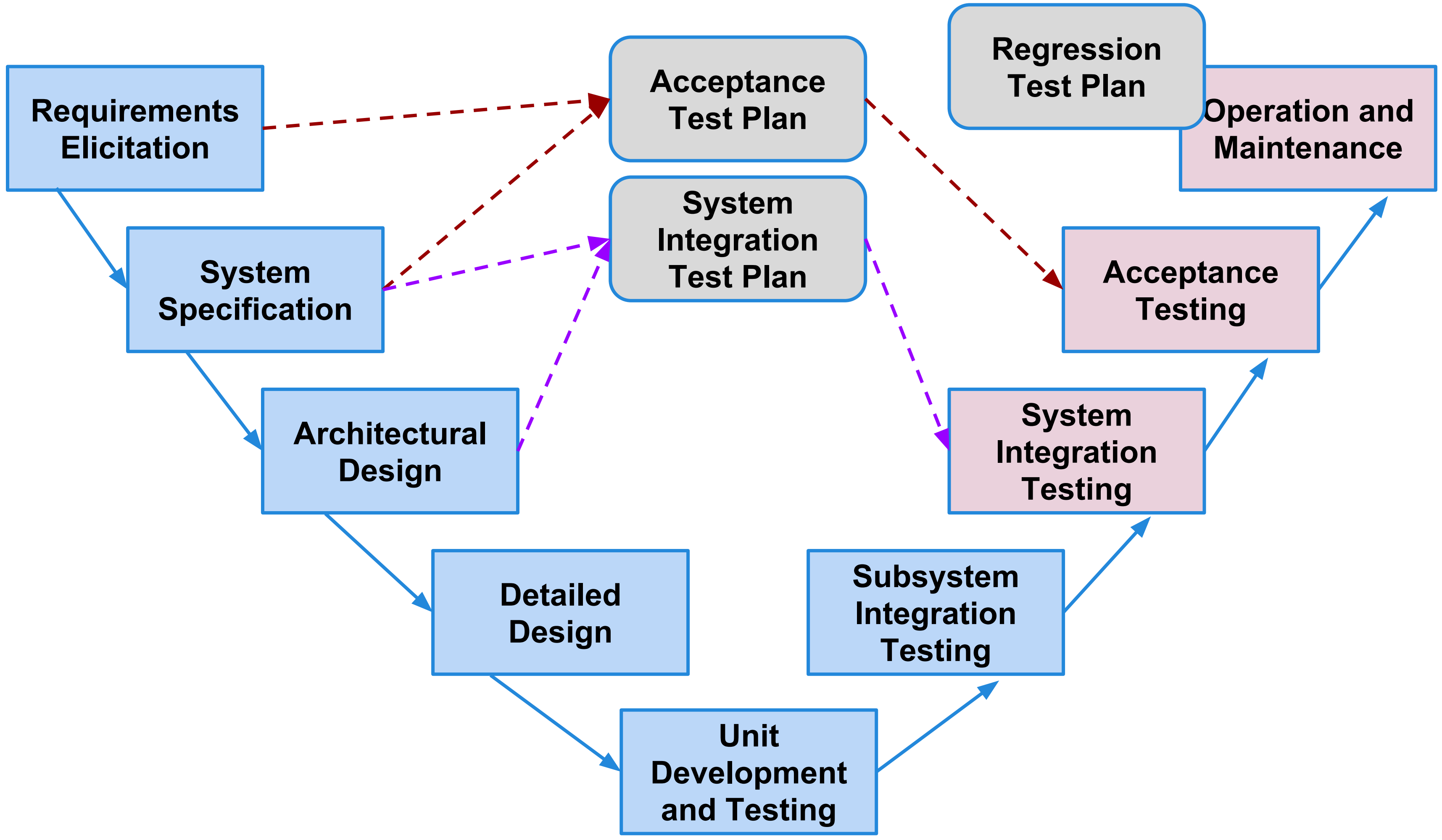
# Testing Close to and Post-Release:

## System, Acceptance, and Regression Testing

CSCE 747 - Lecture 23 - 04/05/2016

# The V-Model of Development



## “Final” Testing Stages

- All concerned with behavior of the system as a whole, but for different purposes.
- System Testing
  - Verification of the completed system against the specifications.
- Acceptance Testing
  - Validation against the user’s expectations.
- Regression Testing
  - Ensuring that the system continues to work as expected when it evolves.

## Verification and Validation

Activities that must be performed to consider the software “done.”

- **Verification:** The process of proving that the software conforms to its specified functional and non-functional requirements.
- **Validation:** The process of proving that the software meets the customer’s true requirements, needs, and expectations.

## System and Acceptance Testing

- System Testing
  - Checks system against specification.
  - Performed by developers and professional testers.
  - Verifies correctness and completion of the product.
- Acceptance Testing
  - Checks system against user needs.
  - Performed by customers, with developer supervision
  - Validates usefulness and satisfaction with the product.

## Regression Testing

- Systems continue to evolve post-release.
  - Patches to newly-discovered faults.
  - New features.
  - Adaptations to new hardware/software dependencies (OS).
- Rechecks test cases passed by previous production systems.
- Guards against unintended changes.

## System Testing

## Unit Testing

- Test cases derived from module specifications in design documents.
- Requires complex scaffolding to execute incomplete dependencies (stubs), simulate execution environment (drivers), and judge test results (oracles).
- Focus is on behavior of individual modules.

## Integration/Subsystem Testing

- Test cases derived from architecture and design specifications.
- Requires scaffolding, but can reuse some from unit testing. Fewer stubs and drivers required, as classes are tested together.
  - (depends on integration order and architecture)
- Focus is on module integration and interactions.

## System Testing

- Test cases derived from requirement specification.
    - Requires no detail of the code.
    - Tests should be designed before code is written.
- Does not need scaffolding, except for oracles. Sometimes operates in a simulated environment.
- Focus on system functionality and further integration errors.

## System Testing

- “Last hurdle” before releasing a system.
  - Final chance to find faults.
  - If all tests pass, the system is “free of faults.”
- Test cases should be developed independently of unit and subsystem tests.
  - Design errors often infect unit test design.
  - Introduces a source of blindness.
- System tests do not require details of the source code.
  - And are able to detect design flaws as a result.

# Who Should Test?

## Developer

- Understands the system, but…
- Tends to test gently and is driven by deadlines.

## Independent Tester

- Needs to learn the system, but…
- Will attempt to break it.
- Better able to focus on quality.

## System Testing

Systems are developed as interacting subsystems. Once units and subsystems are tested, the combined system must be tested.

- Advice about interface testing still important here (you interact with a system through some interface).
- Two important differences from subsystem testing:
  - Reusable components (off-the-shelf systems) need to be integrated with the newly-developed components.
  - Components developed by different team members or groups need to be integrated.

## Non-functional Properties

- Properties desired of the final system not related to functional correctness (f(a) = b) must be tested at the system level.
  - Performance, reliability, safety, security.
- Requires ability to measure and assess fulfillment of the property.
  - Depends on both the system and its environment and use.
  - In some cases, standard measurements exist.
  - In others, more difficult to capture the requirement in a test.

## Non-functional Properties

- US HIPAA regulations on software using medical records:
  - "A covered entity must reasonably safeguard protected health information from any intentional or unintentional use or disclosure that is in violation of the standards, implementation specifications, or other requirements of this subpart."
- Hard to measure satisfaction. Requires context:
  - Personnel that have access, how unauthorized personnel are prevented from gaining access.

## Non-functional Properties

- Properties must be defined in context.
  - Performance standards must consider the operating environment.
  - Under what circumstances must a particular threshold be met?
    - Real-time system must meet computation and response deadlines.
    - Requires definition of event frequency and minimum input arrival times.
- Generally cannot use testing to show that a property is met in all configurations.

## Non-functional Properties

- Not all properties are amenable to traditional testing techniques.
- Inspection and analysis can help with some.
- Security properties are assessed by teams of users that attempt to gain access.
  - Security is a property of a larger system and environment that one piece of software is a small part of.
  - Consider safety of the whole system, and how this piece of software fits into that environment. Look for vulnerabilities in each piece of software.

## Acceptance Testing

## Acceptance Testing

Once the system is internally tested, it should be placed in the hands of users for feedback.

- Users must ultimately approve the system.
- Many faults do not emerge until the system is used in the wild.
    - Alternative operating environments.
    - More eyes on the system.
    - Wide variety of usage types.
- Acceptance testing allows users to try the system under controlled conditions.

## User-Based Testing Types

Three types of user-based testing:

- Alpha Testing
  - A small group of users work closely with development team to test the software.
- Beta Testing
  - A release of the software is made available to a larger group of interested users.
- Acceptance Testing
  - Customers decide whether or not the system is ready to be released.

## Alpha Testing

- Users and developers work together.
  - Users can identify problems not apparent to the development team.
    - Developers work from requirements, users have their own expectations.
- Takes place under controlled conditions.
  - Software is usually incomplete or untested.
- “Power users” and customers who want early information about system features.
  - Agile processes advocate for “customer as a team member”

## Beta Testing

- Early build made available to a larger group of volunteers and customers.
- Software is used under uncontrolled conditions, hardware configurations.
  - Important if the system will be sold to any customer.
  - Discovers interaction problems.
- Can be a form of marketing.
- Should not replace traditional testing.

## Acceptance Testing

- Formal validation activity between developer and customer.
- Software is taken to a group of users that try a set of scenarios under supervision.
  - Scenarios mirror the typical system use cases.
  - Users provide feedback and decide whether the software is acceptable for each scenario.
- Users ultimately decide whether the software is ready for release.
  - Developers may negotiate with users.

## Acceptance Testing Stages

- Define acceptance criteria
  - Work with customers to define how validation will be conducted, and the conditions that will determine acceptance.
- Plan acceptance testing
  - Decide resources, time, and budget for acceptance testing. Establish a schedule. Define order that features should be tested. Define risks to testing process.
- Derive acceptance tests.
  - Design tests to check whether or not the system is acceptable. Test both functional and non-functional characteristics of the system.

## Acceptance Testing Stages

- Run acceptance tests
  - Users complete the set of tests. Should take place in the same environment that they will use the software. Some training may be required.
- Negotiate test results
  - It is unlikely that all of the tests will pass the first time. Developer and customer negotiate to decide if the system is good enough or if it needs more work.
- Reject or accept the system
  - Developers and customer must meet to decide whether the system is ready to be released.

## Qualitative Process

- Results may vary based on the user surveyed and environmental factors.
  - Software may need to be accepted regardless of users' preferences if deadline is strict.
  - May be used as an "excuse" to reject a project.
- Users should be "typical"
  - Usually interested volunteers.
  - How users interact with a beta may not match the real system.
  - May not catch faults that normal users will see.

## Usability

- A **usable** product is quickly learned, allows users to work efficiently, and can be used without frustration.
  - Must be evaluated through user-based testing.
  - Objective criteria:
    - Time and number of operations to perform tasks.
    - Frequency of user error.
  - Subjective criteria:
    - Satisfaction of users.
  - Can be evaluated throughout lifecycle.

## Usability Testing Steps

- Inspecting specifications:
  - Checklists based on prior experience.
- Testing early prototypes:
  - Bring in end users to:
    - Explore mental models (exploratory testing)
    - Evaluate alternatives (comparison testing)
    - Validate usability.
  - May involve mockup GUIs, not working software.
- System and Acceptance Testing:
  - Evaluate incremental builds, compare against competitors, check against compatibility guidelines.

## Exploratory Testing

- Explore the mental model of end users.
    - Early in design stage, ask users how they would like to interact with the system.
- Look for common answers from users.
    - If conflicts, try to combine elements of answers.
    - Larger sample sizes will yield better results.
    - Consider all groups of stakeholders.
        - Some stakeholders will have different usage patterns from others.

## Validation Testing

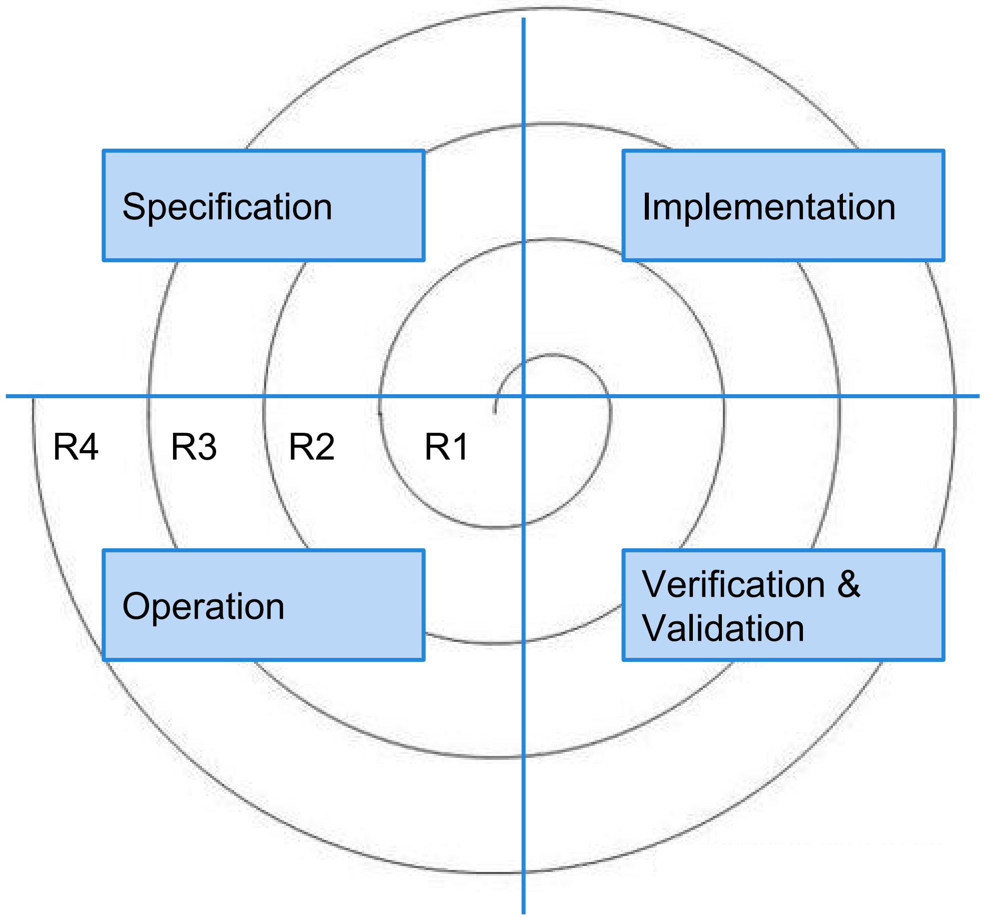
- Used to assess overall usability.
  - Identifies difficulties and obstacles encountered while using the system.
  - Measures error rate, clicks/time to perform a task.
- Preparation phase:
  - Define objectives for the session, identify items to be tested, select population, plan actions.
- Execution phase:
  - Users monitored as they execute planned actions.
- Analysis phase:
  - Results evaluated, software changes planned.

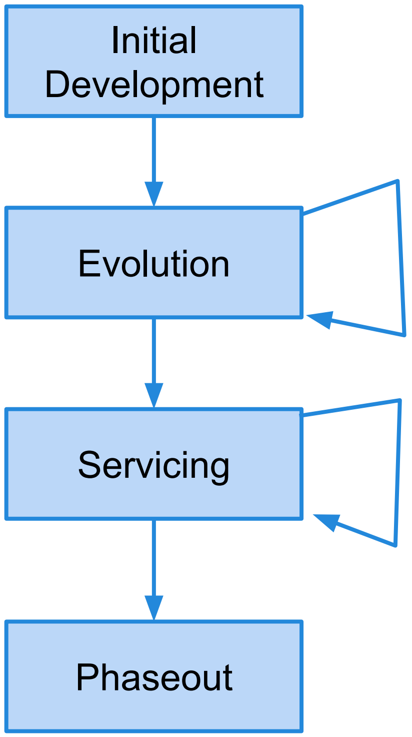
## Validation Testing

- Activities should be based on typical use cases of expected features.
  - Intent is to ensure “normal use” is optimal, not to search for new faults.
- Users should perform tasks independently.
  - Actions are recorded through tracking software.
  - Comments and impressions are collected with post-activity questionnaires.
- Consider accessibility needs.
  - Font size, color choices, audio guidance.

## Regression Testing

# Software Lifecycle





## Software Maintenance

- Fault Repairs
  - Changes made in order to correct coding, design, or requirements errors.
- Environmental Adaptations
  - Changes made to accommodate changes to the hardware, OS platform, or external systems.
- Functionality Addition
  - New features are added to the system to meet new user requirements.

## Maintenance is Hard

It is harder to maintain than to write new code.

- Must understand code written by another developer, or code that you wrote long ago.
- Creates a “house of cards” effect.
- Developers tend to prioritize new development.

New code must be tested. Existing code must also be *retested*.

## System Regression

- System evolution may change existing functionality in unforeseen ways.
- When a new version no longer works as expected, it *regresses* with respect to tested functionality.
  - A basic quality requirement is that new versions are *non-regressive* - if we tested it and it works, it should continue to work.
- Regression testing is used to detect regressive code.

## Regression Testing

- Basic idea: when changes have been made, re-execute tests that were used to verify the original code.
- Not as simple as it sounds:
  - When do you execute regression tests?
    - On check-in? Before patch is publicly released?
  - Can you afford to execute all tests?
    - The number will grow as the system expands.
  - Can you actually execute all tests?
    - Do you need to?
    - Are some tests obsolete?

## Test Case Maintenance

- Test suites must be maintained over time.
- Obsolete tests should be removed.
  - Tests involving requirements, features, classes, or interfaces that no longer exist or have been modified.
- Redundant tests should be identified.
  - Tests that cover the same structural elements, input partitions, other test goals.
  - May be introduced to test changed code, or by concurrently-working testers.
  - Can still be executed, but may not be needed.

## Regression Test Selection

- The number of tests to reexecute may be very large (and grows over time).
- Not all tests *need* to be re-executed.
  - Changes only affect part of the system.
  - Regression testing costs can be reduced by *prioritizing* the set of test cases.
  - Select a subset of tests relevant to the changes introduced. Weigh those tests higher than those unlikely to expose faults.
    - Techniques based on code and specifications.
  - Choose a cut-off based on testing budget.

## Code-Based Test Selection

- Select a test case for execution if it exercises a portion of the code modified (or likely to be affected by a change).
- Control-based selection:
    - Maintain a record of the CFG blocks executed by each test.
    - Compare the structure of the old and new versions.
    - Tests that exercise added, modified, or deleted elements are prioritized.
    - Can be based on control or data flow.

## Example

### Version 1:

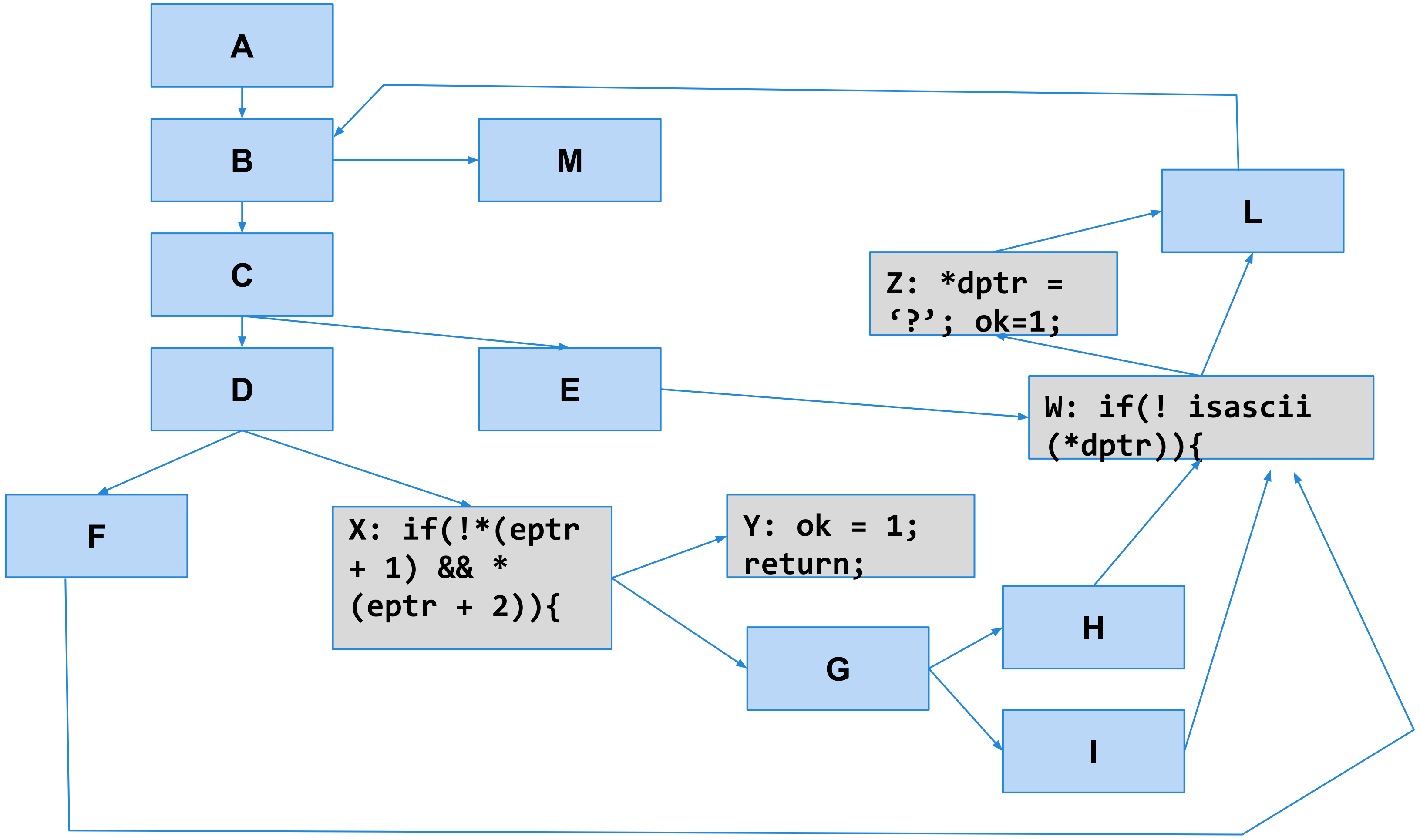
```
} else if (c == ‘%’){
    int digit_high = ..
}
 …

++dptr;
++eptr;
}
```

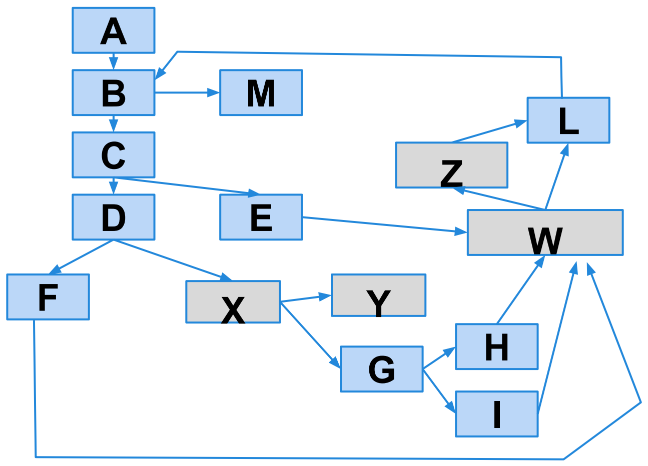
### Version 2:

```
} else if (c == ‘%’){
    if(!*(eptr + 1) && *(eptr + 2)){
        ok = 1; return;
    }
    int digit_high = ..
}
 …
if(! isascii(*dptr)){
        *dptr = ‘?’; ok=1;
}
++dptr;
++eptr;
}
```

# Example


A
B
M
L
C
Z: *dptr = '?'; ok=1;
D
E
W: if(! isascii (*dptr)){
F
X: if(!*(eptr + 1) && *(eptr + 2)){
Y: ok = 1; return;
H
G
I

# Example



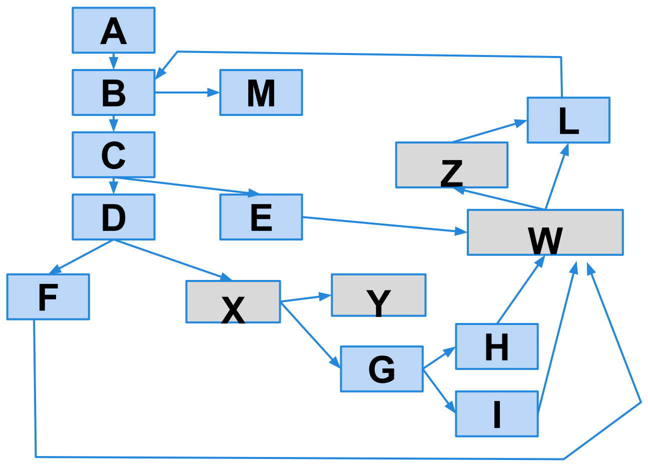
**Corrective Changes Only: Ignores new features, and only considers corrective patches.**

| ID | Input | Path |
|---|---|---|
| 1 | " " | A B M |
| 2 | "test+case%1Dadequacy" | A B C D F L … B M |
| 3 | "adequate+test% 0Dexecution%7U" | A B C D F L … B M |
| 4 | "%3D" | A B C D G H L B M |
| 5 | "%A" | A B C D G I L B M |
| 6 | "a+b" | A B C D F L B C E L B C D F L B M |
| 7 | "test" | A B C D F L B C D F L B C D F L B M |
| 8 | "+%0D+%4J" | A B C E L B C D G I L … B M |
| 9 | "first+test%9Ktest%K9" | A B C D F L … B M |

## Data-Based Test Selection

- New code can introduce new DU pairs and remove existing pairs.
- Re-execute test cases that execute DU pairs in the original program that were deleted or modified in the revised program.
    - Also select test cases that execute a conditional statements modified in the revision.
        - Changed predicates can affect DU paths.

# Example



| Variable | Definitions | Uses |
|---|---|---|
| *eptr | | **X** |
| eptr | | **X** |
| *dptr | **Z** | **W** |
| dptr | | **Z, W** |
| ok | **Y, Z** | |

## Selective Execution

- When a regression suite is too large, we must reduce the number of tests executed.
- Techniques predict “usefulness” of tests:
  - Elements covered.
  - History of effectiveness.
- High priority tests will be selected more often than low priority tests.
  - Eventually, all tests will be selected.
  - However, at varying frequencies.
  - Efficient rotation in which the cases most likely to reveal faults will be selected more often.

## Selective Execution Schema

- Execution History Schema:
  - Simple strategy.
  - Recently executed tests are given low priority.
  - Cases not recently executed are given high priority.
  - Often used to weight along with correlation to changed elements.
- Fault-Revealing Priority Schema:
  - Test cases that have recently revealed faults are prioritized.
  - Faults are not evenly distributed, but tend to cluster around particular functionality/units in the code.
  - Not all faults may have been fixed.

## Selective Execution Schema

- Structural Priority Schema:
  - Weight tests by the number of elements covered.
    - Statements, branches, conditions, etc.
  - Weight each element by when it was last executed.
  - Prioritize tests that cover a large number of elements that have not recently been executed.
  - Ensures that all structural elements are eventually recovered, especially if they have not recently been tested.

## We Have Learned

- Late-stage testing techniques are concerned with behavior of the system as a whole, but for different purposes.
- System Testing
    - Verification of the completed system against the specifications.
- Acceptance Testing
    - Validation against the user's expectations.
- Regression Testing
    - Ensuring that the system continues to work as expected when it evolves.

## Next Time

- When is software ready for release?
  - Measuring dependability
  - Some material in Chapter 4

- Homework:
  - Assignment 4 - due tonight!
  - Assignment 5 is out!
  - Presentations - April 19, 21, **26**, May 3