CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Lecture 5: Test Case Design

Gregory Gay
DIT636/DAT560 - February 2, 2026

Sources of Test Input

Functional Testing (Black Box)

The sort function should yield an array of integers, **sorted in ascending order from smallest to largest**.

Structural Testing (White Box)

```
public int[] sort (int[] unsorted){
    …
    if (unsorted[x] <= unsorted[y]) {
        …
    }
    …
}
```

True

False

Sources of Input

- **Functional (Black Box) Test Design**
  - Use documentation of system behavior to design tests.
    - Requirements, comments, user manuals, intuition.
  - Reflects what code *should* do, not what it currently does.
    - Treated as a “black box”: input -> code -> output
  - Normal form of test design.
    - Complemented by structural testing.
  - Tests can be designed before code is written.
    - (**test-driven development**)

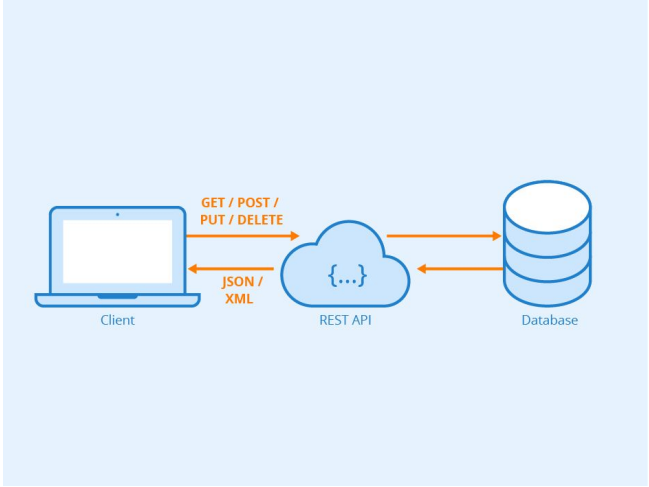
Sources of Input

- **Structural (White Box) Test Design**
  - Input chosen to exercise code in specific way.
    - Oracles still based on requirements.
  - Usually based on **adequacy criteria**:
    - Checklists based on program elements.
    - **Branch Coverage** - All conditional statements evaluate to true/false.
  - Fill in the gaps in functional test design.

Today's Goals

- Introduce API testing, using Postman
- Process for functional test case design.
  - Identify testing targets.
  - For each testing target, identify choices.
  - For each choice, identify representative values.
  - Generate test specifications.
  - Instantiate concrete test cases.

Creating API Tests with Postman



Postman

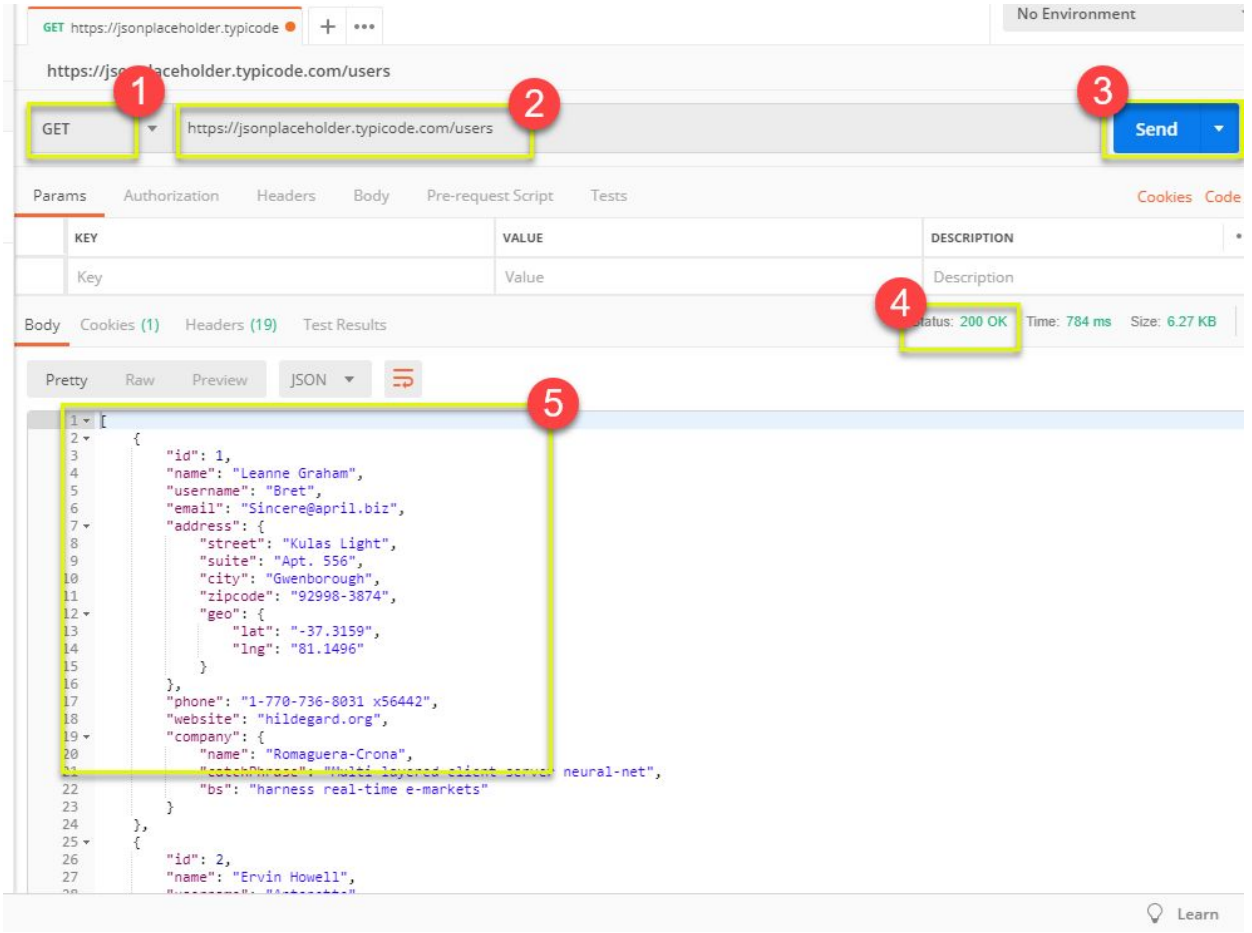
- Testing framework for systems with a REST API.
  - REST: interface with **endpoints** we can interact with.
  - At an endpoint, we can send HTTPS request to:
    - **GET** information
    - **DELETE** information
    - **POST** information into a new resource (i.e., create a new entry)
    - **PUT** information in a resource (i.e., update an existing entry)
- Can create requests and tests using Postman.

Writing Tests in Postman

- Each tab is a request.
- The request defines **test input**.
  - GET/POST/PUT/DELETE
  - Resource acted upon
  - Params, Authorization, Headers, Body
- Post-response scripts tab defines **test oracles**.
  - Write small JavaScript methods to check correctness of output.

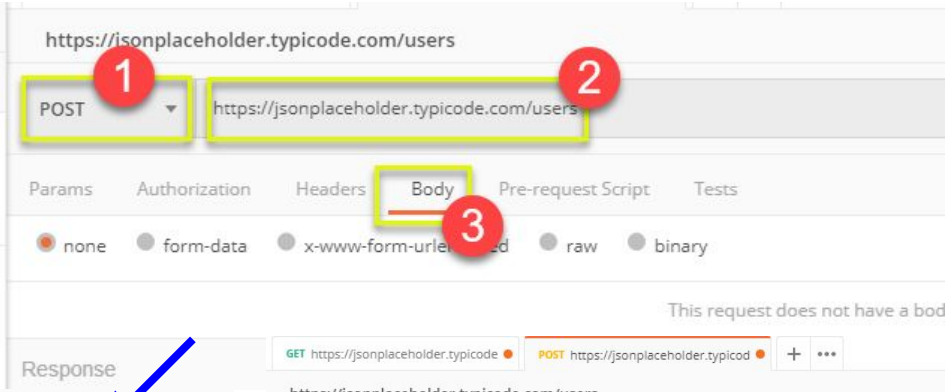
Input - GET

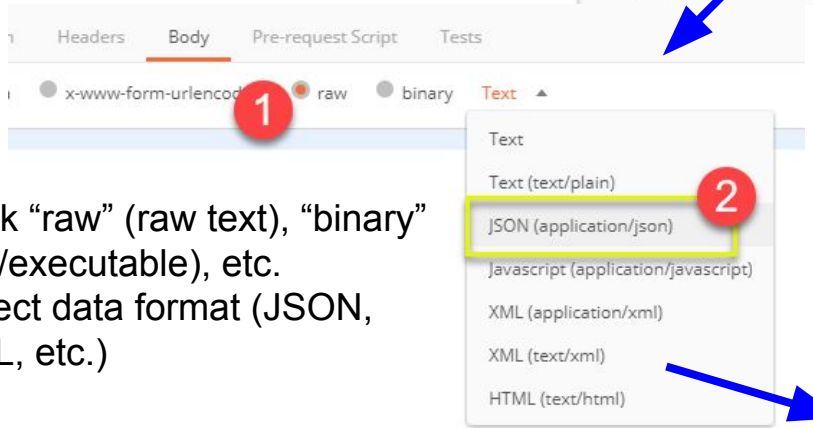
1. Select GET as the request type.
2. Set the resource URL.
3. Click “Send”
4. The response status is indicated.
5. The body contains the returned information.



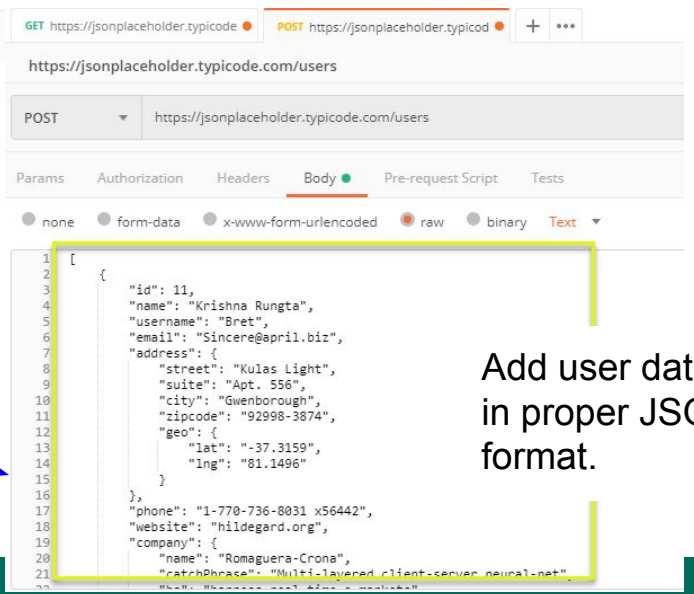
Input - POST

1. Set request to POST.
2. Set the endpoint URL.
3. Select the “Body” tab.





1. Click “raw” (raw text), “binary” (file/executable), etc.
2. Select data format (JSON, XML, etc.)

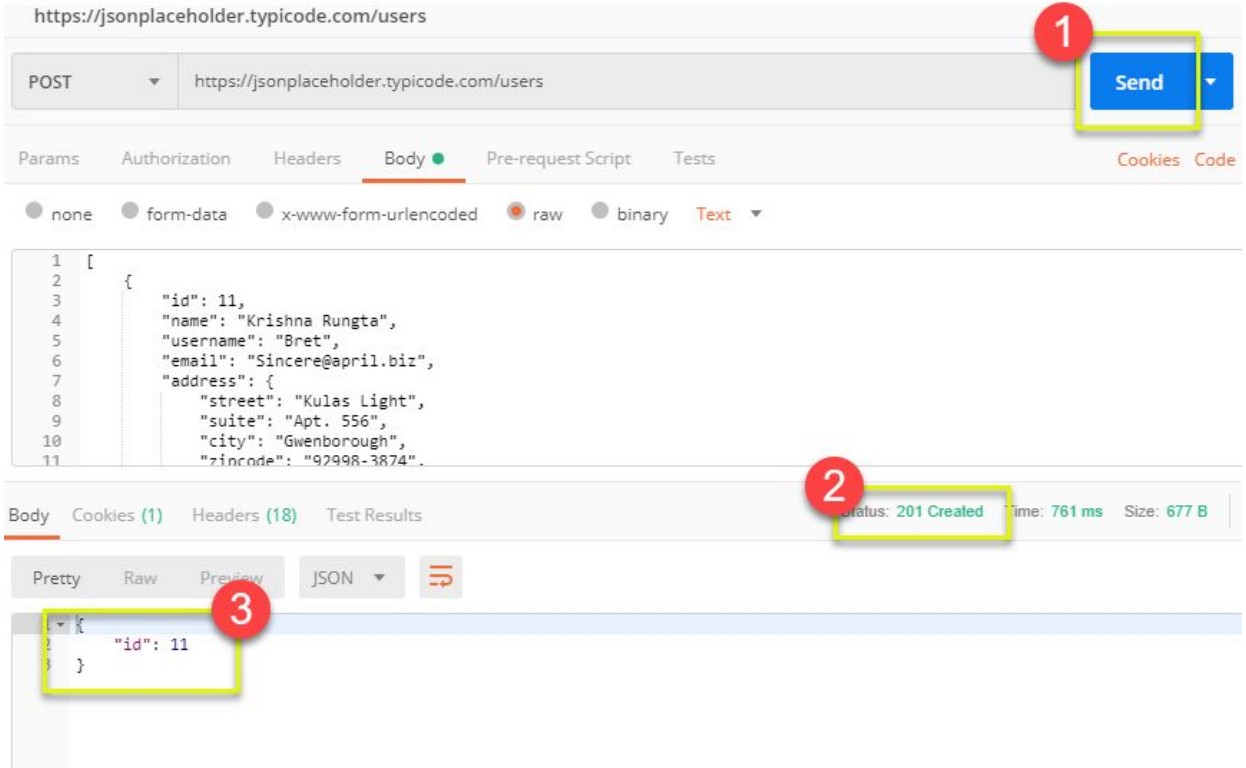


Add user data in proper JSON format.

Adapted from https://www.guru99.com/postman-tutorial.html

Output - POST

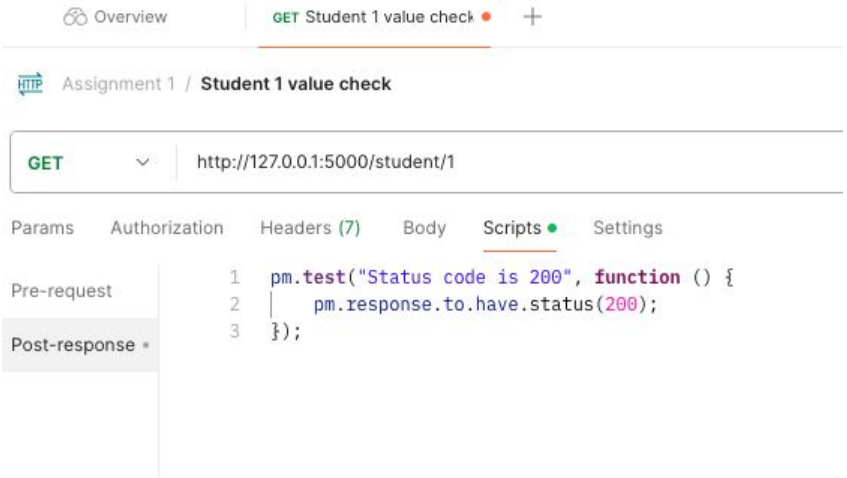
1. Click Send to send request.
2. Response status is indicated (201, data created)
3. Body indicates record “11” was created.

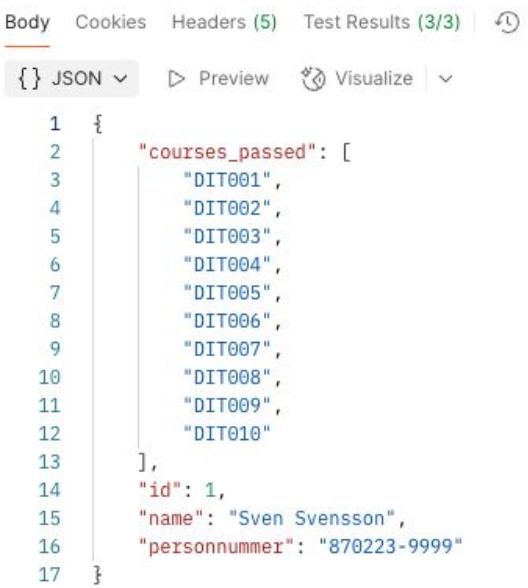


Creating Test Oracles

- Post-response scripts tab allows creation of JavaScript blocks used to verify results.
  - These are **test oracles**.
  - Embed expectations on results and code to compare expected and actual values.
- Use `pm.test` library to create assertions on output.
  - https://learning.postman.com/docs/writing-scripts/script-references/test-examples/ (many example scripts!)

Oracle Example - Status Check





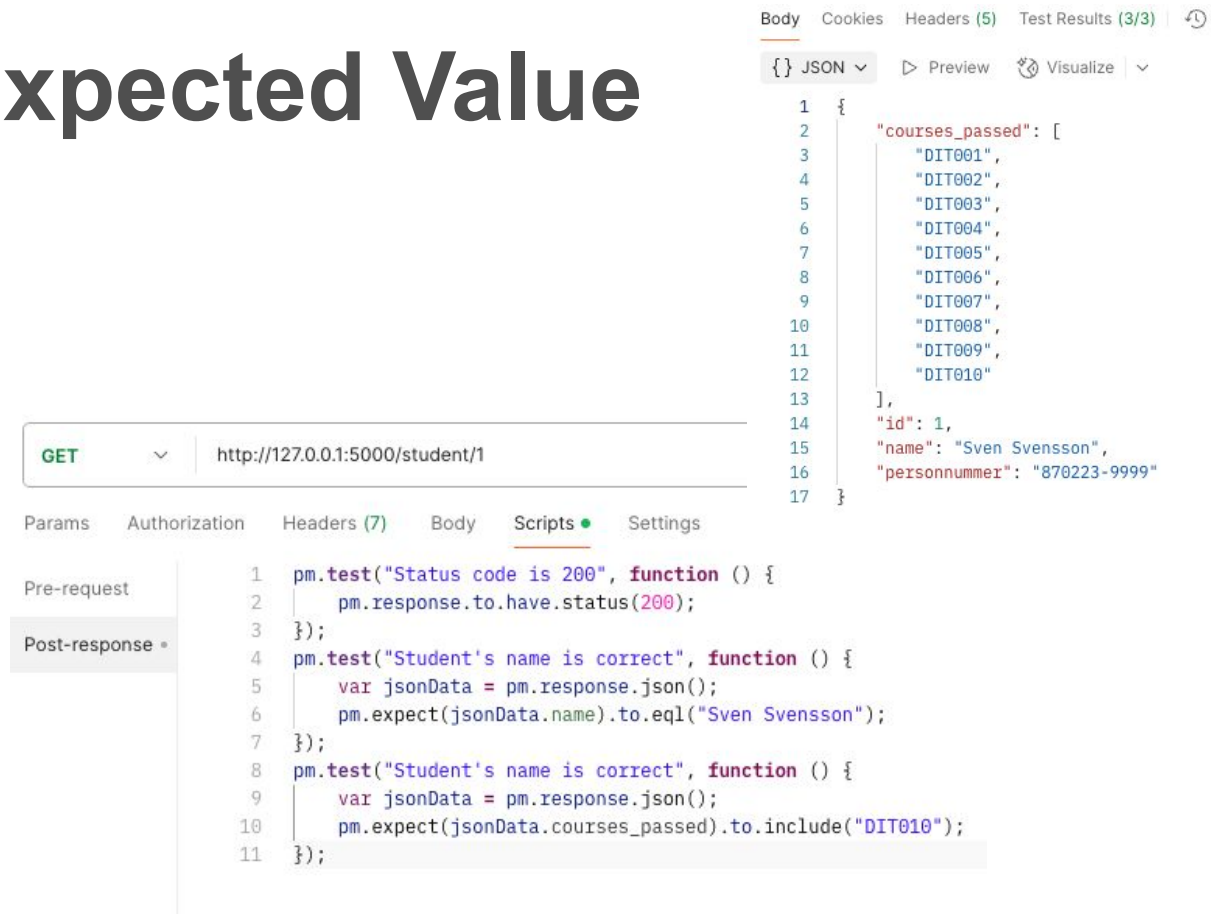
- Create test in post-response scripts tab.
- Snippets offer pre-built test oracles.
- Ex. “status code must be 200”

Adapted from https://www.guru99.com/postman-tutorial.html

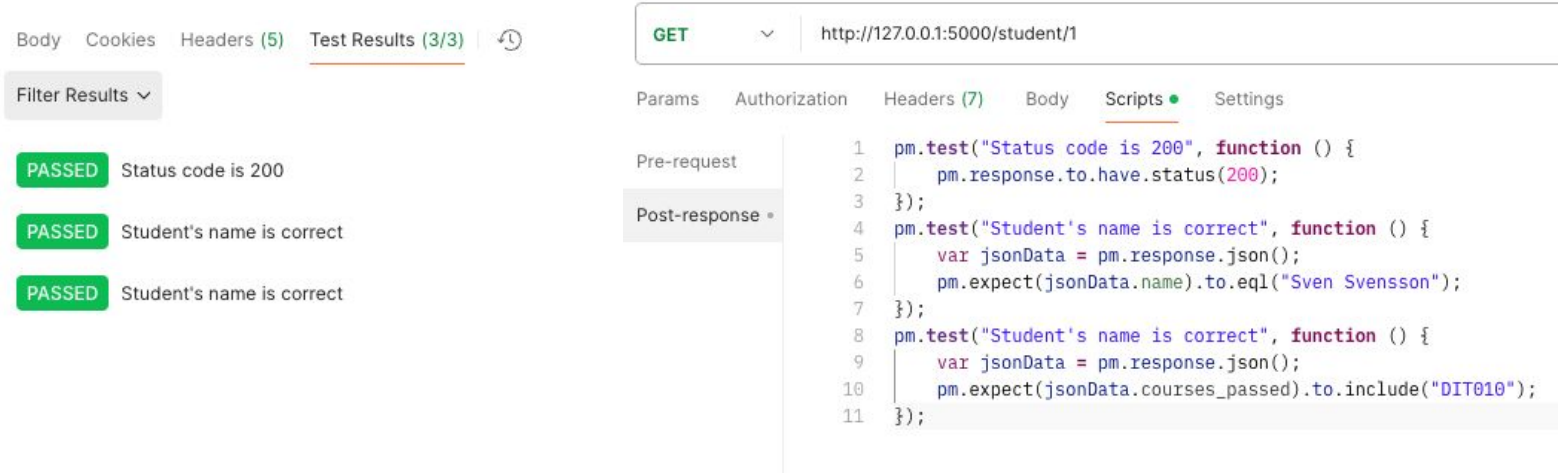
Example - Expected Value

- Snippets “JSON value check”, “Contains String”

- Inserts generic test body.
- Change **test name**, **variable to check** (name), **value to check** (check for name “Sven Svensson”, specific course “DIT010”).



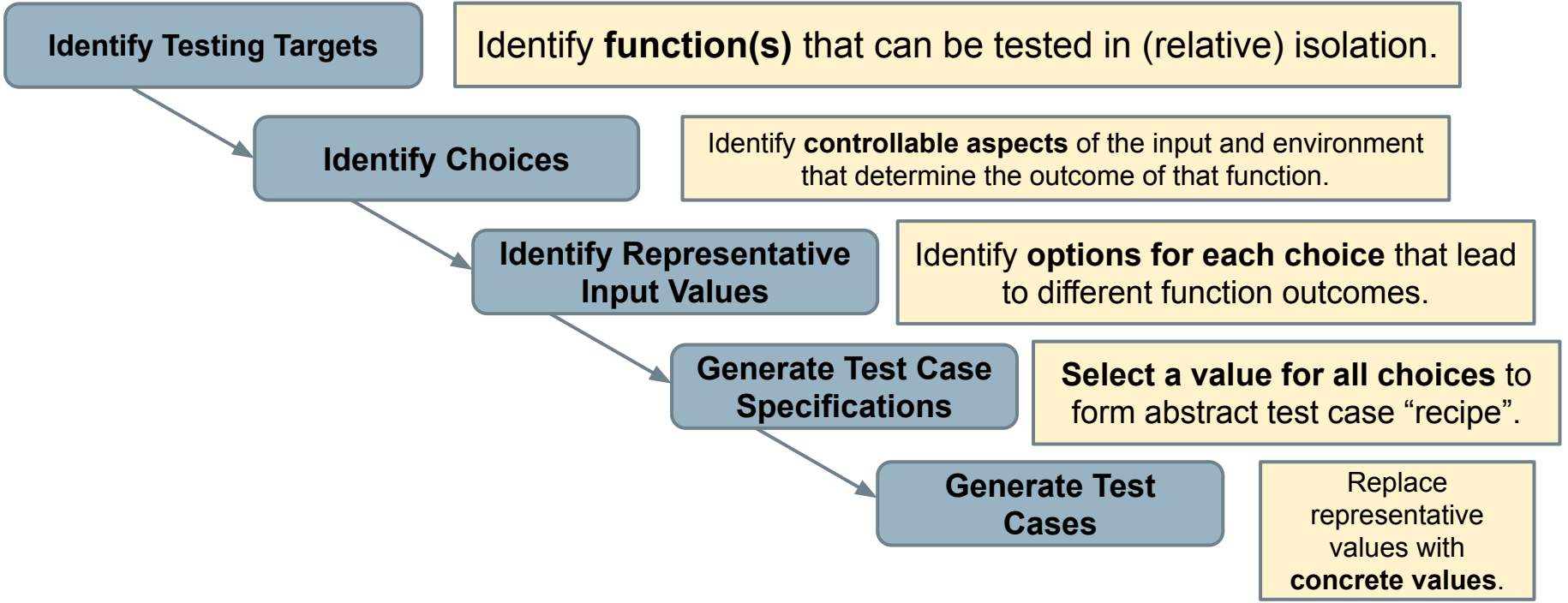
Test Execution Results



- All three tests should pass.
- Status and test names indicated in GUI.

Creating Functional Test Cases

Creating Functional Tests



Independently Testable Functionality

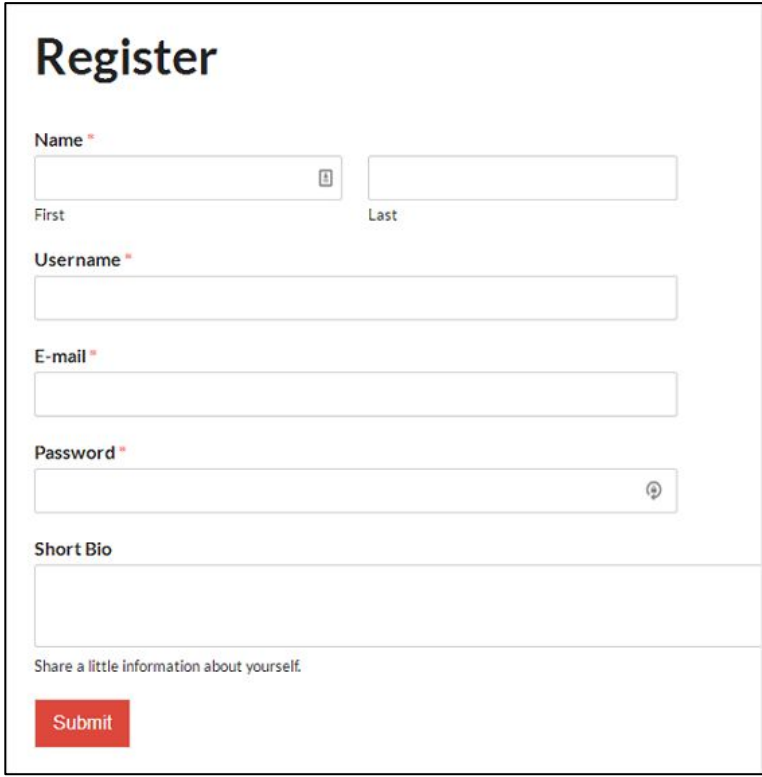
- **Well-defined function(s) that can be tested in (relative) isolation.**
  - Based on the “verbs” - what can we do with this system?
  - Functionality offered by an interface.
  - Depends on the level of testing.
    - Web Forum: Sorted user list can be accessed.
      - System testing: Test through the web interface, examine the complete page loaded by the function (member list, page layout, etc.).
      - Unit testing: Test functions of a class (e.g., sorting function alone).

Identify Choices

- What choices do we make when invoking target?
  - **Anything we *control* that can change the outcome.**
  - What are the ***input parameters*** to that feature?
  - What ***configuration choices*** can we make?
  - Are there ***environmental factors*** we can vary?
    - Networking environment, file existence, file content, database connection, database contents, disk utilization, …

Ex: Register for Website

- From the input parameters:
  - First Name, Last Name, Username, E-Mail Address, Password, Short Bio
- Other environmental factors:
  - Is there a database connection?
  - Is this user already in the database?



Parameter Characteristics

- Identify choices by understanding how parameters are used by the function.
- Type information is helpful.
  - `firstName` is string, database contains `UserRecords`.
- … but context is important.
  - Reject registration if in database.
  - … or database is full.
  - … or database connection down.

Parameter Context

- Input parameter can be split into multiple “choices” based on context.
  - A database affects User Registration, but there is **more than one choice**.
    - Choice: Is there a database connection?
    - Choice: Is there already a record for the user?
    - Choice: How full is the database storage?

Ex: Binary Search

```
Boolean binarySearch(String[] array, String toFind)
```

- **Choice: How many items are in the array?**
  - (Empty array might behave differently than one with several items)
  - (Could also provide a null pointer instead of a real array)
- **Choice: Is the array sorted?**
  - (Binary search assumes the array is sorted)
- **Choice: Is the string in the array?**
  - (Different function outcomes)

Example

Class Registration System

What are some independently testable functions?

- Register for class
- Drop class
- Transfer credits from another university
- Apply for degree

Example - Register for a Class

Input: Route: /registrations/, Method: POST,
Input: { “studentID”: VALUE, “courseID”: VALUE }

Output: Status Code: (201 if registration OK, 200 for input-based errors, others for other errors), JSON message: { “result”: VALUE } (“OK”, error messages)

Example Oracle:
```
pm.test(“Normal Case”, function() {
    pm.response.to.have.status(201);
    var jsonData = pm.response.json();
    pm.expect(jsonData.result).to.eql(“OK”);
});
```

What are the choices we make when we design a test case?

Input: Route: /registrations/, Method: POST,

Input: { “studentID”: VALUE, “courseID”: VALUE }

- Does student meet prerequisites?
- Does the course exist?
- **What else influences the outcome?**

Example Oracle:

```
pm.test(“Normal Case”, function() {
    pm.response.to.have.status(201);
    var jsonData = pm.response.json();
    pm.expect(jsonData.result).to.eql(“OK”);
});
```

Example - Register for a Class

- During setup, we can influence a student's record and the course records.
  - These are "inputs" to consider.
- How are they used?
  - Has a student already taken the course?
  - Do they meet the prerequisites?
  - Does a course exist?
  - What are the prerequisites of a course.

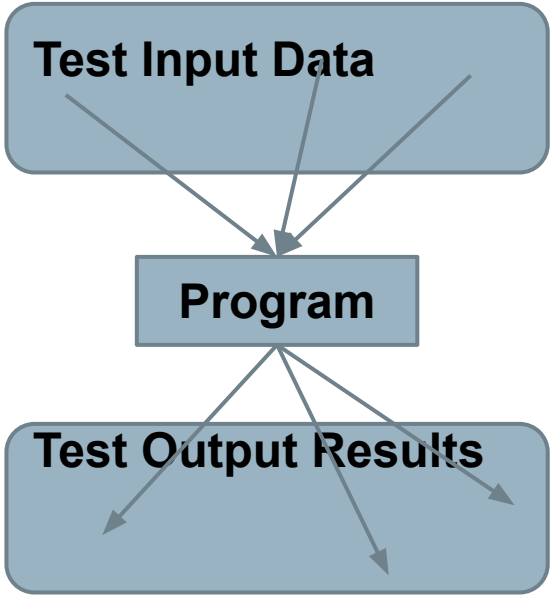
Example - Register for a Class

- **Parameter: studentID**
  - **Choice:** Validity of Student ID
  - **Choice:** Courses Student Has Taken Previously
- **Parameter: courseID**
  - **Choice:** Validity of Course ID
  - **Choice:** Prerequisites of Course ID

Let’s take a break.

Identifying Representative Values

- We know the functions.
- We have choices for each.
- **Representative values** are the options for each choice.



Ex: Binary Search

```
Boolean binarySearch(String[] array, String toFind)
```

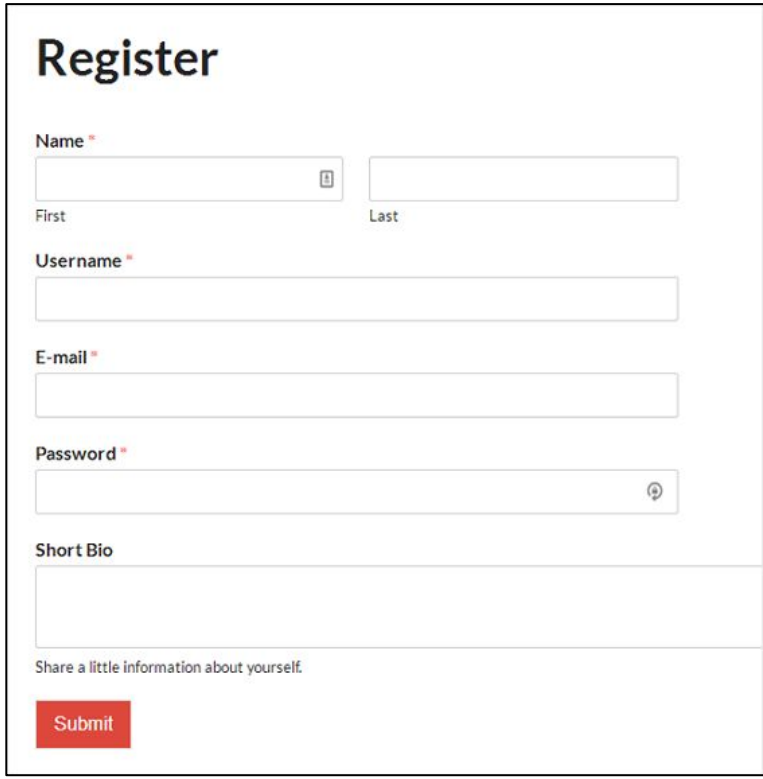
- **Choice: How many items are in the array?**
- **Choice: Is the array sorted?**
  - **Yes**
  - **No**

- **Choice: Is the string in the array?**
  - **Yes**
  - **No**

- **Choice: How many items are in the array?**
  - Null pointer
  - 0
  - 1
  - 2
  - 3
  - 4
  - 5
  - …
  - 1000000000000

Ex: Register for Website

- "Value of X" are **choices**.
  - X = first name, username, etc.
- What are the **representative values** for each choice?
  - *First name could be any string!*



Exhaustive Testing

Take the arithmetic function for the calculator:

```
add(int a, int b)
```

- How long would it take to exhaustively test this function?

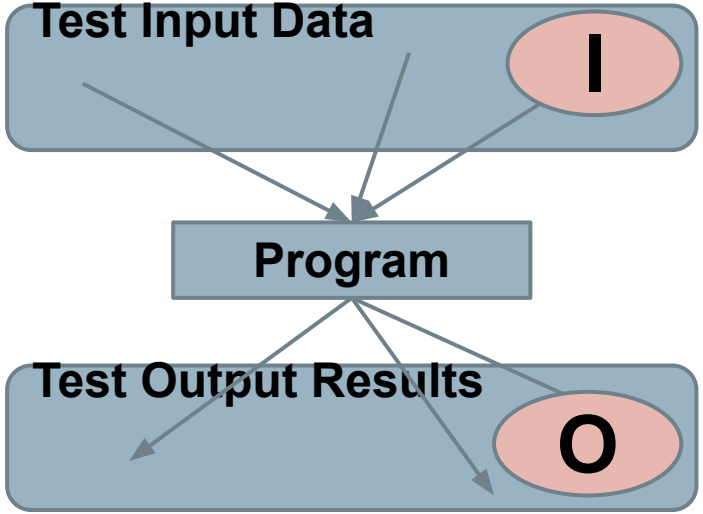
2^{32} possible integer values for each parameter.
= 2^{32} x 2^{32} = 2^{64} combinations = 10^{13} tests.

1 test per nanosecond
= 10^5 tests per second
= 10^{10} seconds

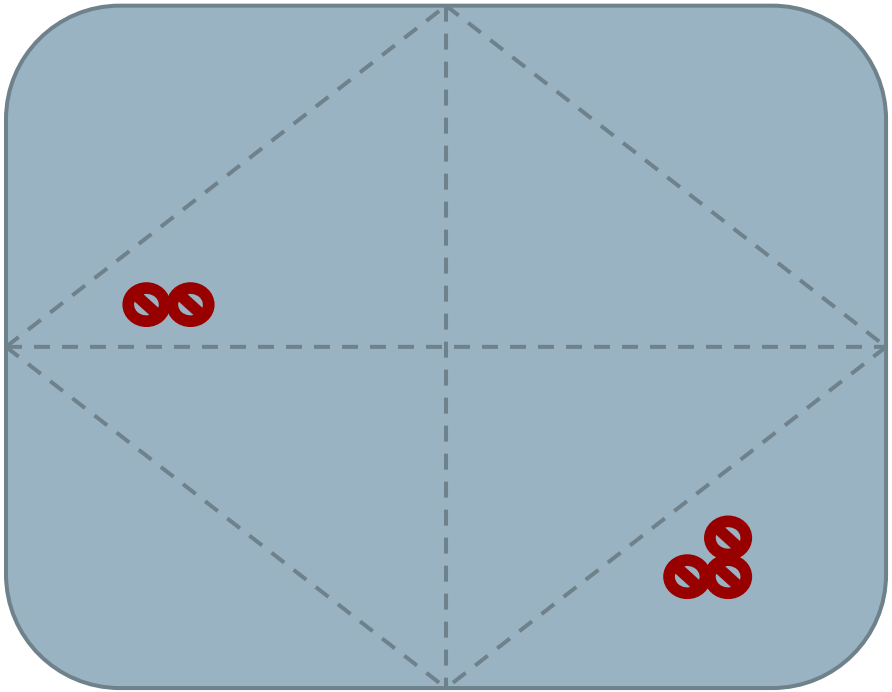
or… about 600 years!

Not all Inputs are Created Equal

- Many inputs lead to same outcome.
- Some inputs better at revealing faults.
  - We can't know which in advance.
  - Tests with different input better than tests with similar input.



Input Partitioning



- Consider possible values for a variable.
- Faults sparse in space of all inputs, but dense in parts where they appear.
  - Similar input to failing input also likely to fail.
- Try input from partitions, hit dense fault space.

Equivalence Class

- Divide the input domain into **equivalence classes**.
  - Inputs from a group interchangeable (trigger same outcome, result in the same behavior, etc.).
  - If one input reveals a fault, others in this class (probably) will too. In one input does not reveal a fault, the other ones (probably) will not either.
- Partitioning based on intuition, experience, and common sense.

Choosing Input Partitions

- What are the function outcomes?
- Ranges of numbers or values.
- Membership in a logical group.
- Time-dependent equivalence classes.
- Equivalent operating environments.
- Data structures.
- Partition boundary conditions.

Function Outcomes

- Look at the outcomes and group input by the outcomes they trigger.

```
Boolean binarySearch(String[] array, String toFind)
```

- **Choice: How many items are in the array?**
  - Null pointer
  - 0
  - 1
  - 2
  - 3
  - 4
  - 5
  - …
  - 1000000000000

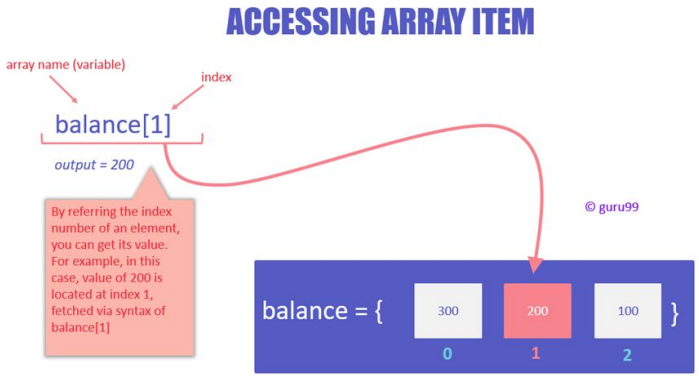
- **Choice: How many items are in the array?**
  - Null pointer (could lead to exception)
  - 0 (could lead to exception/warning)
  - 1+ (normal outcomes)

Data Type

- Try values commonly misused, based on data type.
  - Ex: Integer
    - Basic Split: < 0, 0, >0
    - If conversions take place from String -> Integer, use a non-numeric string.
- Also split based on how variable is used.
  - Integer intended to be 5-digit:
    - < 10000, 10000-99999, >= 100000

Data Type

- Data structures prone to certain types of errors.
- For arrays or lists:
  - Only a single value.
  - Different sizes and number filled.
  - Order of elements: access first, middle, and last elements.



Data Type

```
Boolean binarySearch(String[] array, String toFind)
```

- **Choice: How many items are in the array?**
  - Null pointer (could lead to exception)
  - 0 (could lead to exception/warning)
  - **1 (single item collections often misused)**
  - **2+, # items == array size (normal outcomes)**
  - **2+, # items < array size (could be issues if array is not full)**

Operating Environments

- Environment may affect behavior of the program.
- Environmental factors can be partitioned.
  - Available memory may affect the program.
  - Processor speed and architecture.
  - Client-Server Environment
    - No clients, some clients, many clients
    - Network latency
    - Communication protocols (SSH vs HTTPS)

Timing Partitions

- Timing and duration of input can be as important as value.
  - Timing often implicit input.
    - Trigger an electrical pulse 5ms before a deadline, 1ms before the deadline, exactly at the deadline, and 1ms after the deadline.
    - Close program before, during, and after the program is writing to (or reading from) a disc.

Quality Considerations

- Input partitions likely to affect quality goals.
  - **Performance**: Input likely to lead to performance issues.
    - Ex: Remove resources, large input that will take awhile to process
  - **Security**: Input that attacker could apply.
    - Ex: Code injection in XML input.

Quality Considerations

```
Boolean binarySearch(String[] array, String toFind)
```

- **Choice: How many items are in the array?**
  - Null pointer (could lead to exception)
  - 0 (could lead to exception/warning)
  - 1 (single item collections often misused)
  - 2+, # items == array size (normal outcomes)
  - 2+, # items < array size (could be issues if array is not full)
  - **10000 (could lead to performance issues)**

Input Partition Example

What are the input partitions for:

```
max(int a, int b) returns (int c)
```

We could consider `a` or `b` in isolation:

```
a < 0, a = 0, a > 0
```

Consider combinations of `a` and `b` that change outcome:

```
a > b, a < b, a = b
```

Example - Register for a Class

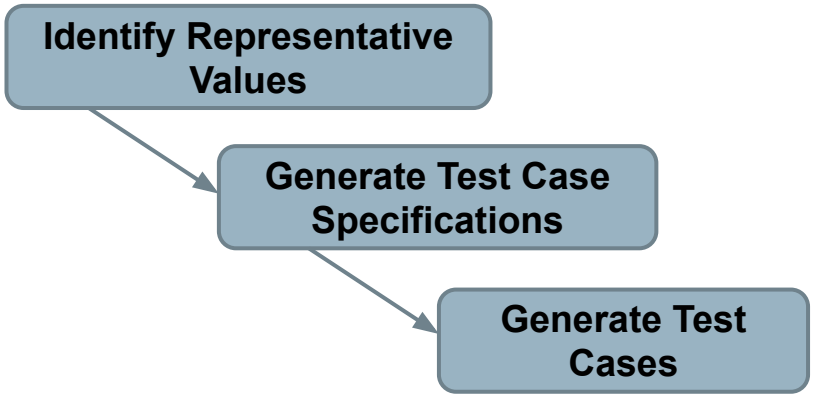
Parameter: studentID

- Validity of Student ID
  - Active Student
  - Inactive Student
  - Non-Existent Student
- Courses Student Has Taken Previously
  - Matches Prerequisites
  - Does Not Match Prerequisites

Parameter: courseID

- Validity of Course ID
  - Existing Course
  - Non-Existent Course
- Prerequisites of Course ID
  - Only Courses Taken By Student
  - Only Courses Not Taken By Student
  - Some Courses Taken by Student

Revisit the Roadmap



For each choice:

1. Partition options into representative values.
2. Choose a value for each choice to form a test specification.
3. Assign concrete values to create test cases.

Basic Test Specification

```
// Set Up
    POST /studentRecords/VALUE, {  … “status”: VALUE, “coursesTaken”: [VALUES]}
    POST /courses/VALUE, { … “prerequisites”: [VALUES] }
// Attempt to register for a course
    POST /registrations/, { “studentID”: VALUE, “courseID”: VALUE }
// Check the result of registration
    pm.test(“Normal Case”, function() {
        pm.response.to.have.status(VALUE);
        var jsonData = pm.response.json();
        pm.expect(jsonData.result).to.eql(VALUE);
    });
```

Forming Specification

Parameter: studentID

- Validity of Student ID
  - Active Student
  - Inactive Student
  - Non-Existent Student
- Courses Student Has Taken Previously
  - Matches Prerequisites
  - Does Not Match Prerequisites

Parameter: courseID

- Validity of Course ID
  - Existing Course
  - Non-Existent Course
- Prerequisites of Course ID
  - Only Courses Taken By Student
  - Only Courses Not Taken By Student
  - Some Courses Taken by Student

Test Specifications:

- Active, Matches, Existing, Only Taken
- Active, Does Not Match, Existing, Only Not Taken
- Active, Does Not Match, Existing, Some Taken
- Active, - , Non-Existing, -
- Inactive, Matches, Existing, Only Taken
- Inactive, Does Not Match, Existing, Only Not Taken
- Inactive, Does Not Match, Existing Some Taken
- Inactive, - , Non-Existing, -
- Non-Existing, -, Existing, -
- Non-Existing, -, Non-Existing, -
- …

Specifications: 3 * 2 * 2 * 3 = 36 - Illegal Combinations

Generate Test Cases

Specification:

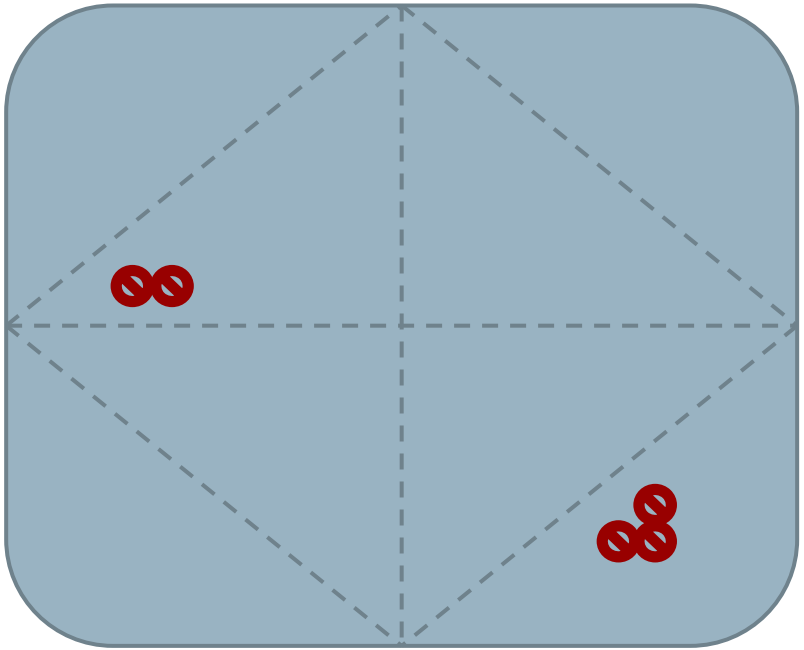
Active, Matches, Existing, Only Taken

```
// Set Up
    POST /studentRecords/ggay, {“status”: active, “coursesTaken”: [“DIT050”, “DIT360”]}
    POST /courses/DIT636, { … “prerequisites”: [“DIT360”] }
// Attempt to register for a course
    POST /registrations/, { “studentID”: ggay, “courseID”: DIT636}
// Check the result of registration
    pm.test(“Normal Case”, function() {
        pm.response.to.have.status(201);
        var jsonData = pm.response.json();
        pm.expect(jsonData.result).to.eql(“OK”);
    });
```

- Fill in concrete values that match the representative values classes.
- Can create MANY concrete tests for each specification.

Boundary Values

- Errors tend to occur at the boundary of a partition.
- Remember to select inputs from those boundaries.

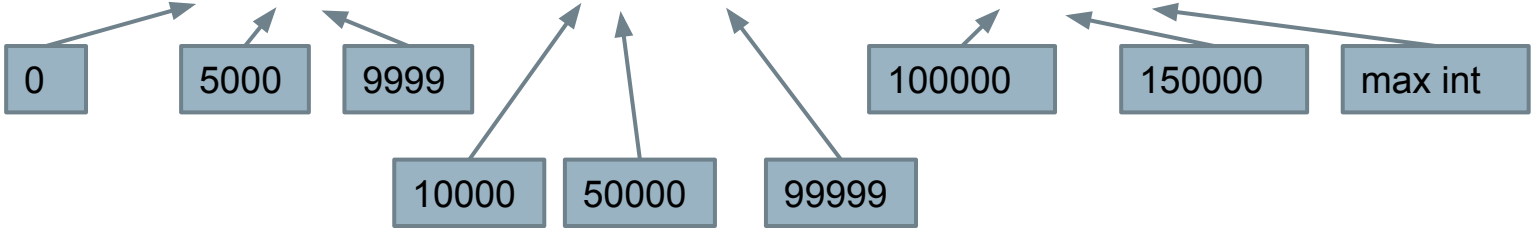


Boundary Values

Choose test case values at the boundary (and typical) values for each partition.

- If an input is intended to be a 5-digit integer between 10000 and 99999, you want partitions:

<10000, 10000-99999, >100000



Example - Message Board Creation

`createMessageBoard (String name, String description, Boolean public)`

- Returns `true` if board created, `false` otherwise.
  - User requesting must be an admin, board must not exist, name and description must not contain banned words.
  - Exception can be thrown if error.
  - Connects to user database, JSON of existing boards, JSON of banned words.

Example - Message Board Creation

- **Choice:** User
  - Admin
  - Not an Admin
- **Choice:** Board Name
  - Valid, does not exist
  - Exists already
  - Contains banned word
  - Blank string
  - Null
- **Choice:** Description
  - Contains banned word
  - Does not contain banned word
  - Blank string
  - Null
- **Choice:** Public
  - Public
  - Private
  - Null

We Have Learned

- Process to create functional tests:
  - Identify **testing targets**.
  - Identify **choices** that influence function outcome.
  - Partition choices into **representative values**.
  - Form specifications by **choosing a value for each choice**.
  - Turn specifications into **concrete** test cases.

Next Time

- Next Time: Test Case Design and Unit Testing
- Exercise Session: Test Case Design

- Assignment 1 - Due Feb 5
  - Based on Lectures 1-3
- Assignment 2 - Due Feb 15
  - Lectures 4-6

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY