PIN
2	User enters valid PIN	Await PIN	ATM displays Transaction Selection screen	Get Txn
3	User selects withdrawal transaction for £50	Get Txn	ATM checks balance, debits account, dispenses cash	Take cash
4	User takes card and money	Take cash	ATM displays "Enter your card," returns to Idle state	Idle

- Choice of Scenario has to take account of
 - Valid sequences (must work correctly)
 - Invalid sequences must produce correct error messages (and not

Where do we find our test cases and scenarios?

- Test cases and scenarios need to explore
 - Correct behaviour
 - Error cases
- Baseline (or blueprint) documents define correct behaviour, depending on the level of testing
 - Requirements – System Test
 - High level design – Integration Test
 - Low level design – Unit Test
 - Agile developers write tests before they write code
- Test cases and scenarios are derived from the baseline documentation
 - Large numbers are often required, determined by
 - Logical complexity – Unit test and Integration test

How Many Tests?

- A balance must be struck
 - We want high quality - thorough testing
 - We need to deliver the software – cannot go on testing for ever
- We have to pick specific values to use in test cases
 - Not enough time to test *all* possible values
- *Equivalence Partition Analysis* helps to select a sensible number of test cases to run

Equivalence Partition Analysis

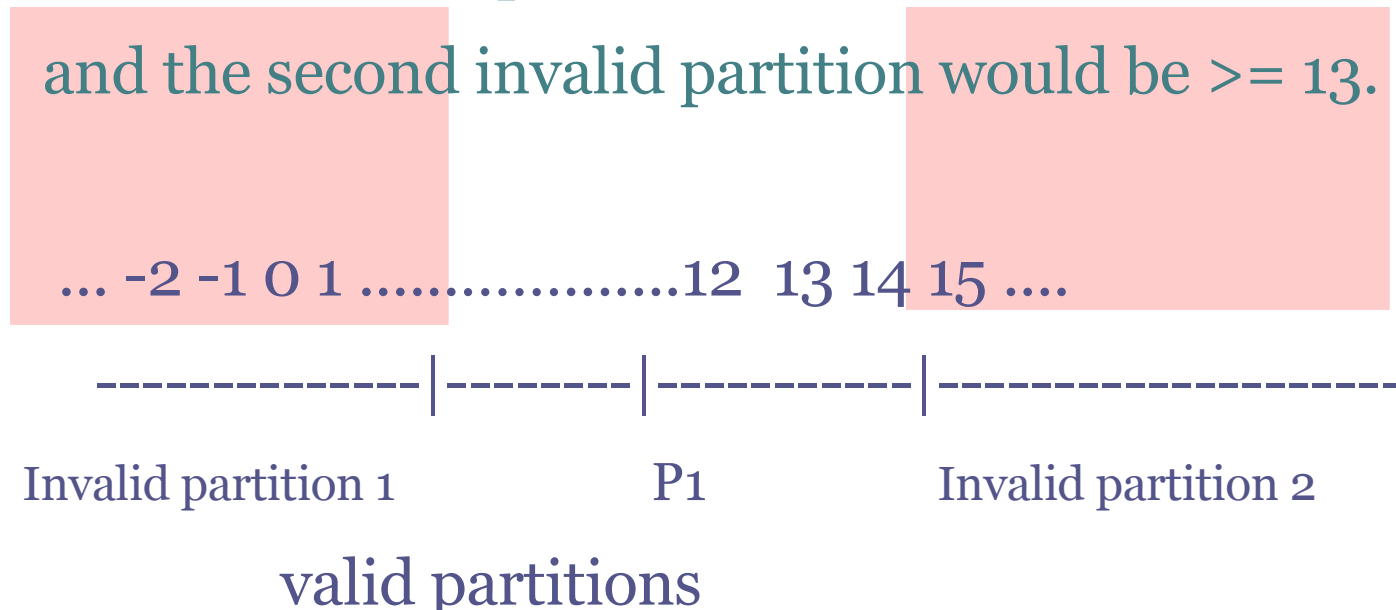
- **To reduce the number of test cases to a necessary minimum.**
 - Only one test case of each partition is needed to evaluate the behaviour of the program for the related partition.
 - To use more or even all test cases of a partition will not find new faults in the program.
 - The values within one partition are considered to be "equivalent".
 - Thus the number of test cases can be reduced considerably
- **To select the right test cases to cover all possible behaviour**
 - you also find the so called "dirty" test cases.
 - An inexperienced tester may be tempted to use as test cases the input data range and forget to select some out of the invalid partitions.
 - This would lead to a huge number of unnecessary test cases on the one hand, and a lack of test cases for the dirty ranges on the other hand.

Equivalence Partition Analysis

- Equivalence partitions are usually derived from the specification of the systems or component 's behaviour.
 - **consider**
 - Input equivalence partitions
 - Output equivalence partitions
- An input has values which are valid and other values which are invalid.
 - These can be used to identify the equivalence partitions
- If you had defined a function which has to pass the parameter "month" of a date.
 - *What are the valid and invalid values?*

Equivalence partitions

- The valid range for the month is 1 to 12, standing for January to December.
 - This valid range is called a valid partition.
- there are two partitions of invalid ranges.
 - The first invalid partition would be ≤ 0
 - and the second invalid partition would be ≥ 13 .



Equivalence Partitioning

- Divide the domain of all possible inputs into classes of equivalent inputs
- input conditions

Type of input condition	Number of Valid equivalence classes needed	Number of Invalid equivalence classes needed
range	1	2
value	1	2
set	1	1
Boolean (true/false)	1	1

- Construct one test case for each class

Equivalence Partitions - *for value ranges*

- An input has certain ranges which are valid and other ranges which are invalid.
- An example is the value range for the month parameter that we looked at earlier
 - The valid range for the month is 1 to 12, standing for January to December. This is the valid partition
 - There are 2 invalid ranges
 - these are the invalid partitions: ≤ 0 and ≥ 13

Equivalence Partitions - *for a set*

- Test particular input item matches a set of values
 - If each case will be treated the same way
 - Identify **one valid class** for values **in** the set
 - and **one invalid class** representing values **outside** the set
 - E.g valid course codes for semester 1 are: (COMP1209, COMP1202, COMP1216, COMP1201)

Valid class: code *is* one of (COMP1209, COMP1202, COMP1216, COMP1201)

Invalid class: code *not one of* (COMP1209, COMP1202, COMP1216, COMP1201)

- If each case will be treated differently, Identify:
 - **one valid equivalence class** for each element and
 - **one invalid equivalence class** for values outside the set
 - E.g valid boat codes are (catamaran, dinghy, rib)
 - **Valid class is** catamaran
 - **Valid class is** dinghy
 - **Valid class is** rib
 - **Invalid class is not** one of (catamaran, dinghy, rib)

Boundary Value Analysis - Selecting Values for Test Data for each Equivalence Class

- Once equivalence classes (partitions) are identified, we can think about test data values
 - We need some values that fall into each class
- Select values at the boundaries and somewhere in the middle of each class
 - This called **Boundary Value Analysis**
 - If these data points produce correct results, it is fairly safe to assume that other values will also be correctly processed
- **NOTE:** Correct operation at the other points is not guaranteed but we have to find sensible ways to limit testing

Boundary Value Analysis

- Using the domain of all possible inputs and outputs
- Divide the domain into valid and invalid classes
- For each boundary: construct test cases with these inputs

Test case	Test case Inputs
1	Boundary value as input
2	Slightly less than boundary value
3	Slightly less than boundary value

Test plan for a student enrolment system

A student enrolment system:

- the system accepts students between 16 and 99 years of age
- Find the equivalence partitions
- Use boundary value analysis to identify values for your

Equivalence class	value (age)	Expected result (message)	Pass/fail

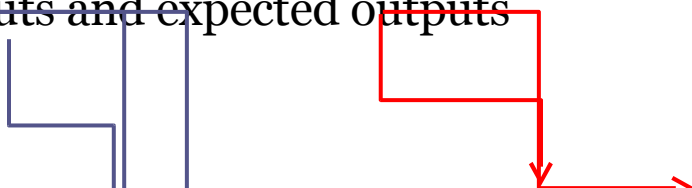
Using Equivalence Partitioning & Boundary Value Analysis for the userID problem

- Going back to our first example: A user ID consists of two characters < 1 alpha char> < 1 digit>
 - The < 1 alpha char> consists of the characters A..Z or a..z
 - < 1 digit> consists of the characters 0..9
- What test cases do you think that you might need to ensure that software processes a user ID correctly - using equivalence partitioning (and boundary value analysis?)

Loan Interest calculator example

Equivalence Partitions

Build a table for your equivalence classes:
showing inputs and expected outputs



<i>Equivalence class</i>	<i>Interest rate</i>	<i>Months</i>	<i>Amount</i>	<i>Re-payment</i>	<i>Message</i>

Selecting Values - Valid Partition

<i>Equivalence class</i>	<i>Interest rate</i>	<i>Number of Months</i>	<i>Amount</i>	<i>Re-payment</i>	<i>Message</i>
Valid	1.00	1	1,000	1,000.83	OK
Valid					
Valid					

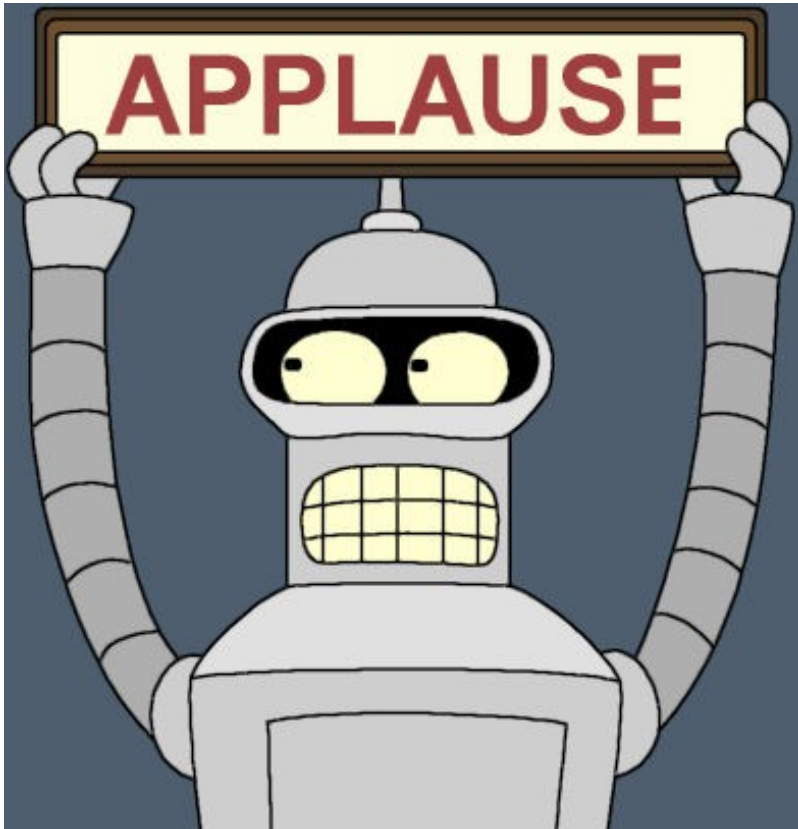
Selecting Values - Interest Error

<i>Equivalence class</i>	<i>Interest rate</i>	<i>Number of Months</i>	<i>Amount</i>	<i>Re-payment</i>	<i>Message</i>
Interest invalid	0.99	1	1000	N/A	Interest Error
Interest invalid				N/A	Interest Error
Interest invalid				N/A	Interest Error
Interest invalid				N/A	Interest Error
Interest invalid				N/A	Interest Error

Summary

- Test cases are the smallest steps
- Test cases are assembled into scenarios
 - Represent a typical usage of the system
- Equivalence Partitioning
 - Used to find groups of inputs / outputs that produce similar behaviour
- Boundary Value Analysis

Happiness café?



Oh for crying out loud. I never wanted to be a dev anyway. What I *really* want is to be a... tester!
I'm a tester and I'm okay,

I sleep all night and I work all day.

(He's a tester and he's okay,

he sleeps all night and he works all day.)

I look for bugs, I log those bugs, I do it all

I'm a tester and I'm okay,

I sleep all night and I work all day.

(He's a tester and he's okay,

he sleeps all night and he works all day.)

I black box test, I white box test, I write docs