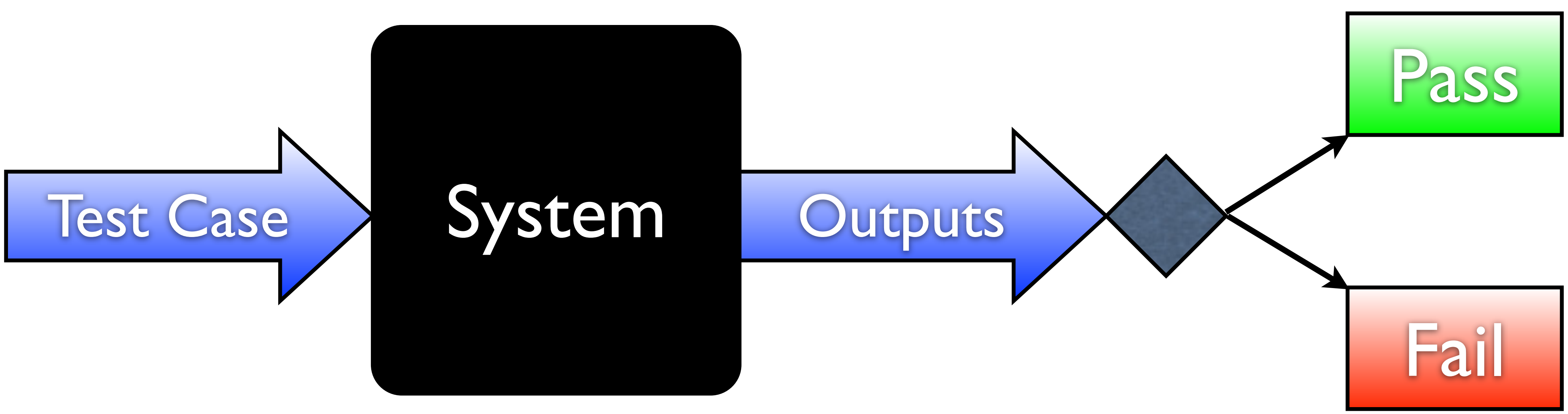
Testing Fundamentals 2

Black Box Testing

Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

–IEEE



Black Box Testing (2)

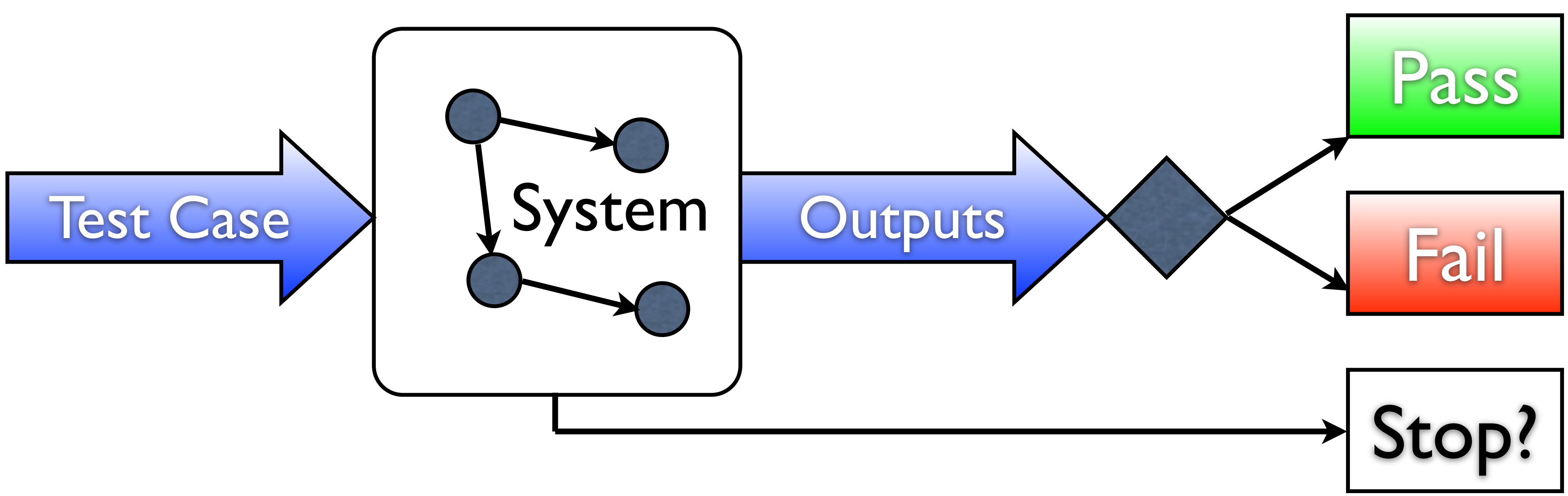
- Also known as Functional Testing
- Derive sets of inputs that will fully exercise all of the functional requirements of a system
- It is not an alternative to white-box testing

White Box Testing

Testing that takes into account the internal mechanism of a system or component

–IEEE

- Also known as Structural Testing, Glass Box Testing
- Gray Box Testing: Hybrid White/Black Box Testing



White Box Testing and Coverage

- White box testing is typically carried out with respect to some well-defined ***coverage criterion***
  - *Metric of completeness with respect to a test selection criterion –Boris Beizer*
  - *The degree to which a given test or set of tests addresses all specified properties of interest for a given system or component. –IEEE*
- Provides measurement of testing activity
- Influences test strategy

Why So Many Testing Strategies?

- The competent programmer hypothesis
  - Programmers tend to write code that is *mostly* correct
  - The more faults you expose, the harder it will be for you to find more
- Different strategies to
  - Achieve different testing goals
  - Capture different kinds of faults
    - (for different environments, languages, target software domains)

Basic Principles

Main Principles

- General engineering principles
  - Partition: divide and conquer
  - Visibility: making information accessible
  - Feedback: tuning the development process
- Specific Software Quality Principles
  - Sensitivity: better to fail every time than sometimes
  - Redundancy: making intentions explicit, allow for consistency checks
  - Restriction: making the problem easier

Sensitivity

- Cost of faults increases as time goes by
- Thus, the earlier that we can find a fault, the better
- Consider faults found during
  - Compilation
  - Unit tests
  - Integration testing
  - System testing
  - Deployment

Sensitivity (2)

- A fault that causes a failure on every execution will be found quickly
- A fault that rarely causes a failure will likely last much longer and be very expensive

Sensitivity (3)

- Sensitivity Principle: Should make faults easier to detect by making them cause failures more often.
- Can be done:
  - At the design level
  - At the analysis and testing level
  - At the environment level

Sensitivity (Design Level)

- Assertions/Checks for properties that you expect (null checks, bounds checking, etc.)
- Choice of programming constructs or library calls that do more checking (fast-fail iterators)

Sensitivity (Testing Level)

- Prefer techniques that are more apt to cause faults to manifest as failures
- Applying testing and analysis techniques that target specific vulnerable fault types
- Examples: Stress testing for code vulnerable to buffer overflows, concurrency analysis for code vulnerable to deadlocks and race conditions

Sensitivity (Environment Level)

- Any time there are outside influences that may affect sensitivity, try to control for them
- Example: code inspections may be affected by
  - Developer experience, mood, team interactions, time, etc.
  - Can create checklists that enforce that particular aspects be considered

Redundancy

- Opposite of independence
- Having multiple representations of intention
- One part of software artifact constrains the content of another
- Makes checking consistency possible

Examples of Redundancy

- A specification and the final program
- An algorithmic check for consistency
- Static type checking
- Programming language constructs (e.g., Java exception "throws")

Redundancy

- Challenge is to check for redundancy – preferably automatically
- Run-time checks, automatic source-code to specification checks, inspection checklists

Restriction

- Choosing suitable restrictions can reduce hard (or unsolvable) problems to simpler (or solvable) problems
- Examples:
  - It is impossible (in general) to show that pointers are used correctly, but the simple Java requirement that pointers are initialized before use is simple to enforce
  - It is impossible (in general) to show that type errors do not occur at run-time in a dynamically typed language, but statically typed languages impose stronger restrictions that are easily checkable.

Restriction

- Of course, restrictions limit developers. So, these should be chosen wisely.
- Trade-offs are always inherent in engineering decisions.

Partition

- Divide and conquer
- Divide complex activities into sets of simple activities that can be tackled independently
- Concept that is applied to all engineering disciplines, if not all of human endeavor

Partition (Process)

- We partition testing process into: unit, integration, system, ... testing
- We partition analysis into modeling and analyzing the model

Partition

- Difficult testing and verification problems can be handled by suitably partitioning the input space
- Partition the specification space for testing
- Partition the program structure for testing

Visibility (Process)

- The ability to measure progress or status against goals
- X visibility = ability to judge how we are doing on X (e.g., schedule visibility = “are we ahead of behind schedule”, quality visibility = “does quality meet our objectives?”
- Involves setting goals that can be assessed at each stage of development or testing

Visibility (Program)

- Related to observability – the ability to extract useful information from a software artifact
- Output to a human readable format
- Embedded debugging logging information

Feedback

- Learning from experience
- Applying lessons from experience in process and techniques
- Examples:
- Checklists are built on the basis of errors revealed in the past
- Error taxonomies can help in building better test selection criteria
- Design guidelines can avoid common pitfalls

Summary of Principles

The discipline of test/analysis can be characterized by six main principles:

- Sensitivity: better to fail every time than sometimes
- Redundancy: making intentions explicit; consistency
- Restriction: making the problem easier
- Partition: divide and conquer
- Visibility: making information accessible
- Feedback: tuning the development process

Survey Results

GitHub and JUnit Practicum