FACULDADE ENGENHARIA
Departamento de Informática

Qualidade de Software (14450)

System Testing

(adapted from lecture notes of the “DIT 635 - Software Quality and Testing” unit, delivered by Professor Gregory Gay, at the Chalmers and the University of Gothenburg, 2022)

Today's Goals

- Discuss testing at the system level.
  - System (Integration) Testing versus Unit Testing.
- Understand how interactions can create faults.
- Introduce process for creating System Tests.
  - Identify a Independently Testable Function
  - Identify Choices
  - Identify Representative Values
  - Generate Test Case Specifications
  - Generate Concrete Test Cases
- Examine **how to select system tests** to increase likelihood of detecting interaction faults.
  - Category-Partition Method
  - Combinatorial Interaction Testing

Unit Testing

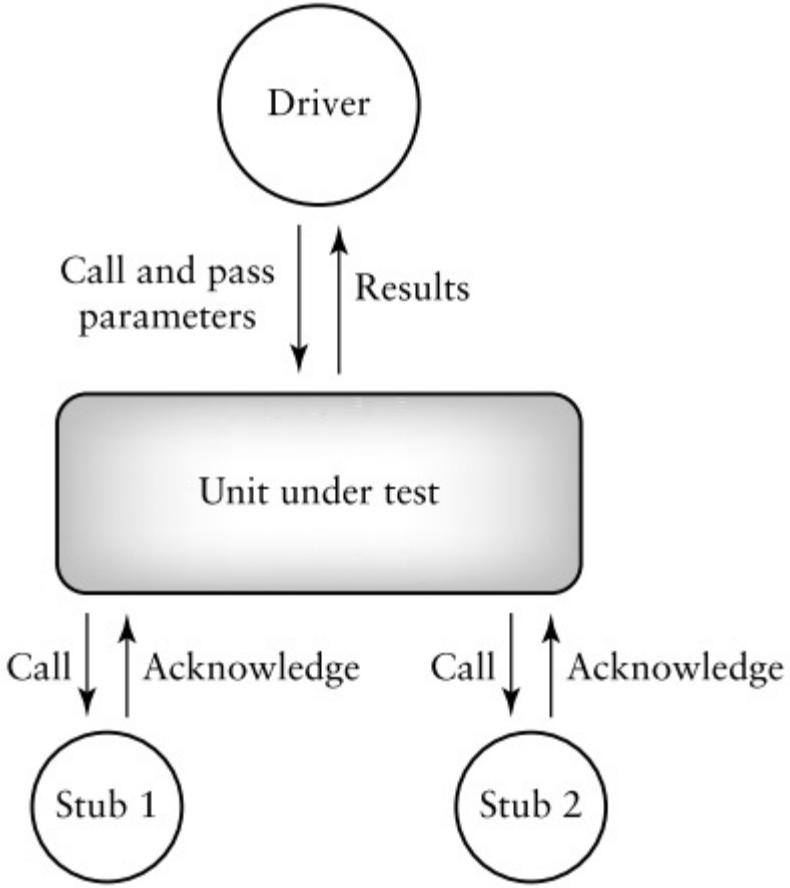
- ✧ Testing the smallest “unit” that can be tested.
  - Often, a class and its methods.
- ✧ Tested in **isolation** from all other units.
  - **Mock** the results from other classes.
- ✧ Test input = method calls.
- ✧ Test oracle = assertions on output/class variables.

Unit Testing

✧ For a unit, tests should:

- Test all “jobs” associated with the unit.
  - Individual methods belonging to a class.
  - Sequences of methods that can interact.
- Set and check value of all class variables.
  - Examine how variables change after method calls.
  - Put the variables into all possible states (types of values).

Unit Testing



Unit Testing - Advantages

- Unit testing increases confidence in changing/ maintaining code. If good unit tests are written and if they are run every time any code is changed, we will be able to promptly catch any defects introduced due to the change. Also, if codes are already made less interdependent to make unit testing possible, the unintended impact of changes to any code is less.

- Codes are more reusable. In order to make unit testing possible, codes need to be modular. This means that codes are easier to reuse.

- Unit testing are more reliable than ‘developer tests’.

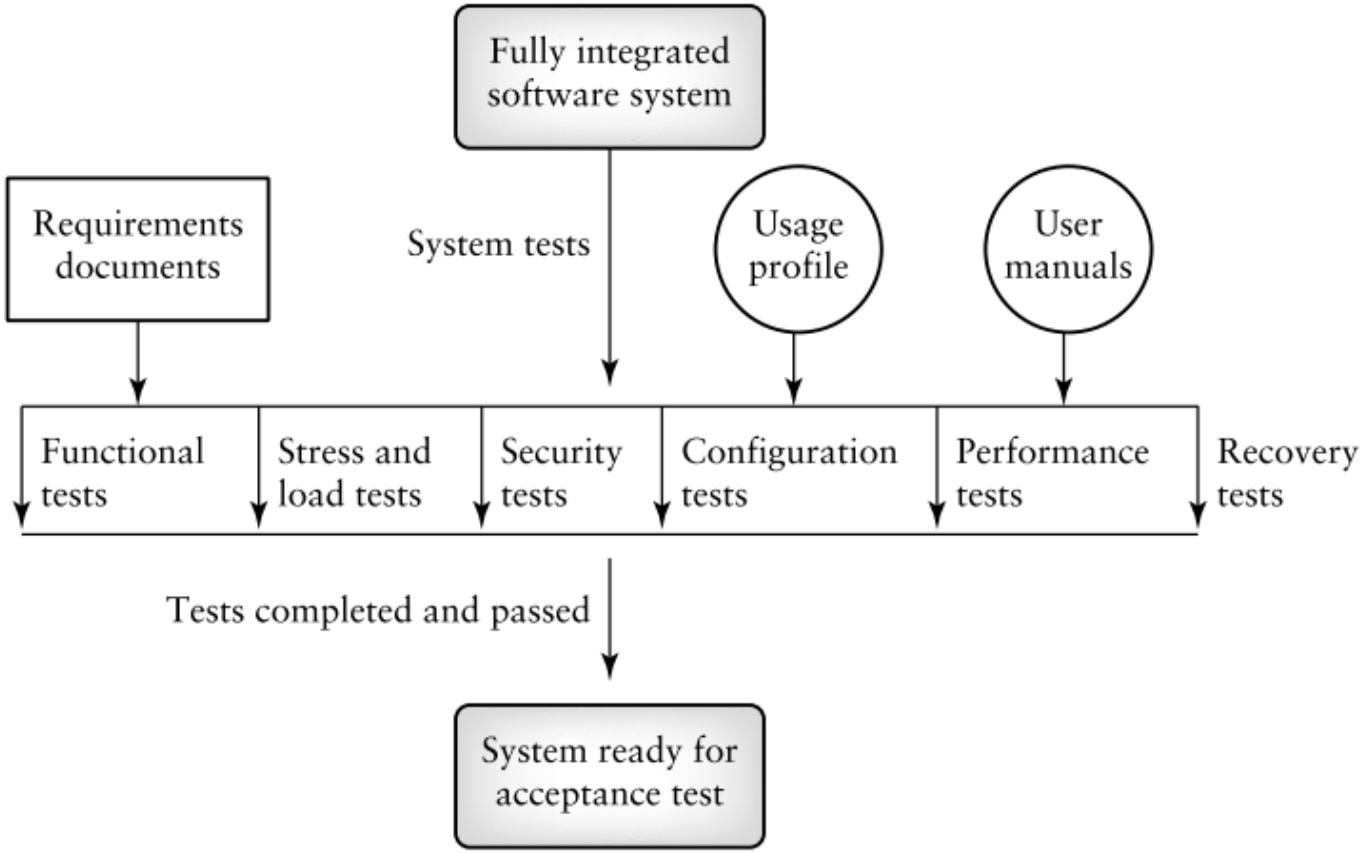
Unit Testing - Advantages

- The effort required to find and fix defects found during unit testing is very less in comparison to the effort required to fix defects found during system testing or acceptance testing.
- The cost (time, effort, destruction, humiliation) of fixing a defect detected during unit testing is lesser in comparison to that of defects detected at higher levels.
- Debugging is easy. When a test fails, only the latest changes need to be debugged. With testing at higher levels, changes made over the span of several days/weeks/months need to be scanned.

System Testing

- ✧ After testing units, test their **integration**.
  - Integrate units in one subsystem.
  - Then integrate the subsystems.
- ✧ Test through a **defined interface**.
  - Focus on showing that functionality accessed through interfaces is correct.
  - Subsystems: “Top-Level” Class, API
  - System: API, GUI, CLI, …

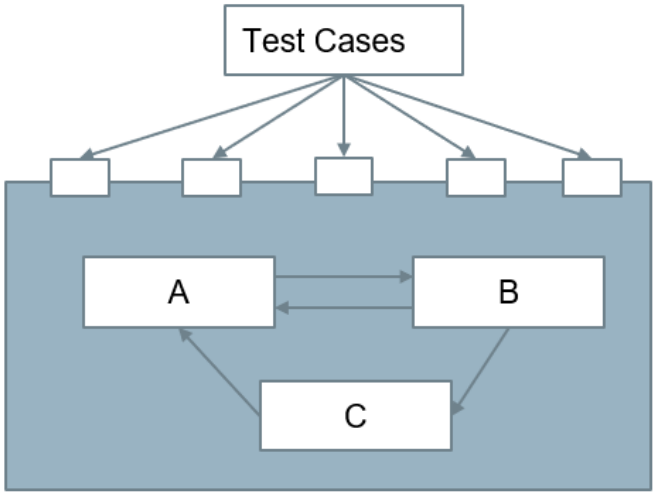
System Testing - Example



System Testing

Subsystem made up classes of A, B, and C. We have performed unit testing...

- ✧ Classes work together to perform subsystem functions.
- ✧ Tests applied to the interface of the subsystem they form.
- ✧ Errors in combined behavior not caught by unit testing.



System Testing

- Functional tests at the system level are used to ensure that the behaviour of the system adheres to the requirements specification. All functional requirements for the system must be achievable by the system.
- Functional tests are black box in nature. The focus is on the inputs and proper outputs for each function.
- This is the only phase of testing which tests both functional and non-functional requirements of the system (e.g. usability, performance, security, reliability, stress and load, …).

Unit vs System Testing

- ✧ Unit tests focus on a **single class**.
  - Simple functionality, more freedom.
  - Few method calls.
- ✧ System tests **bring many classes together**.
  - Focus on testing through an interface.
  - One interface call triggers many internal calls.
    - Slower test execution.
  - May have complex input and setup.

System Testing and Requirements

- ✧ **Tests can be written early in the project.**
  - Requirements discuss high-level functionality.
  - Can create tests using the requirements.
- ✧ Creating tests supports requirement refinement.
- ✧ Tests can be made concrete once code is built.

Interface Types

- ✧ Parameter Interfaces
  - Data passed from through method parameters.
  - Subsystem may have interface class that calls into underlying classes.
- ✧ Procedural Interfaces
  - Interface surfaces a set of functions that can be called by other components or users (API, CLI, GUI).
  - Integrates lower-level components and controls access.

Interface Types

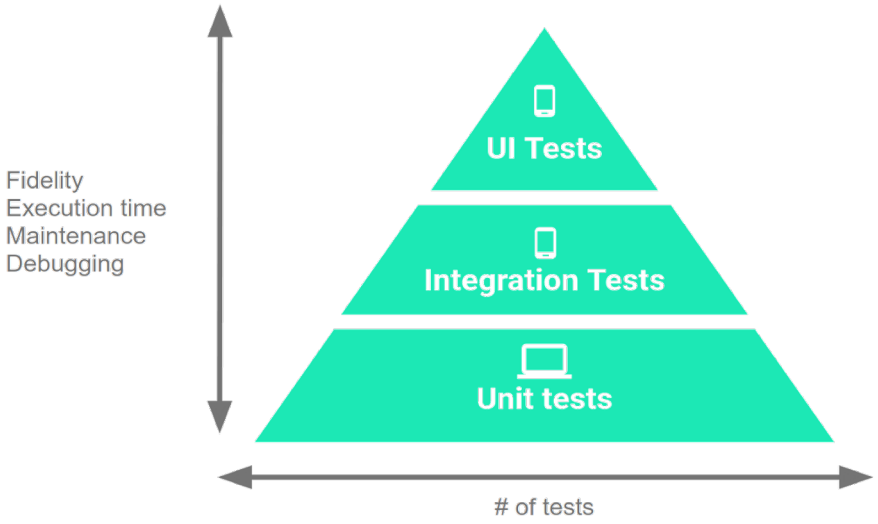
- ✧ Shared Memory Interfaces
  - A block of memory is shared between (sub)systems.
    - Data placed by one (sub)system and retrieved by another.
  - Common if system architected around data repository.
- ✧ Message-Passing Interfaces
  - One (sub)system requests a service by passing a message to another.
    - A return message indicates the results.
  - Common in parallel systems, client-server systems.

Interface Errors

- ✧ Interface Misuse
  - Malformed data, order, number of parameters.
- ✧ Interface Misunderstanding
  - Incorrect assumptions made about called component.
  - A binary search called with an unordered array.
- ✧ Timing Errors
  - Producer of data and consumer of data access data in the wrong order.

Testing Percentages

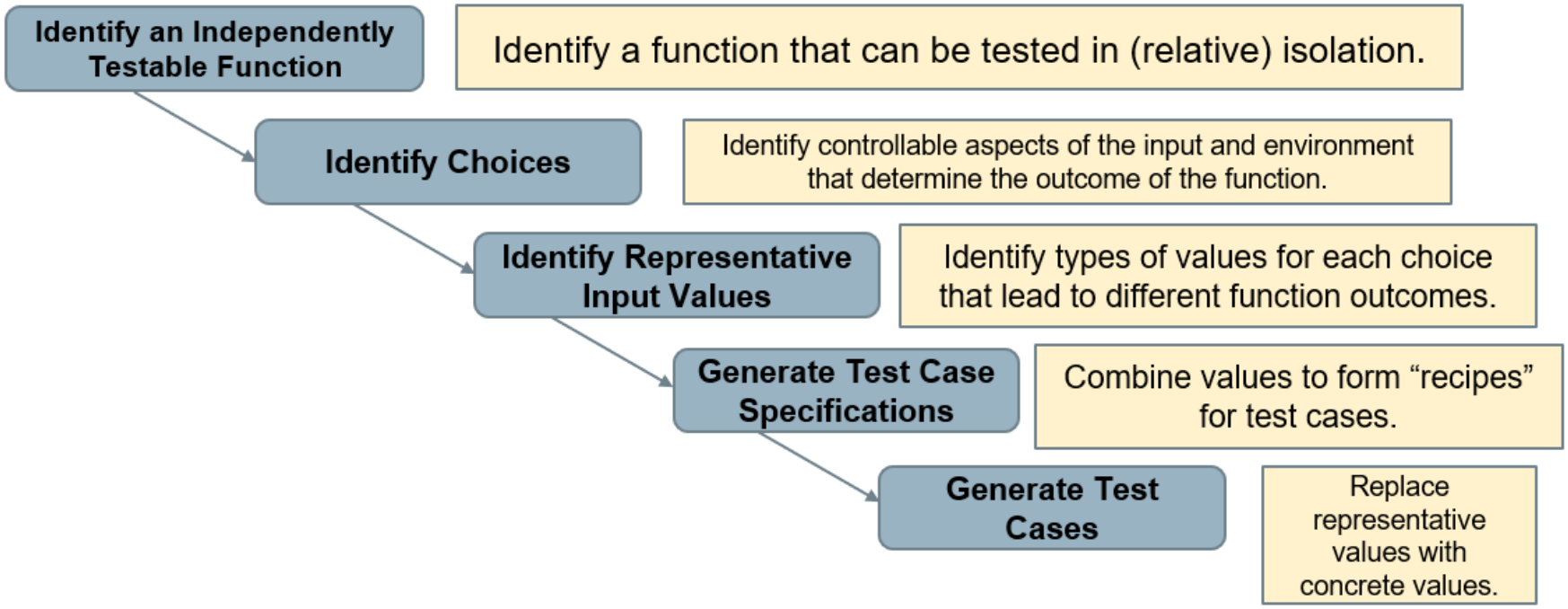
- ✧ Unit tests verify behavior of a single class.
  - 70% of your tests.
- ✧ System tests verify class interactions.
  - 20% of your tests.
- ✧ GUI tests verify end-to-end journeys.
  - 10% of your tests.



Testing

- ✧ 70/20/10 recommended.
- ✧ Unit tests execute quickly, relatively simple.
- ✧ System tests more complex, require more setup, slower to execute.
- ✧ UI tests very slow, may require humans.
- ✧ Well-tested units reduce likelihood of integration issues, making high levels of testing easier.

Creating System-Level Tests



Independently Testable Functionality

Identify an Independently Testable Function

- **A well-defined function that can be tested in (relative) isolation.**
  - Based on the “verbs” - what can we do with this system?
  - The high-level functionality offered by an interface.
  - UI - look for user-visible functions.
    - Web Forum: Sorted user list can be accessed.
    - Accessing the list **is** a testable functionality.
    - Sorting the list is **not** (low-level, unit testing target)

Units and “Functionality”

Identify an Independently Testable Function

- Many tests written in terms of “units” of code.
- An independently testable function is a *capability* of the software.
  - Can be at class, subsystem, or system level.
  - **Defined by an interface.**

Identify Input Choices

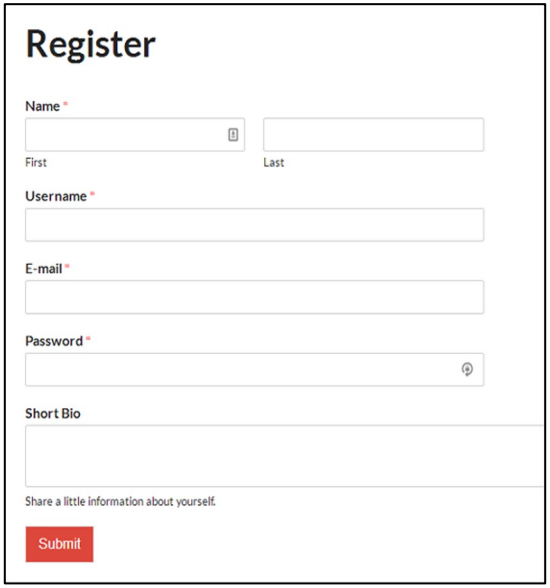
Identify Choices

- What choices do we make when using a function?
  - **Anything we control that can change the outcome.**
- What are the ***inputs*** 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

Identify Choices

- What are the inputs to that feature?
  - `(first name, last name, date of birth, e-mail)`
- Website is part of product line with different database options.
  - `(database type)`
- Consider implicit environmental factors.
  - `(database connection, user already in database)`



Parameter Characteristics

Identify Choices

- ✧ 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

Identify Choices

- Input parameter split into multiple “choices” based on contextual use.
  - “Database” is an implicit input for User Registration, but it leads to **more than one** choice.
  - “Database Connection Status”, “User Record in Database”, “Percent of Database Filled” influence function outcome.
    - **The Database “input” results in three input choices when we design test cases.**

Examples

Identify an Independently Testable Function

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

Identify Choices

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

- Course number to add
- Student record
- What about a course database? Student record database?
- **What else influences the outcome?**

Example - Register for a Class

Identify Choices

- ✧ Student Record is an implicit input choice.
- ✧ How is it used?
  - Have you already taken the course?
  - Do you meet the prerequisites?
  - What university are you registered at?
  - Can you take classes at the university the course is offered at?

Example - Register for a Class

Identify Choices

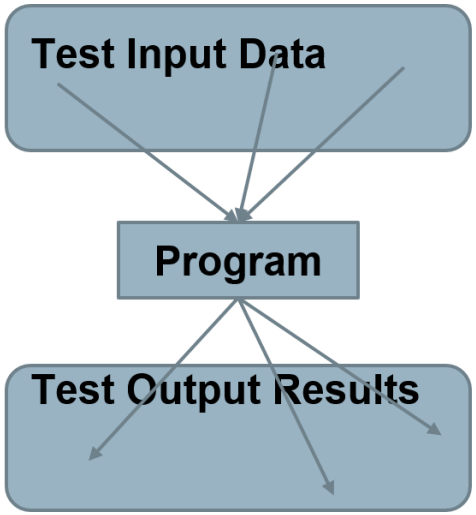
✧ Potential Test Choices:

- Course to Add
- Does course exist?
- Does student record exist?
- Has student taken the course?
- Which university is student registered at?
- Is course at a valid university for the student?
- Can student record be retrieved from database?
- Does the course exist?
- Does student meet the prerequisites?

Identifying Representative Values

Identify Representative Input Values

- We know the functions.
- We have a set of choices.
- What values should we try?
  - For some choices, finite set.
  - For many, near-infinite set.
- **What about exhaustively trying all options?**



Exhaustive Testing

Identify Representative Input Values

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.

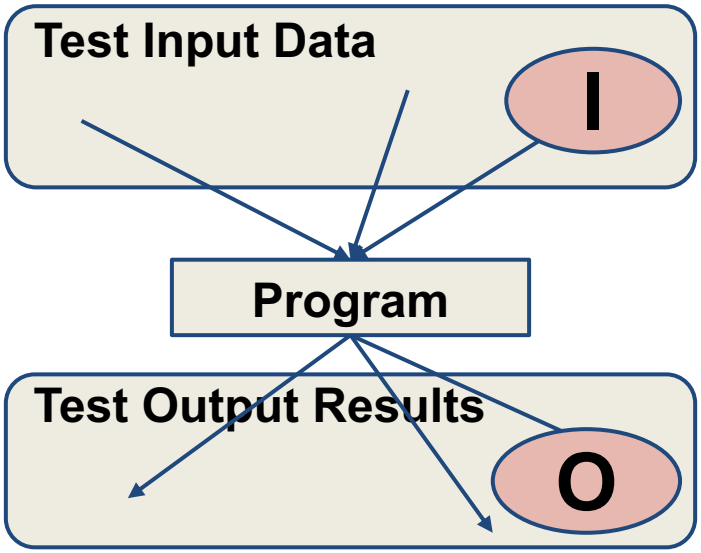
1 test per nanosecond
= 10^5 tests per second
= 10^{10} seconds

or… about 600 years!

Not all Inputs are Created Equal

Identify Representative Input Values

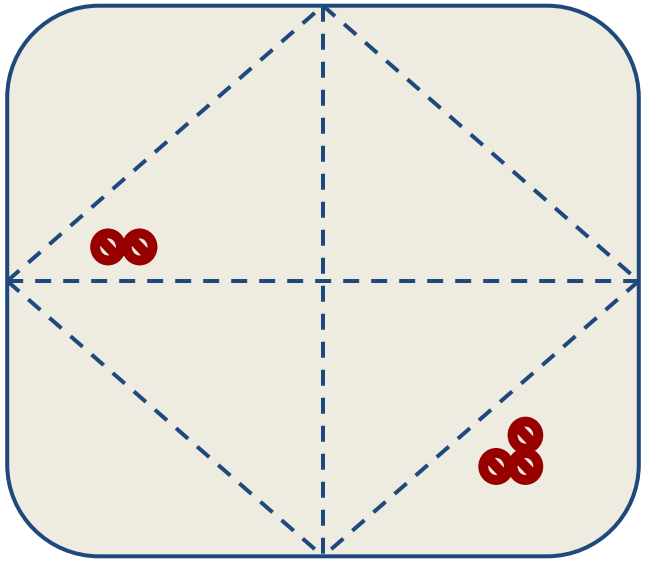
- ✧ Many inputs lead to same outcome.
- ✧ Some inputs better at revealing faults.
  - We can't know which in advance.
  - Tests with different input better than tests with similar input.



Input Partitioning

Identify Representative Input Values

- 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

Identify Representative Input Values

- 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.

Example

Identify Representative Input Values

```
substr(string str, int index)
```

What are some possible partitions?

- index < 0
- index = 0
- index > 0
- str with length < index
- str with length = index
- str with length > index
- ...

Choosing Input Partitions

Identify Representative Input Values

- ✧ Equivalent output events.
- ✧ Ranges of numbers or values.
- ✧ Membership in a logical group.
- ✧ Time-dependent equivalence classes.
- ✧ Equivalent operating environments.
- ✧ Data structures.
- ✧ Partition boundary conditions.

Look for Equivalent Outcomes

Identify Representative Input Values

- ✧ Look at the outcomes and group input by the outcomes they trigger.
- ✧ Example: **`getEmployeeStatus(employeeID)`**
  - Outcomes include: Manager, Developer, Marketer, Lawyer, Employee Does Not Exist, Malformed ID
  - Abstract values for choice `employeeID`.
    - Can potentially break down further.

Look for Ranges of Values

Identify Representative Input Values

- ✧ Divide based on data type and how variable used.
  - Ex: Integer input. Intended to be 5-digit:
    - < 10000, 10000-99999, >= 100000
    - Other options: < 0, 0, max int
    - Can you pass it something non-numeric? Null pointer?
- ✧ Try “expected” values and potential error cases.

Look for Membership in a Group

Identify Representative Input Values

Consider the following inputs to a program:

- ✧ A floor layout
- ✧ A country name.

✧ All can be partitioned into groups.

- Apartment vs Business, Europe vs Asia, etc.

✧ Many groups can be subdivided further.

✧ Look for context that an input is used in.

Timing Partitions

Identify Representative Input Values

- ✧ Timing and duration of an input may be as important as the 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.

Operating Environments

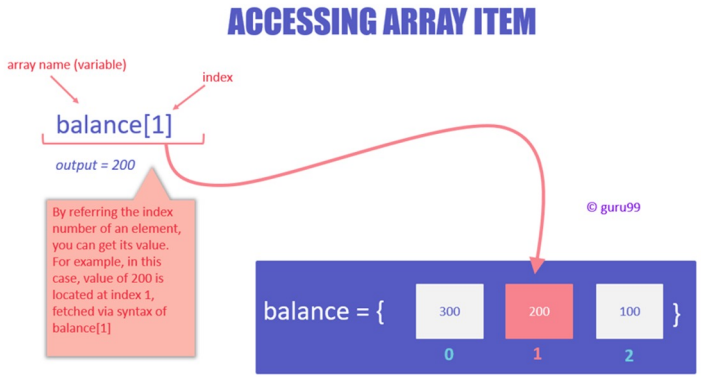
Identify Representative Input Values

- ✧ Environment may affect behavior of the program.
- ✧ Environmental factors can be partitioned.
  - 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)

Data Structures

Identify Representative Input Values

- ✧ Data structures are 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.



Input Partition Example

Identify Representative Input Values

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`

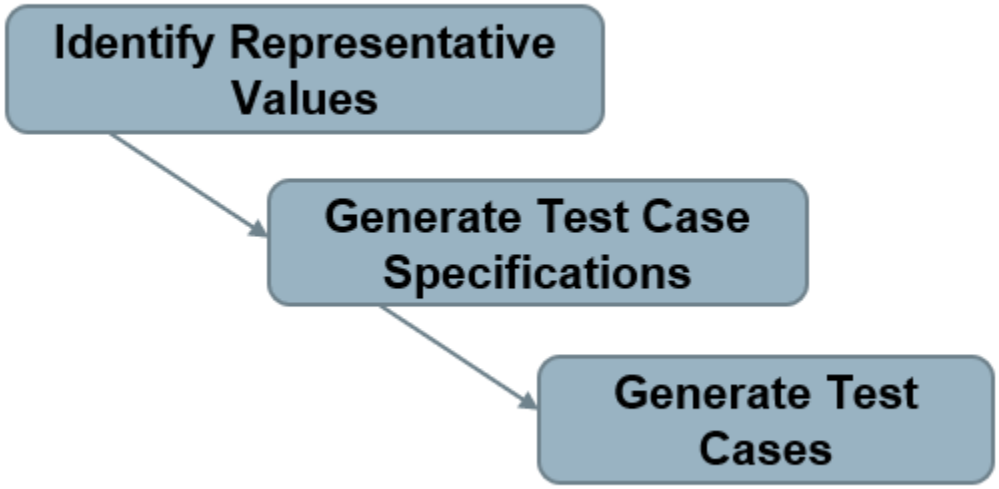
Consider combinations of `a` and `b` that change outcome:

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

Revisit the Roadmap

For each testing choice for a function, we want to:

1. Partition each choice into representative values.
2. Choose a value for each choice to form a test specification.
3. Assigning concrete values from each partition.



Forming Specification

Generate Test Case Specifications

Function `insertPostalCode(int N, list A)`.

- ✧ **Choice:** int N
  - 5-digit integer between 10000 and 99999
  - **Representative Values:** <10000, 10000-99999, >100000
- ✧ **Choice:** list A
  - list of length 1-10
  - **Representative Values:** Empty List, List of Length 1, List Length 2-10, List of Length > 10

Forming Specifications

Generate Test Case Specifications

Choose concrete values for each combination of input partitions: `insertPostalCode(int N, list A)`

`int N`

< 10000
10000 - 99999
> 99999

`list A`

Empty List
List[1]
List[2-10]
List[>10]

(3 * 4 = 12 abstract specifications)

Test Specifications:

```
insert(< 10000, Empty List)
insert(10000 - 99999, list[1])
insert(> 99999, list[2-10])
...
```

Concrete Test Cases:

(Each specification = 1000s of potential test cases)

```
insert(5000, {})
insert(96521, {11123})
insert(150000, {11123, 98765})
...
```

Generate Test Cases


```
substr(string str, int index)
```

Specification:

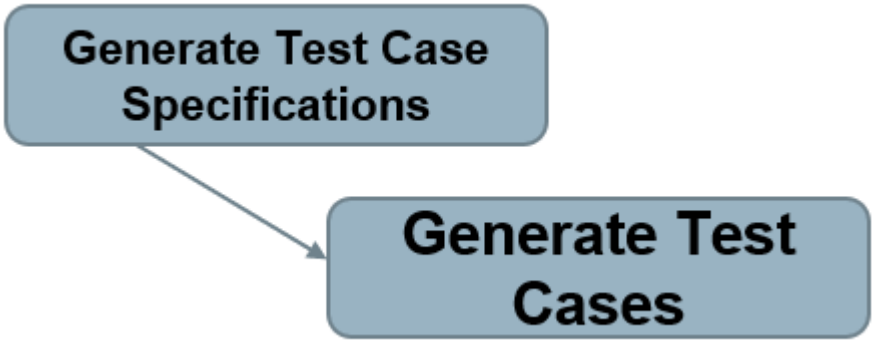
`str`: length >=2, contains special characters

`index`: value > 0

Test Case:

`str` = “ABCC!\n\t7”

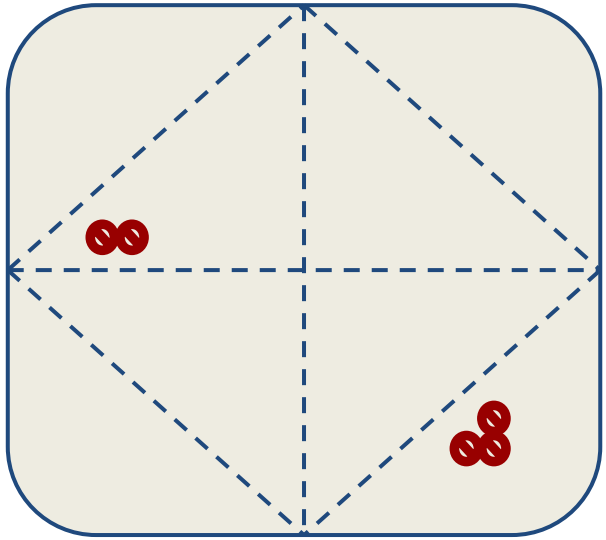
`index`= 5



Boundary Values

Generate Test Cases

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



Boundary Values

Generate Test Cases

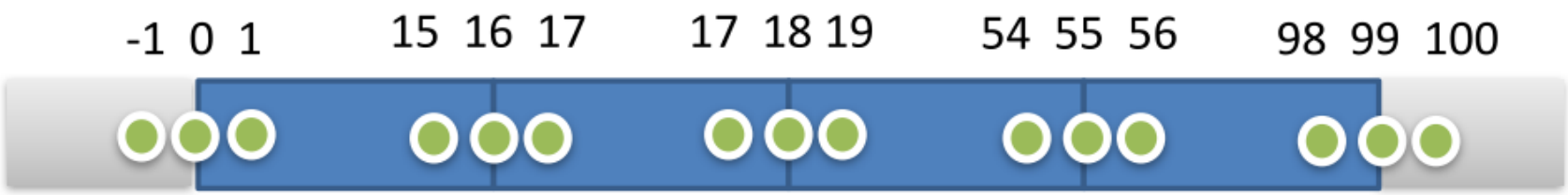
- Boundary value testing focuses on the boundaries between equivalence classes, simply because that is where so many defects hide. The defects can be in the requirements or in the code.

- Method
  - Identify the equivalence classes.
  - Identify the boundaries of each equivalence class.
  - Create test cases for each boundary value by choosing one point on the boundary, one point just below the boundary, and one point just above the boundary.

Boundary Values

Generate Test Cases

Age	Employment status
0-15	Don’t hire
16-17	Can hire part-time
18-54	Can hire full-time
55-99	Don’t hire



Example - `Set` Microservice

- Microservice related to Sets:
  - `void insert(Set set, Object obj)`
  - `Boolean find(Set set, Object obj)`
  - `void delete(Set set, Object obj)`
- For each **function**, identify **choices**.
- For each choice, identify **representative values**.
- Create **test specifications** with expected outcomes.

Example - `Set` Microservice

```
void insert(Set set, Object obj)
```

Identify an Independently Testable Function

✧ What are our choices?

Identify Choices

- **Parameter:** set
  - **Choice 1:** Number of items in the set
- **Parameter:** obj
  - **Choice 2:** Is obj already in the set?
  - **Choice 3:** Type of obj (e.g., valid, invalid, null)

Example - Set Microservice

```
void insert(Set set, Object obj)
```

Identify Representative Input Values

Parameter: set

- **Choice:** Number of items in the set
  - **Representative Values:**
    - Empty Set
    - Set with 1 item
    - Set with 10 items
    - Set with 10000 items

Parameter: obj

- **Choice:** Is obj already in the set?
  - **Representative Values:**
    - obj already in set
    - obj not in set
- **Choice:** Type of obj
  - **Representative Values:**
    - Valid obj
    - Null obj

Example - `Set` Microservice

Generate Test Case Specifications

Set Size	Obj in Set	Obj Status	Outcome
Empty	No	Valid	Obj added to Set
Empty	No	Null	Error or no change
1 item	Yes	Valid	Error or no change
1 item	No	Valid	Obj added to Set
1 item	No	Null	Error or no Change
10 items	Yes	Valid	Error or no change
10 items	No	Valid	Obj added to Set
10 items	No	Null	Error or no Change
10000	Yes	Valid	Error or no change (may be slowdown)
10000	No	Valid	Obj added to Set(may be slowdown)
10000	No	Null	Error or no Change (may be slowdown)

```
void insert(Set set,
Object obj)
```

- (4 * 2 * 2) = 16 specifications
  - Some are invalid (null in set). Can remove/ignore those.
- Each can become 1+ test cases.

Generate Test Cases

- (1 item, Yes, Valid) becomes:
- `insert({“Bob”}, “Bob”);`

Internal Interaction

- **Low-level functions are expected to interact.**
  - Usually this is planned!
  - Sometimes unplanned interactions break the system.
  - **We want to select tests that thoroughly test interactions.**

Triggering Interactions

- **Interactions** result from combining **values** of individual **choices**.
  - Inadvertent interactions cause unexpected behavior
  - (ex. incorrect output, timing)
- Want to detect, manage, and resolve inadvertent interactions.

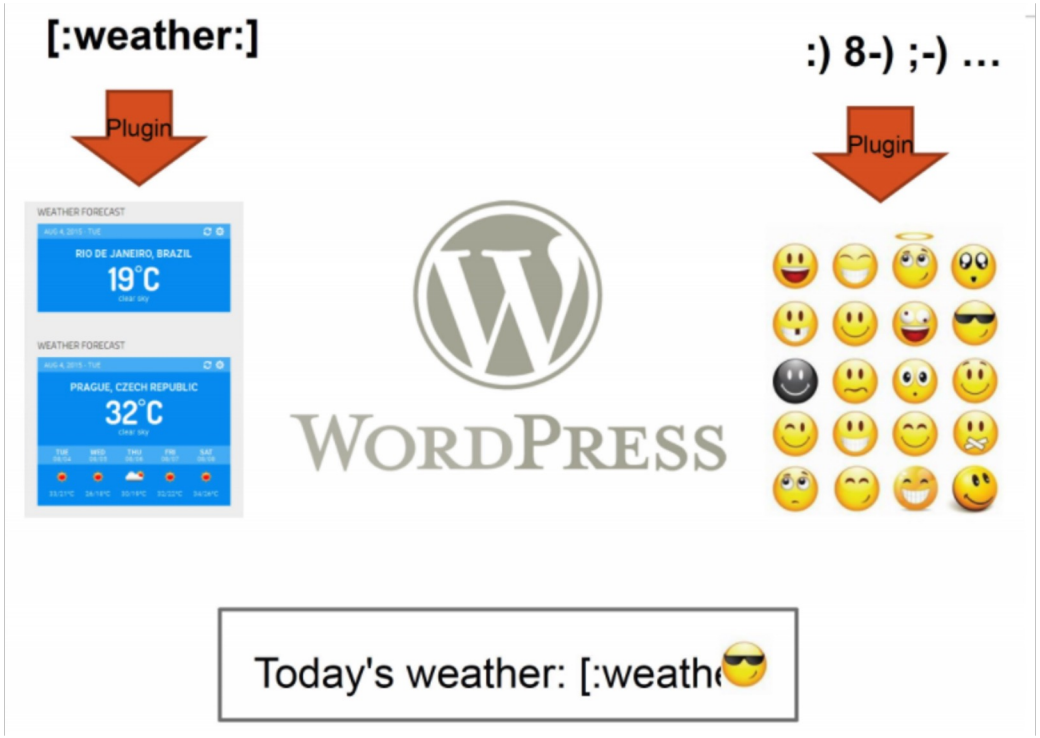
Fire and Flood Control

- FireControl activates sprinklers when fire detected.
- FloodControl cuts water supply when water detected on floor.
- **Interaction means building burns down.**

WordPress Plug-Ins

✧ Weather and emoji plug-ins tested independently.

✧ Their interaction results in unexpected behavior.



Feature Interactions

Selecting Test Specifications

- ✧ We want to select *interesting* specifications.
- ✧ **Category-Partition Method**
  - Apply constraints to reduce the number of specifications.
- ✧ **Combinatorial Interaction Testing**
  - Identify a subset that covers all interactions between pairs of choices.

Category-Partition Method

Creates a set of test specifications.

✧ **Choices**, **representative values**, and **constraints**.

- **Choices:** What you can control when testing.
- **Representative Values:** Logical options for each choice.
- **Constraints:** Limit certain combinations of values.

✧ Apply more constraints to further limit set.

Identify Choices

- ✧ Examine parameters of function.
  - *Direct input*, *environmental parameters (i.e., databases)*, and *configuration options*.
- ✧ Identify characteristics of each parameter.
  - What aspects influence outcome? (**the choices**)
- ✧ **Choices** are also called ***categories*** if you look up category-partition method.

Example: Computer Configurations

- ✧ Web shop that sells custom computers.
- ✧ A *configuration* is a set of options for a *model*.
  - Some combinations are invalid (i.e., display port monitor with HDMI video output).
- ✧ Function: **`checkConfiguration(model,configuration)`**
  - What are the parameters?
  - What are the choices to be made for each parameter?

Example: Computer Configuration

- **Model:** Identifies a product and determines constraints on available components. Identified by a model number. Characterized by a set of slots. Slots may be required (must be filled) or optional (may be left empty).

- **Configuration:** Set of <slot, component> pairs. Must correspond to the required and optional slots of the model. Available components and a default for each slot are determined by the model. Slots may be empty (may be default for optional slots). Components can be compatible or incompatible with a model or with each other.

Example: Configuration Choices

- **Parameter: Model**
  - Model number
  - Number of required slots (must have a component)
  - Number of optional slots (component or empty)
- **Parameter: Configuration**
  - Selected configuration valid for model?
  - Number [required/optional] slots with non-empty selections.
  - Selected components for [required/optional] slots OK?
- **Parameter: Product Database**
  - Number of models in database
  - Number of components in database

Identify Representative Values

- Many values can be selected for each choice.
- Partition each choice into *types of values*.
  - Consider all outcomes of function.
  - Consider logical ranges or groupings.
- A test specification is a selection of values for all choices.
  - Concrete test case fills values for each abstract selection.

Values for Each Choice

Parameter: Model

- Choice: Model number
  - malformed
  - not in database
  - valid
- Choice: Number of required slots
  - 0
  - 1
  - many
- Choice: Number of optional slots
  - 0
  - 1
  - many

Parameter: Product Database

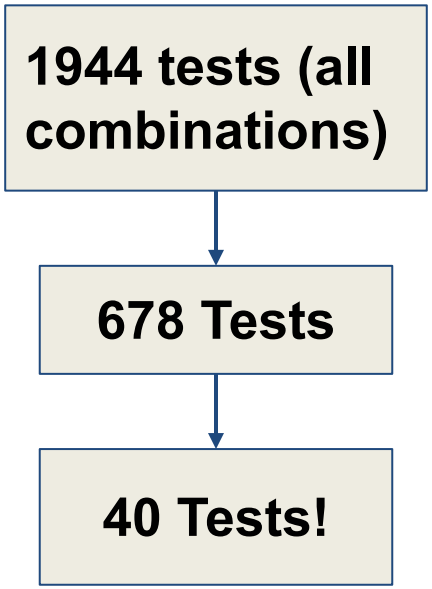
- Choice: Number of models in database
  - 0
  - 1
  - many
- Number of components in database
  - 0
  - 1
  - many

Parameter: Configuration

- Choice: Configuration Matches Model
  - complete correspondence
  - omitted slots in configuration
  - extra slots in configuration
  - mismatched number of required and optional slots
- Choice: Number of empty required slots that are empty
  - all required slots filled
  - some required slots empty
  - all required slots empty
- Choice: Number of optional slots that are empty
  - all optional slots filled
  - some optional slots empty
  - all optional slots empty
- Choice: Selected components for required slots
  - all valid
  - some kept at default
  - >= 1 incompatible with slot
  - >= 1 incompatible with another component
  - >= 1 not in database
- Choice: Selected components for optional slots
  - all valid
  - some kept at default
  - >= 1 incompatible with slot
  - >= 1 incompatible with another component
  - >= 1 not in database

Generate Test Case Specifications

- ✧ Test specification = selection of values for choices.
- ✧ **Constraints** limit number of specifications.
  - Eliminate impossible pairings.
  - Remove unnecessary options.
  - Choose a subset to turn into concrete tests.



Values for Each Choice

- Seven choices with three values, one with four values, two with five values.
  - $3^7 \times 5^2 \times 4 = 218700$ test specifications
- Not all combinations correspond to reasonable specifications.

Parameter: Model

- Choice: Model number
  - malformed
  - not in database
  - valid
- Choice: Number of required slots
  - 0
  - 1
  - many
- Choice: Number of optional slots
  - 0
  - 1
  - many

Parameter: Product Database

- Choice: Number of models in database
  - 0
  - 1
  - many
- Number of components in database
  - 0
  - 1
  - many

Parameter: Configuration

- Choice: Configuration Matches Model
  - complete correspondence
  - omitted slots in configuration
  - extra slots in configuration
  - mismatched number of required and optional slots
- Choice: Number of empty required slots that are empty
  - all required slots filled
  - some required slots empty
  - all required slots empty
- Choice: Number of optional slots that are empty
  - all optional slots filled
  - some optional slots empty
  - all optional slots empty
- Choice: Selected components for required slots
  - all valid
  - some kept at default
  - >= 1 incompatible with slot
  - >= 1 incompatible with another component
  - >= 1 not in database
- Choice: Selected components for optional slots
  - all valid
  - some kept at default
  - >= 1 incompatible with slot
  - >= 1 incompatible with another component
  - >= 1 not in database

Constraints Between Values

- ✧ IF-CONSTRAINT
  - This value only needs to be used under certain conditions (**if X is true, use value Y**)
- ✧ ERROR
  - Value causes error regardless of values of other choices.
- ✧ SINGLE
  - Only a single test with this value is needed.
  - Corner cases that should give “good” outcome.

Example - Substring

```
substr(string str, int index)
```

Choice: Str length

length = 0 — property zeroLen, TRUE if length = 0

length = 1 — if !zeroLen

length >= 2 — if !zeroLen

Choice: Str contents

contains letters and numbers — if zeroLen

contains special characters

empty

Choice: index

value < 0 — ERROR

value = 0

value = 1 — SINGLE

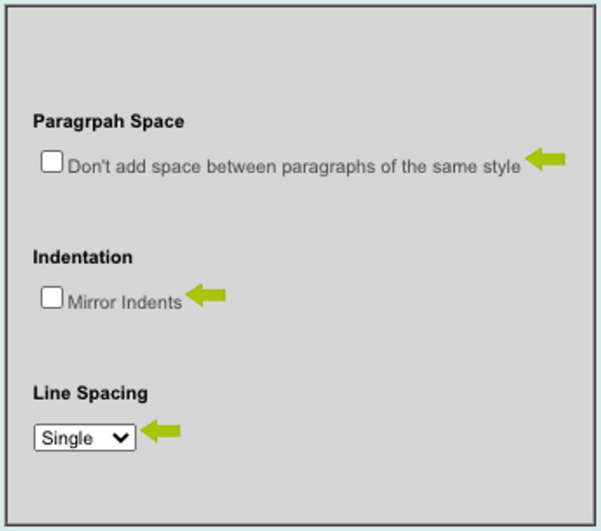
value > 1

Combinatorial Interaction Testing

- ✧ Cover all k-way interactions (k < N).
  - Typically **2-way (pairwise)** or 3-way.
- ✧ Set of all combinations grows exponentially.
- ✧ Set of pairwise combinations grows logarithmically.
  - (last slide) 432 combinations.
  - Possible to cover all pairs in 16 tests.

Example - Paragraph Effects

Paragraph spaces has two values: selected and unselected.
Mirror indents has two values: selected and unselected.
And finally, line spacing has three values: single, multiple and double.



Paragraph Space	Indentation	Line Spacing
Selected	Selected	Single
Unselected	Unselected	Double
		Multiple

2 * 2 * 3 = 12 combinations

Example - Paragraph Effects

Single	Indent Selected	Paragraph Selected
Single	Indent Unselected	Paragraph Selected
Single	Indent Selected	Paragraph Unselected
Single	Indent Unselected	Paragraph Unselected
Multiple	Indent Selected	Paragraph Selected
Multiple	Indent Unselected	Paragraph Selected
Multiple	Indent Selected	Paragraph Unselected
Multiple	Indent Unselected	Paragraph Unselected
Double	Indent Selected	Paragraph Selected
Double	Indent Unselected	Paragraph Selected
Double	Indent Selected	Paragraph Unselected
Double	Indent Unselected	Paragraph Unselected

Single	Indent Selected	Paragraph Selected
Single	Indent Unselected	Paragraph Selected
Single	Indent Selected	Paragraph Unselected
Single	Indent Unselected	Paragraph Unselected
Multiple	Indent Selected	Paragraph Selected
Multiple	Indent Unselected	Paragraph Selected
Multiple	Indent Selected	Paragraph Unselected
Multiple	Indent Unselected	Paragraph Unselected
Double	Indent Selected	Paragraph Selected
Double	Indent Unselected	Paragraph Selected
Double	Indent Selected	Paragraph Unselected
Double	Indent Unselected	Paragraph Unselected

Single	Indent Selected	Paragraph Selected
Single	Indent Unselected	Paragraph Selected
Single	Indent Selected	Paragraph Unselected
Single	Indent Unselected	Paragraph Unselected
Multiple	Indent Selected	Paragraph Selected
Multiple	Indent Unselected	Paragraph Selected
Multiple	Indent Selected	Paragraph Unselected
Multiple	Indent Unselected	Paragraph Unselected
Double	Indent Selected	Paragraph Selected
Double	Indent Unselected	Paragraph Selected
Double	Indent Selected	Paragraph Unselected
Double	Indent Unselected	Paragraph Unselected

Example - Paragraph Effects

✧ Goal of CIT is to produce **covering array**.

- Set of configurations that covers all K-way combinations.
  - (2-way here).
- Cover in 6 test specifications.

Single	Indent Selected	Paragraph Selected
Single	Indent Unselected	Paragraph Unselected
Multiple	Indent Selected	Paragraph Selected
Multiple	Indent Unselected	Paragraph Unselected
Double	Indent Selected	Paragraph Unselected
Double	Indent Unselected	Paragraph Selected

Constraints

- ✧ Remove all ERROR/SINGLE cases before CIT.
  - Error output, one-time corner cases
- ✧ Constraints on value combinations specified:
  - OMIT(Text-Only, *, *, Full Size, *)
  - OMIT(*, *, *, Full Size, Minimal)
- ✧ Further reduces number of test specifications.

CIT Tools

- ✧ Pairwise Independent Combinatorial Testing (Microsoft): https://github.com/microsoft/pict
- ✧ Automated Combinatorial Testing for Software (NIST): https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software
- ✧ .. Many more: http://www.pairwise.org/tools.asp

Key Points (1 of 2)

- ✧ Unit testing focus on a single class.
- ✧ System tests focus on high-level functionality, integrating low-level components through a UI/API.
  - Identify an independently testable function.
  - 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.

Key Points (2 of 2)

- ✧ Process for deriving system-level tests often results in **too many test specifications**.
- ✧ Two methods that **identify important interactions**:
  - **Category-Partition Method:** Use constraints to eliminate unnecessary tests.
  - **Combinatorial Interaction Testing:** Identify important pairs of input values.


HOW
WHY
WHERE
WHEN
WHERE
WHY
WHERE
WHO
WHEN
WHAT
WHAT
WHY
WHAT
WHEN
WHO
WHEN
HOW
WHO
WHERE
WHY
HOW