同济大学软件学院
School of Software Engineering, Tongji University

Software Testing Foundation Level

Chapter 2:Testing Throughout the Software Life Cycle

刘 琴 Qin Liu

Software Testing Foundation Level

- **1 Fundamental of testing**
- **2 Testing throughout the life cycle**
- **3 Static techniques** ***(Assignment 1)***
- **4 Test Design Techniques**
- [Extension]: Risk Based Testing Analysis and Design
- [Extension]: Usability Testing
- **5 Test management**
- **6 Tool support for testing**
- [Extension]: JUnit/Selenium ***(Assignment 2)***

Assignment 3

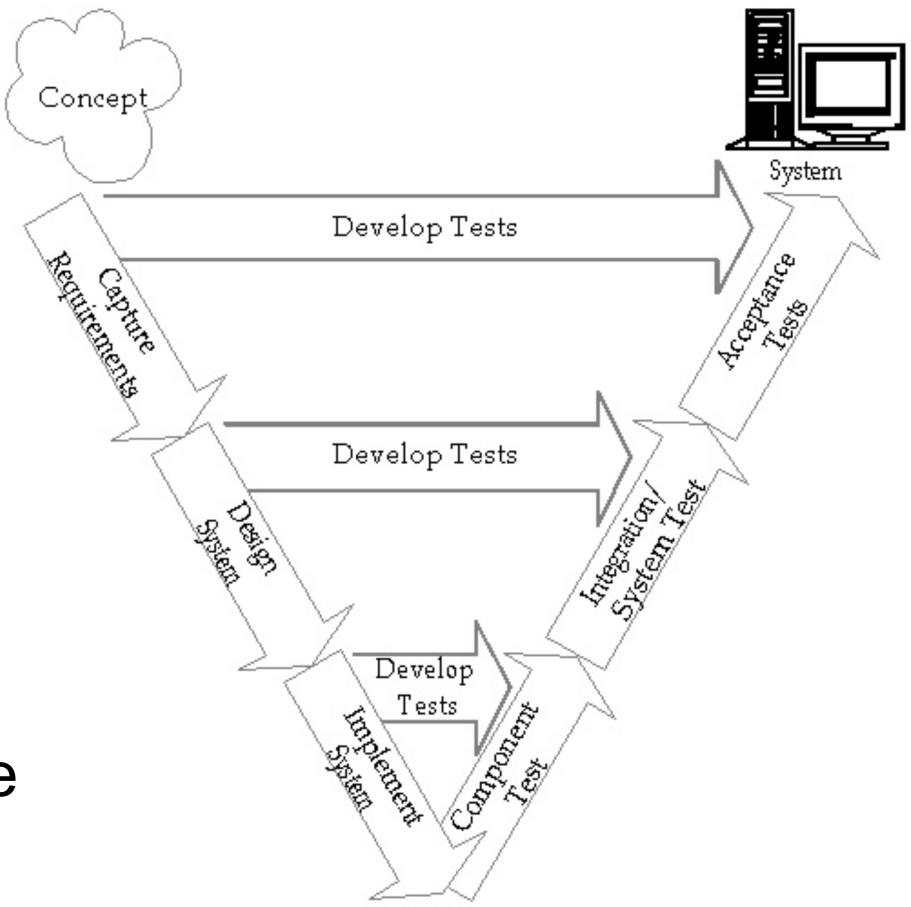
2 Testing throughout the life cycle

2.1.1 "V" or Sequential Model

- Usually schedule- and budget-risk-driven
- Do ever-deeper levels of design, then build, then test
- Intuitive and familiar model
- Beats chaos!

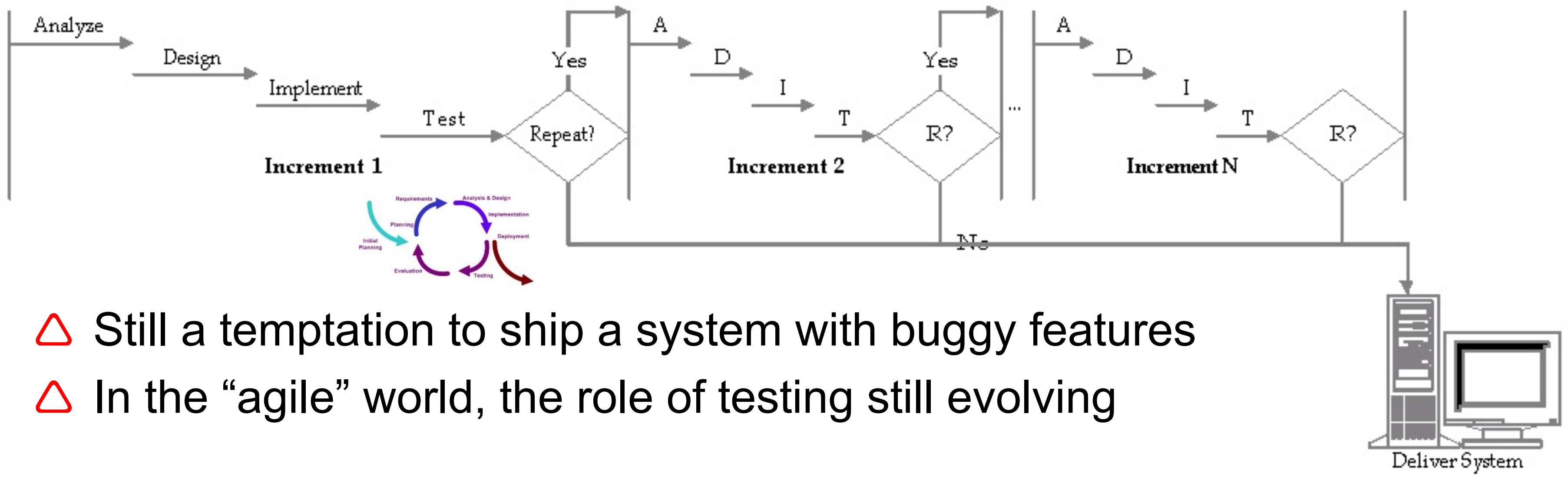
△ It's hard to plan that far in advance!

△ When plans fail, test--at the end--suffers!



2.1.2 Iterative, Evolutionary, or Incremental Model

- Schedule-risk-driven to hit market window or delivery date
- Feature set grown around core functionality
- Can ship (something) any time once the core functionality is ready
- Becoming a popular approach
- Ranges in formality from Extreme Programming to RAD and RUP

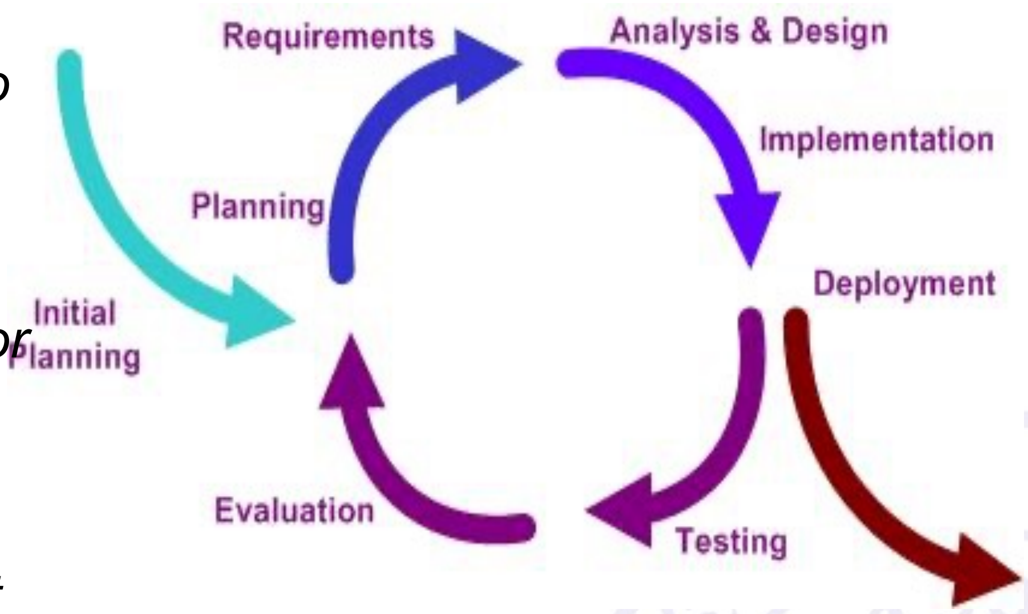


- △ Still a temptation to ship a system with buggy features
- △ In the “agile” world, the role of testing still evolving

2.1.3 General characteristics of good testing:

Regardless of Models

- *Conduct analysis, design and testing activities correspond to every <u>developing activity</u> at each test level*
- *Specify clear <u>test objectives</u> for each test level*
- *Tester should get involved <u>reviewing</u> as soon as the draft is available*



Exercise: Models and Reality

- Famed quality expert W.E. Deming said, “All models are wrong; some are useful.”
- Indicate which lifecycle model in this section applied most closely to your past project (or, if there was no organizing model, indicate “code-and-fix”).
- To what extent do you think the model was useful?
- To what extent, if any, was it harmful?

Exercise: Omninet Lifecycle Considerations

- Read the Omninet Marketing Requirements Document.
- Is a V-model (sequential) or incremental model more appropriate for this project? Why?
- Are maintenance, integration, or verification and validation important for this project? Why?
- Discuss.

2 Testing throughout the life cycle

2.2 Test levels or phases

1. Unit/Component Test:

individual pieces before fully integration

2. Integration/String Test:

relationships and interfaces between pairs and groups of components

3. System Test:

overall and particular behaviors, functions, and responses

4. Acceptance/Pilot Test:

demo, deployment/release

5. Maintenance Test:

during development of the changes

6. Operational Test:

in the operational environment (e.g., reliability or availability)

2.2.1 Component Test

- Depending on programming language, software units may be called by different names, e.g. modules, units, classes
- Generally, component testing is based on component requirements and component design. If white box applied, source code can be analyzed
- Test basis: Code, database, reqs/design, quality risks; Test types: Functionality, resource use, performance, structural

2.2.1 Component Test (cont.)

- Test object: program modules/units, classes, (database) scripts, etc.
- Main characteristic: components are tested individually, isolated from other components
  - Necessary to prevent external influences
- Component could be a unit composed of several other components
  - Remember to examine internal aspects only

2.2.1 Component Test (cont.)

- Test environment
- Test driver: case study
- Useful extensions include
  - Recording test data and results, including date/time
  - Reading test cases from table/file/database
- Writing test drivers requires programming skills and knowledge of the component under test
  - This is why developers usually perform component testing
- Use of component testing frameworks (e.g. xUnit) reduces effort in writing test drivers, and helps to standardize a project's component testing architecture
  - E.g. JUnit, nUnit, CppUnit

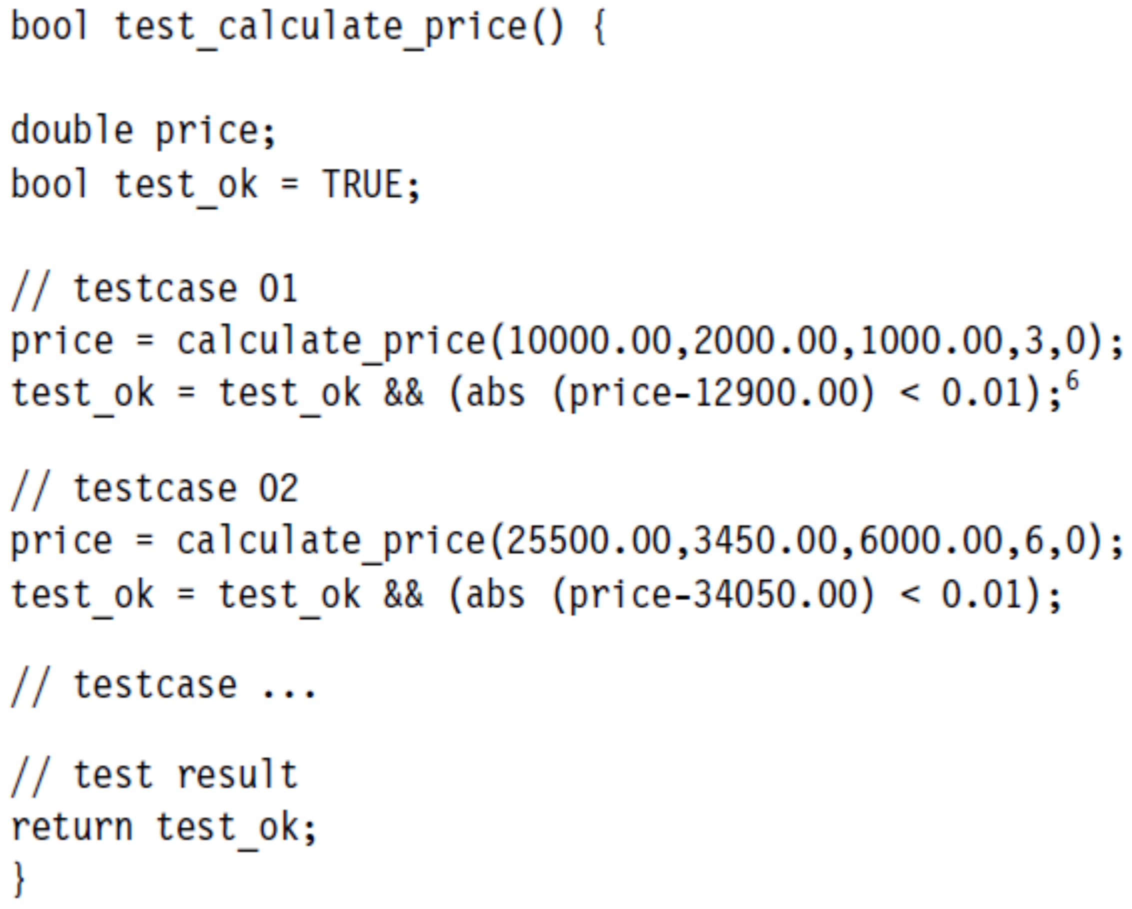
2.2.1 Component Test (cont.)

- Case study: calculating the price of the car

```
double calculate_price
   (double baseprice, double specialprice,
   double extraprice, int extras, double discount)
{
   double addon_discount;
   double result;

   if (extras >= 3) addon_discount = 10;
   else if (extras >= 5) addon_discount = 15;
   else addon_discount = 0;
   if (discount > addon_discount)
      addon_discount = discount;

   result = baseprice/100.0*(100-discount)
   + specialprice
   + extraprice/100.0*(100-addon_discount);
   return result;
}
```

2.2.1 Component Test (cont.)

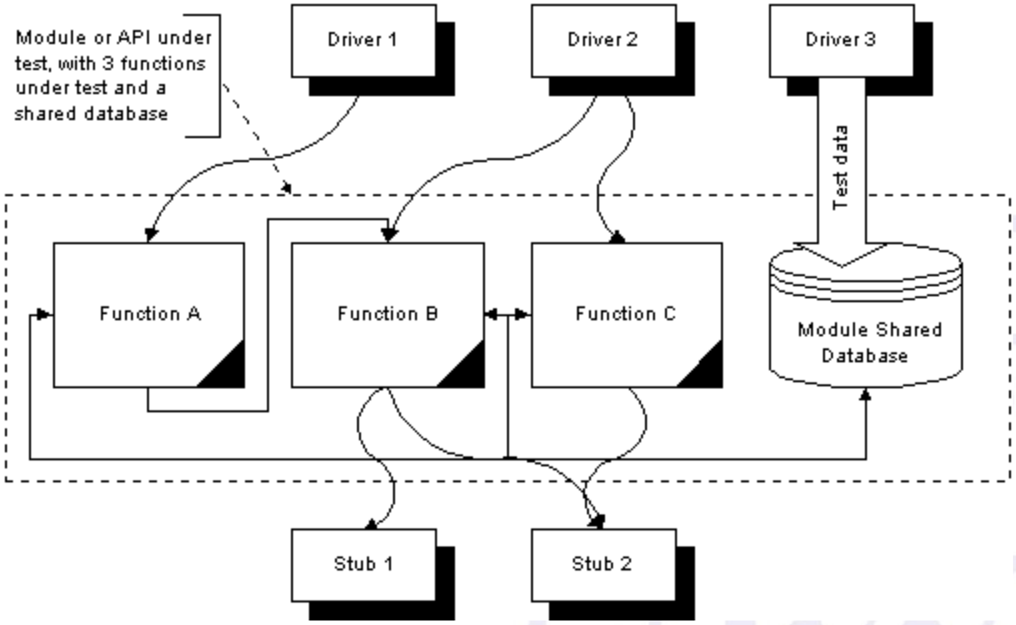
- For testing the price calculation



```
bool test_calculate_price() {

double price;
bool test_ok = TRUE;

// testcase 01
price = calculate_price(10000.00,2000.00,1000.00,3,0);
test_ok = test_ok && (abs (price-12900.00) < 0.01);[6]

// testcase 02
price = calculate_price(25500.00,3450.00,6000.00,6,0);
test_ok = test_ok && (abs (price-34050.00) < 0.01);

// testcase ...

// test result
return test_ok;
}
```

2.2.1.1 Drivers and Stubs

- During unit, component, and integration testing--and for testing APIs--it's often necessary to simulate parts of call flow reachable from module(s) under test
- Data setup sometimes necessary, also
- Driver: function(s) that call module(s) under test
- Stub: function(s) called--directly or indirectly--by module(s) under test



2.2.2 Integration Test

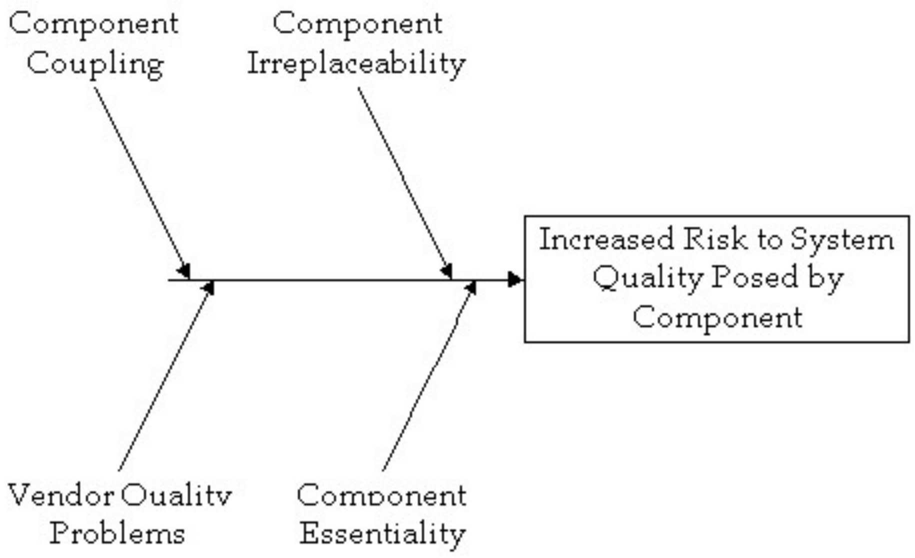
- Precondition: components have already been tested, and defects have been corrected
- Integration: developers, testers or integration teams compose components to form larger units and subsystems
- Integration test: make sure all components collaborate correctly
  - Goal: expose faults in interfaces and interaction between integrated components
- Test basis: software/system design, system architecture, workflows

2.2.2 Integration Techniques

- Big bang
  - Take all tested modules; put them all together; test
  - Quick, but where’s the bug?
  - Why wait until all code is written to start integration?
- Bottom up
  - Start with bottom layer modules; use appropriate drivers; test
  - Repeat process, replacing drivers with modules, until done
  - Good bug isolation, but what if nasty problems are at the top?
- Top down
  - Like bottom up, but start from top and use stubs
  - Good bug isolation, but what if nasty problems are at the bottom?
- Backbone
  - Start with critical modules; build initial backbone; use drivers and stubs; test
  - Repeat process, replacing stubs and drivers with modules in risk order
  - Good bug isolation and finds integration bugs in risk order

2.2.2 System Integration

- Many projects involve integrating components
- Risk-mitigation options
  - Integrate, track, and manage vendor testing in distributed test effort
  - Trust vendor component testing
  - Fix vendor testing/quality
  - Disregard and replace their testing (ouch!)
  - Watch politics in last two options
- Plan on integration and system testing yourself

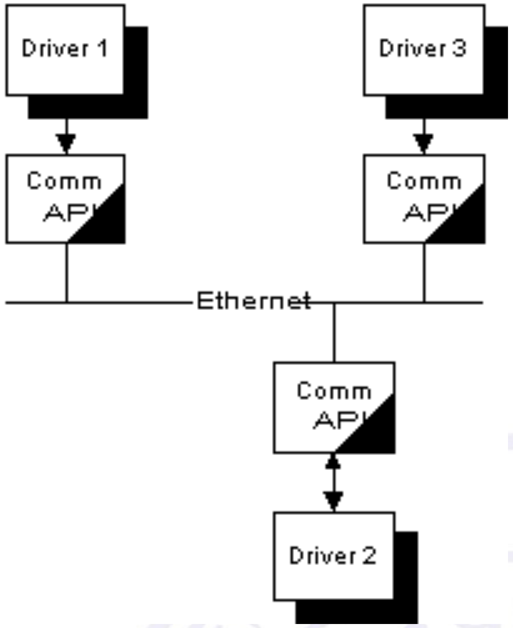

Component Coupling
Component Irreplaceability
Increased Risk to System Quality Posed by Component
Vendor Quality Problems
Component Essentiality

Coupling: Strong interaction or consequence of failure between component and system
Irreplaceability: Few similar components available
Essential: Key features in system unavailable if component does not work properly
Vendor quality problems: Increased likelihood of a bad component

2.2.2 Backbone Integration Technique

Backbone 0 (BB0)

- Consider the following example (based on a real project)
- We start with a basic backbone
  - Communication APIs
  - Basic networking architecture
- Test basic functionality, error handling and recovery, reliability, and performance
- Quality risk: Is the underlying system architecture untenable?

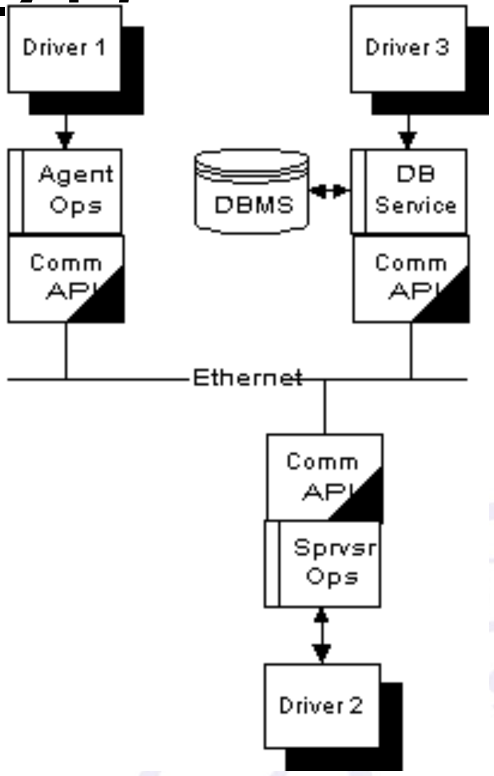


Backbone 0: Testing the basic communication software and network architecture

2.2.2 Backbone Integration Technique

Backbone 1 (BB1)

- Add the modules that implement the most critical core operations and services
- Again, test basic functionality, error handling and recovery, reliability, and performance
- Quality risk: Do the core operations and functions integrate with the transport layer?
- Continue process with next level of quality risk….

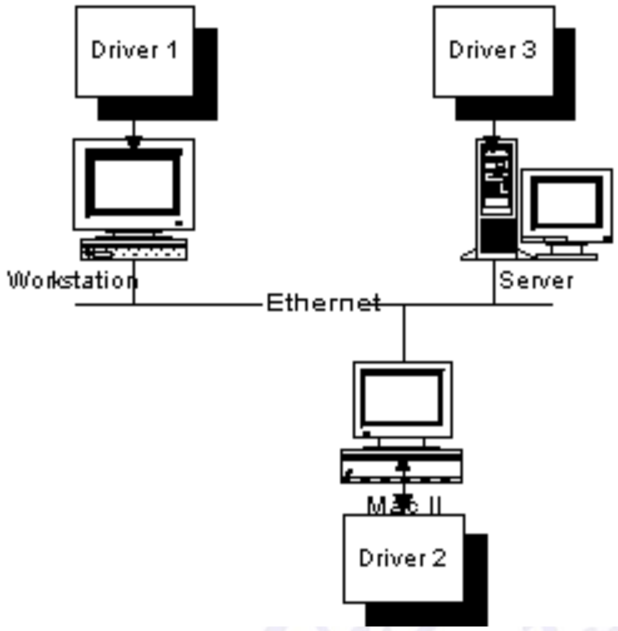


Backbone 1: Testing some core operations and services through the communication API and network

2.2.2 Backbone Integration Technique

Backbone N (BBn)

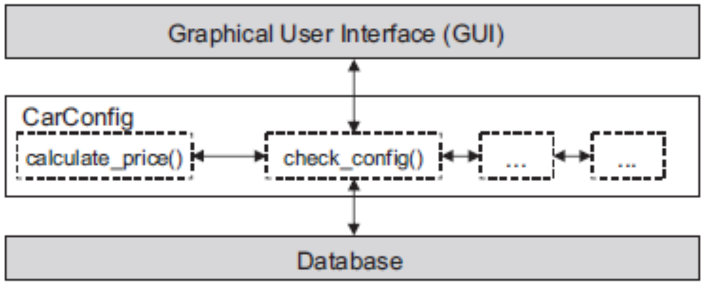
- As the last step, we are testing (in this case, using automated test drivers) end-to-end through the GUIs on the host systems
- Quality risk: Does the fully integrated system work?
- The final backbone for integration testing is also the first fully integrated build for system test

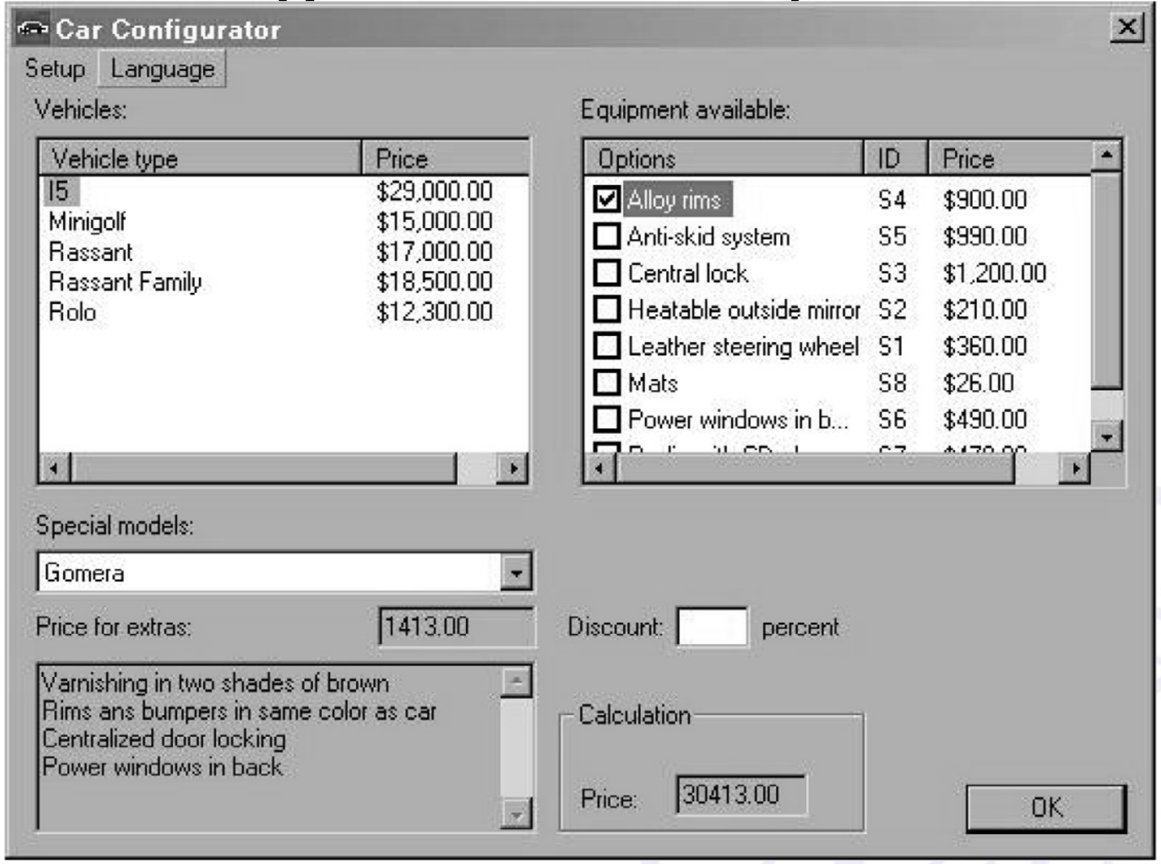


Backbone N: Testing the entire, completely integrated system end-to-end. When this works, we're ready for System Test

2.2.2 Integration Test (cont.)

- Why integration testing is necessary
- Example





Vehicles:

Vehicle type	Price
I5	$29,000.00
Minigolf	$15,000.00
Rassant	$17,000.00
Rassant Family	$18,500.00
Rolo	$12,300.00

Equipment available:

Options	ID	Price
☑ Alloy rims	S4	$900.00
☐ Anti-skid system	S5	$990.00
☐ Central lock	S3	$1,200.00
☐ Heatable outside mirror	S2	$210.00
☐ Leather steering wheel	S1	$360.00
☐ Mats	S8	$26.00
☐ Power windows in b...	S6	$490.00

Special models: Gomera

Price for extras: 1413.00

Discount: percent

Varnishing in two shades of brown
Rims ans bumpers in same color as car
Centralized door locking
Power windows in back

Calculation
Price: 30413.00

2.2.2 Integration Test Levels

- There can be more than one level of integration testing on a project
  - Component integration testing: testing interactions between units or components following unit/component test
  - System integration test: testing interactions between entire systems following system test
- System integration testing is complex
  - Multiple organizations controlling the systems' interfaces, making change dangerous
  - Business processes can span systems
  - Hardware/system compatibility issues can arise

2.2.3 System Test Levels

- Objective: Find bugs, build confidence, and reduce risk in the overall and particular behaviors, functions, and responses of the system under test as a whole
- Basis: Requirements, high-level design, use cases, quality risks, experience, checklists, environments
- Test types: Functionality, security, performance, reliability, usability, portability, etc.
- Item Under Test: Whole system, in as realistic-as-possible test environment
- Harnesses and tools: API, CLI, or GUI, freeware and commercial
- Responsible: Typically independent testers

2.2.4 Acceptance Test Levels

- Objective: Demonstrate that the product is ready for deployment/release
- Basis: Requirements, contracts, experience
- Test types: Functional, portability, performance
- Item Under Test: Whole system, sometimes in the production or customer environment
- Harnesses and tools: GUI usually
- Responsible: Often users or customers, but also independent testers

2.2.4 Variations in Acceptance Testing

- User acceptance testing: Business users verify fitness for functional purposes
- Operational testing: Acceptance by system administrators (e.g., backup-restore, disaster recovery, user management, maintenance, security)
- Contract and regulation testing: Verification of conformance to contractually-agreed or legally mandated requirements, regulations, or standards
- Alpha, Beta, and field testing: Testing and confidence-building by potential or existing customers, with beta testing and field testing are performed in the actual environment(s)

2.2.4 Acceptance Test

- All previous test levels are under the producer's responsibility
- Acceptance test: focusing on customer's and user's perspective
  - The only test involving customers/users
- Can be executed as a part of lower test levels
  - Product can be checked for acceptance during integration or installation
  - Usability of a component can be tested during component test
  - New functionality can be checked on prototypes

2.2.4 Acceptance Test (cont.)

- Typical forms of acceptance testing
  - Contract acceptance testing
  - User acceptance testing
  - Operational acceptance testing
  - Field testing (alpha and beta testing)
- How much acceptance testing?
- Test basis: any document from user/customer’s view point

2.2.4 Acceptance Test (cont.)

- Contract acceptance testing
  - Customer perform contract acceptance testing in cooperation with the vendor
  - Acceptance criteria determined in the development contract
  - Test cases designed/reviewed by customer
  - Running in customer’s actual operational environment

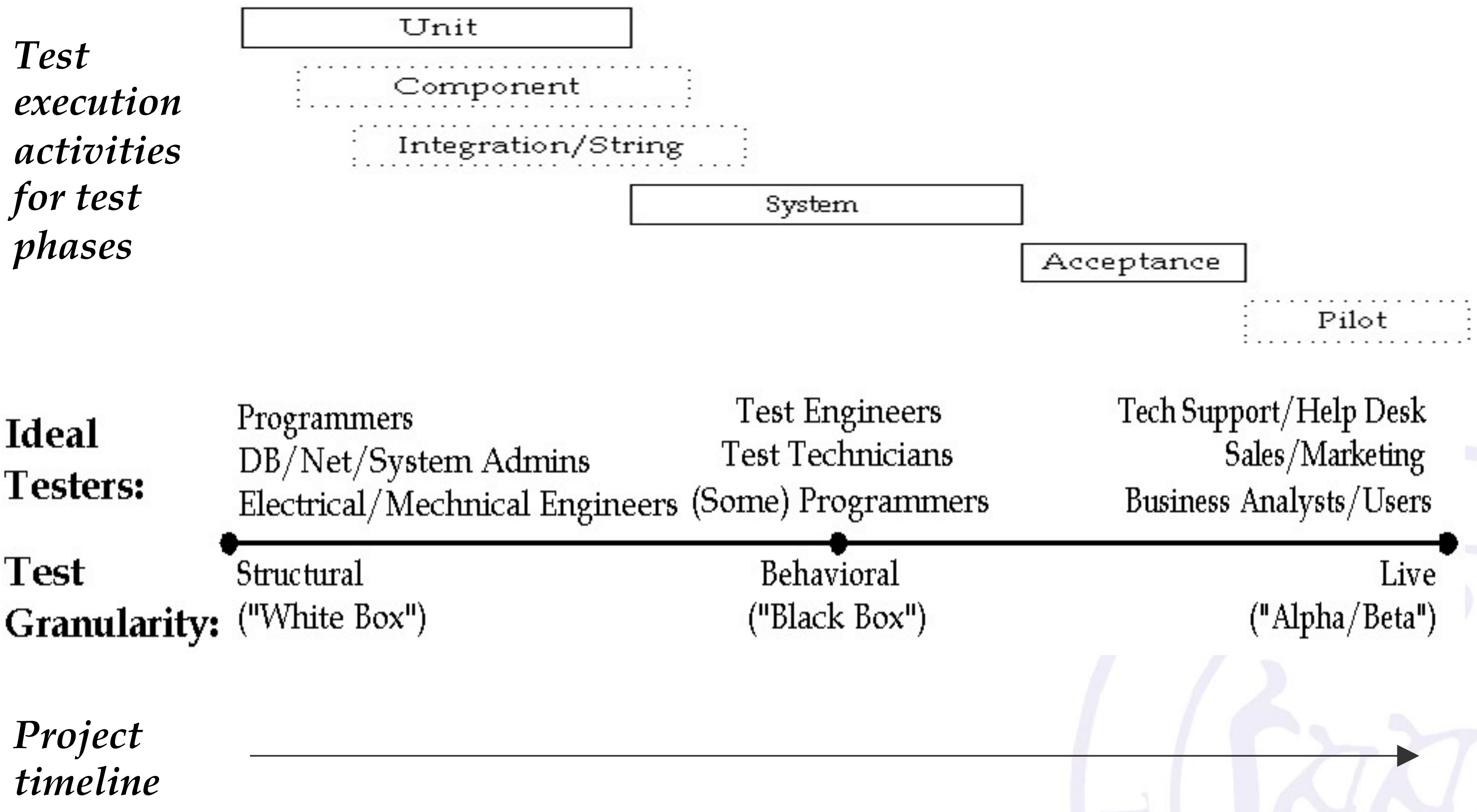
2.2.4 Acceptance Test (cont.)

- Testing for user acceptance
  - Recommended if the customer and the user are different (example: VSR project)
  - Different user groups may have different expectations
  - Present prototypes to users early
- Operational (acceptance) testing
  - Assuring the acceptance by system administrators
  - Backup/restore cycles, disaster recovery, user management, security, etc

2.2.4 Acceptance Test (cont.)

- Field testing
  - If running in many different operational environments, very expensive/impossible
  - Objective: identify influences from users' environments not entirely known/specified
  - Recommended if system is for general market
  - Stable prerelease versions to preselected customers (as representative customers)
  - Alpha and beta testing
  - Field test should not replace system test
  - Dogfood test

2.2 Pervasive Testing: Early, Cross-Functional

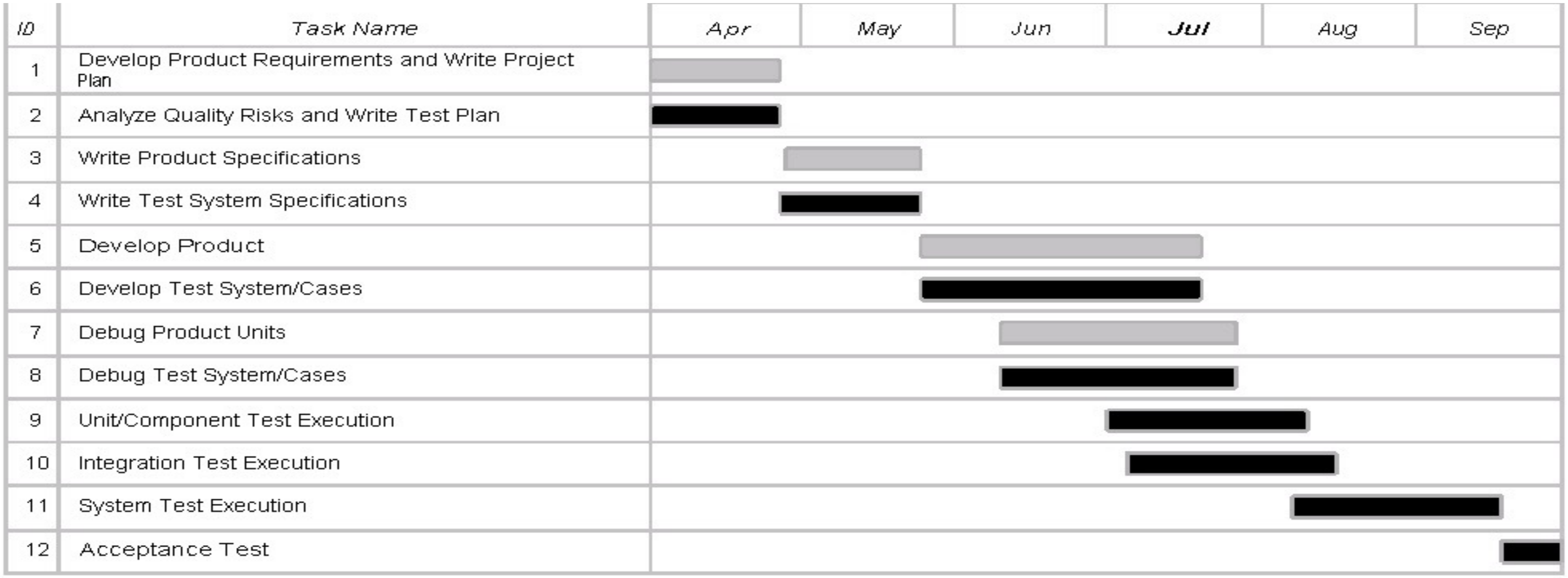


2.2 Why Pervasive Testing?

- Different participants can test different granularities
  - Different skills for different granularities
- Different granularities emphasized in each level (or phase)
  - Unit testing: primarily structural
  - System testing: primarily behavioral
  - Acceptance testing: primarily live
- It's important to be flexible, though
  - Test techniques of various granularities can be useful in all the test levels
  - Test level execution overlap depends on entry and exit criteria
  - Not all test levels occur on all projects

2.2 When Testing Pervades Projects

- Test tasks occur throughout the development effort
- Test level execution planned with multiple cycles to allow for fix time
- Features will be dropped or slipped into later iterations rather than slipping test level entry dates or entering before ready

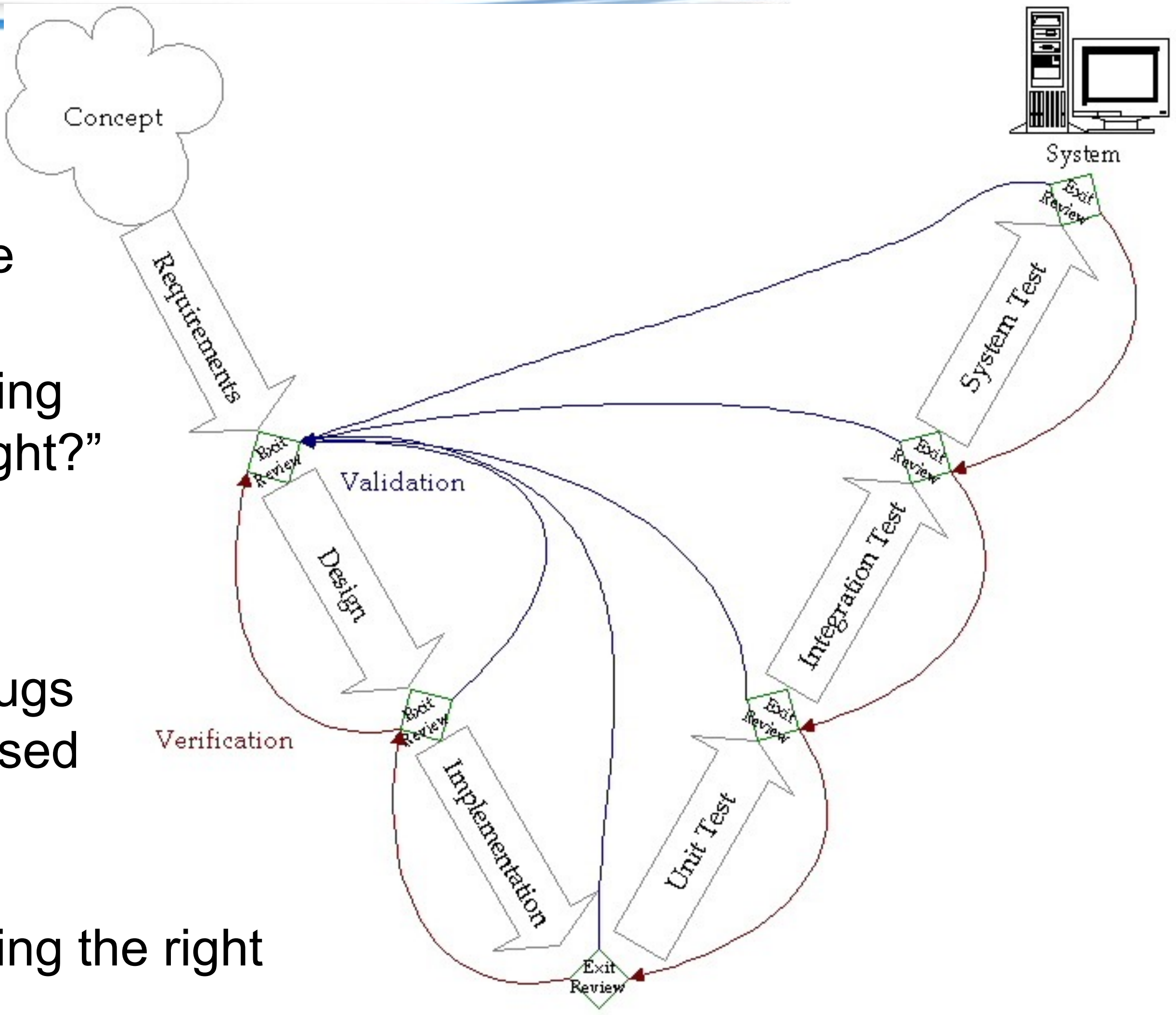


ID	Task Name
1	Develop Product Requirements and Write Project Plan
2	Analyze Quality Risks and Write Test Plan
3	Write Product Specifications
4	Write Test System Specifications
5	Develop Product
6	Develop Test System/Cases
7	Debug Product Units
8	Debug Test System/Cases
9	Unit/Component Test Execution
10	Integration Test Execution
11	System Test Execution
12	Acceptance Test

2.2 Verification & Validation

- Verification
  - Look for bugs in phase deliverables
  - “Are we building the system right?”

- Validation
  - Looking for bugs in system, based on phase deliverables
  - “Are we building the right system?”



Exercise: Omninet Test Levels (A2.1.3)

- Read the Omninet System Requirements Document.
- If you were managing all the testing for Omninet, which levels or phases of testing would you plan? Why?
- What would the major goals of each level or phase of testing be?
- What kind of acceptance test, if any, would you plan? Why?
- How do these test levels relate to and affect the lifecycle model you selected in the previous exercise?
- Discuss.

2 Testing throughout the life cycle

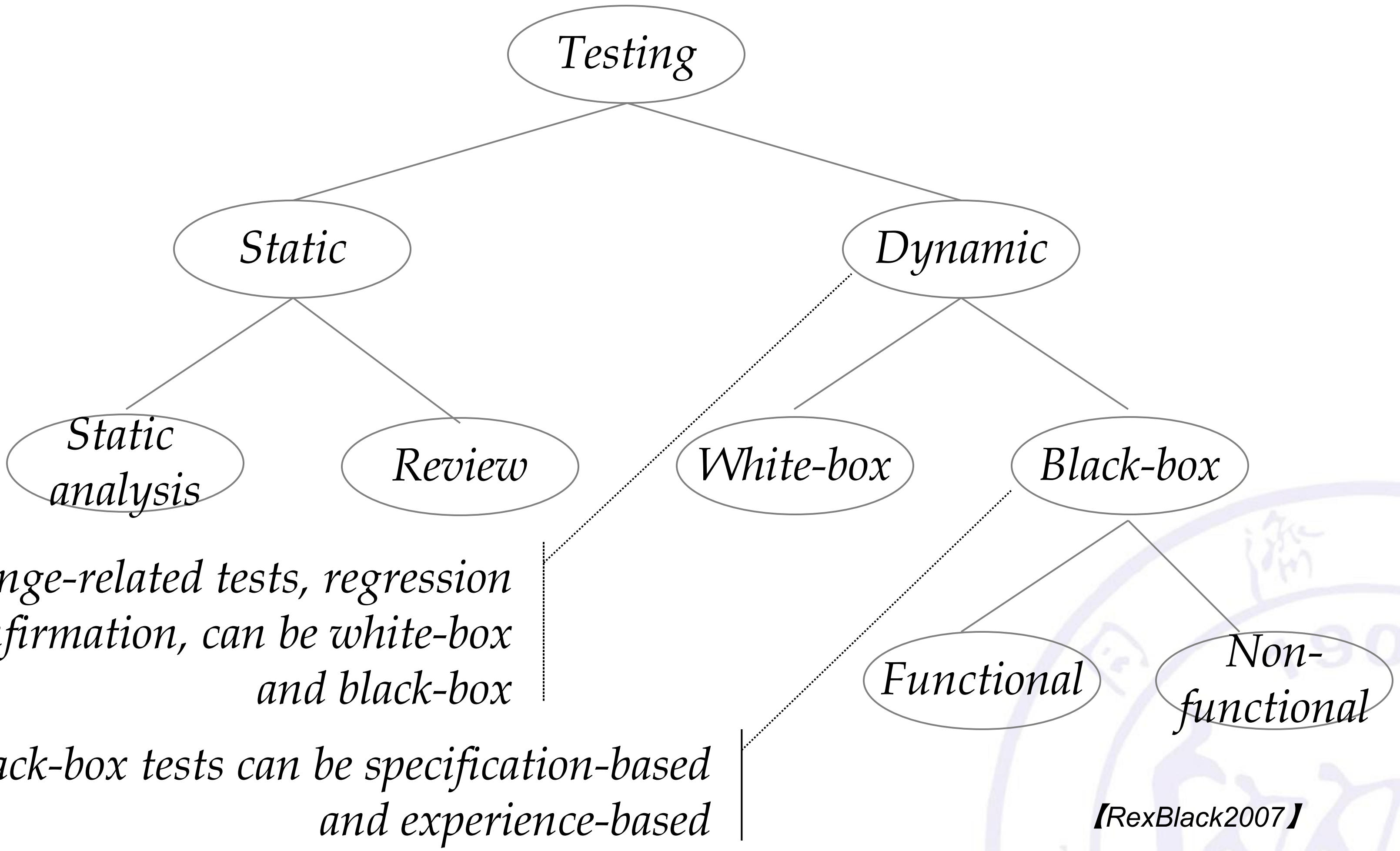
2.1 Software development models

2.2 Test levels or phases

2.3 Test types

2.4 Maintenance testing

2.3 Comparing Test Types



Change-related tests, regression and confirmation, can be white-box and black-box

Black-box tests can be specification-based and experience-based

【RexBlack2007】

2.3 ISO 9126 Quality Standard

Characteristic

Subchar-acteristics

Functionality: Suitability, accuracy, interoperability, security, compliance

Reliability: Maturity (robustness), fault-tolerance, recoverability, compliance

Usability: Understandability, learnability, operability, attractiveness, compliance

Efficiency: Time behavior, resource utilization, compliance

Maintainability: Analyzability, changeability, stability, testability, compliance

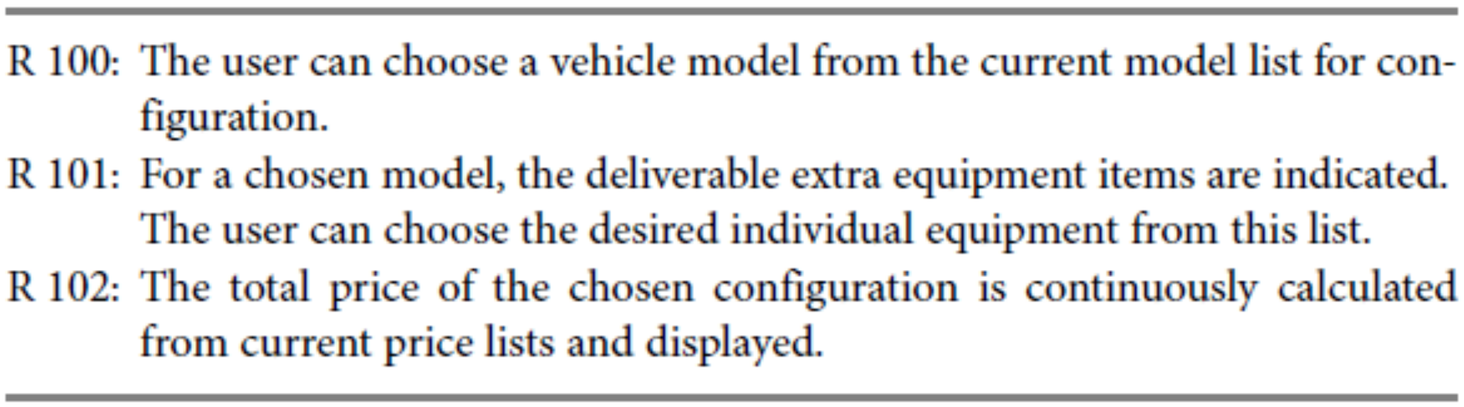
Portability: Adaptability, installability, co-existence, replaceability, compliance

Addressed by functional tests

Addressed by non-functional tests

2.3.1 Functional Testing

- All kind of tests that verify a system's input/output behavior
- Test basis: functional requirements
  - Characteristics: suitability, accuracy, interoperability, security
- Example



R 100:	The user can choose a vehicle model from the current model list for configuration.
R 101:	For a chosen model, the deliverable extra equipment items are indicated. The user can choose the desired individual equipment from this list.
R 102:	The total price of the chosen configuration is continuously calculated from current price lists and displayed.

2.3.1 Functional testing

Reasonable or required action not provided, inaccessible, or seriously impaired

- No add function on a calculator
- Add function implemented, “+” key doesn’t work
- Can only add integers, not real numbers

Right action, wrong result

- Add function: 2 + 2 = 5?
- Multiply function: 2 * 2 = 5?

2.3.1 Functional testing

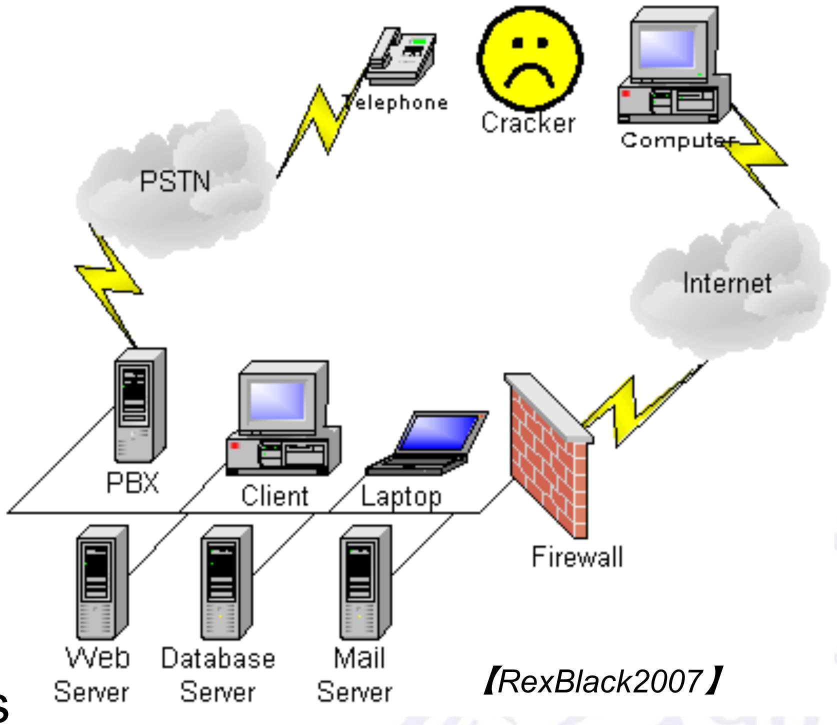
Right action, right result, wrong side-effect

- Divide function: 2/2=I (Roman numeral format)
- Minus function: 2.00-1.99=0(Not accurate enough)

Can be carried out at all levels of the test

2.3.1 Functional testing *- Security*

- Security threats include
  - Virii
  - Cracking into servers
  - Denial of service
- Common bad assumptions
  - Encryption (HTTPS) on Web server solves security problems
  - Buying a firewall solves problems
  - Unskilled network or system administrators can solve problems
- Specialized field of test expertise



【RexBlack2007】

The determined--or bored--cracker might break into any of your servers. One "exploit" can lead the cracker to other vulnerabilities--and to valuable data.

2.3.2 Non-functional Testing

The testing of a software application for its non-functional requirements.

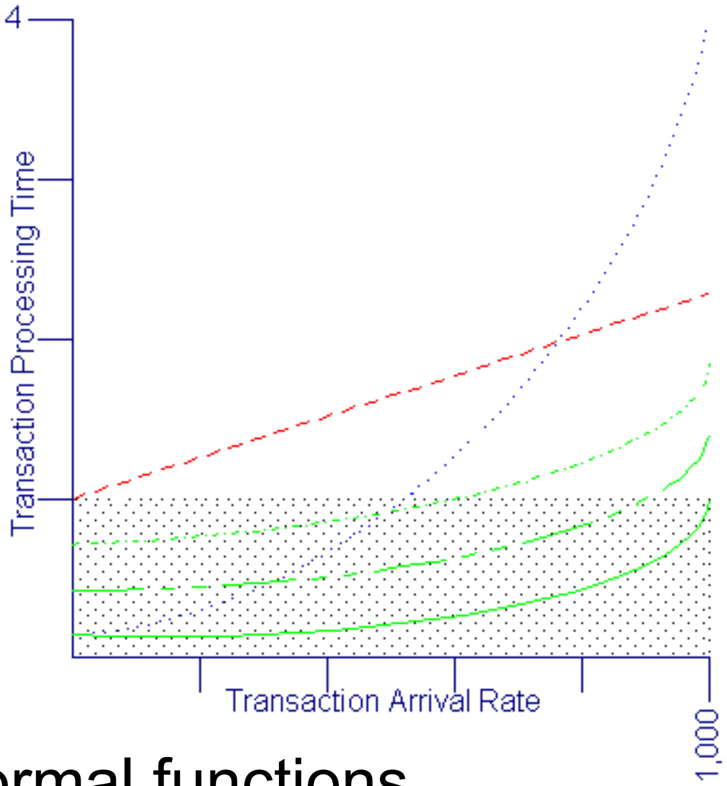
Often used interchangeably because of the overlap in scope between various non-functional requirements.

1. *Performance*
2. *Reliability*
3. *Usability*
4. *Volume*
5. *Maintainability*
6. *Portability*
7. *Robustness*
8. *Recoverability*
9. *Compatibility*

Can be carried out at all levels of the test, and mainly at system levels

2.3.3.1 *Performance And Reliability*

- Performance
  - Too slow throughout performance curve
  - Unacceptable “knee” in performance curve
  - Unacceptable performance degradation over time



- Reliability
  - System fails to complete normal functions
  - System functions normally, but randomly crashes or hangs

2.3.3.2 *Usability and User Interface*

- A system can function properly but be unusable by the intended customer
- Cumbersome interfaces that do not follow workflows
- Inaccessible functionality
- Inappropriately difficult for the users to learn
- Instructional, help, and error messages that are misleading, confusing, or misspelled

2.3.3.3 *Stress, Capacity, and Volume*

- Stress: extreme conditions cause failure
  - Combinations of error, capacity, and volume tests
- Capacity: functionality, performance, or reliability problems due to resource depletion
  - Fill hard drive or memory to 80+%
- Volume: functionality, performance, or reliability problems due to rate of data flows
  - Run 80+% of rated transactions per minute, number of simultaneous users, etc.

2.3.3.4 *Maintenance And Maintainability*

- Maintenance
  - Update and patch install and deinstall processes don’t work
  - Configurations can’t be changed appropriately (e.g., plug-and-play, hot plugging, adding disk space, etc.)

- Maintainability
  - Software itself (source code) not maintainable
  - Databases not upgradeable
  - Databases grow monotonically
  - Software not efficiently testable during maintenance; e.g., excessive regression

2.3.3.5 *Configuration and Portability*

- A single platform may be configured in many different ways in software
- A family of platforms may support various hardware configurations
- Are configuration changes handled?
  - Add disk space or other storage
  - Add memory
  - Upgrade or add CPU
- Portability to various environments

2.3.3.6 Other Functional/Non-Functional Tests

- Localization (user interface)
- Localization (operational)
- Standards and regulatory compliance
- Interoperability
- Error handling and recovery
- Disaster recovery
- Networked/internet-worked or distributed
- Timing and coordination
- Data quality
- Data conversion
- Operations
- Installation
- De-installation
- Date and time handling
- Documentation
- And many others…

2.3.2 Structure-Based Tests

- Tests based on how the system is built
  - Code
  - Data
  - Design
- Structure-based (white box) coverage can be measured after functional and non-functional specification-based (black box) tests are run to check for omissions
- This topic will be covered in more depth later…

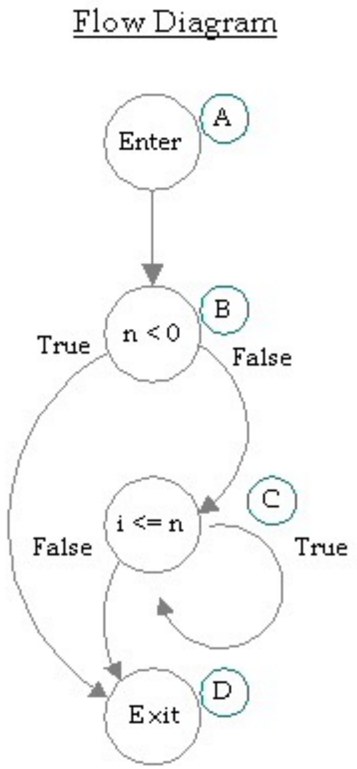
2.3.2 Structural Testing

The structural testing is the testing of the structure of the system or component.

Program

```
main()
{
  int i, n, f;
  printf("n = ");
  scanf("%d", &n);
  if (n < 0) {
    printf("Invalid: %d\n", n);
    n = -1;
  } else {
    f = 1;
    for (i = 1; i <= n; i++) {
      f *= i;
    }
    printf("%d! = %d.\n", n, f);
  }
  return n;
}
```

Flow Diagram



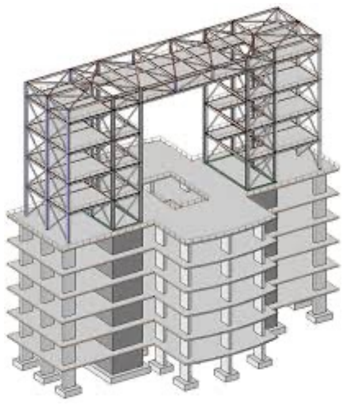
Basis Paths

1. ABD
2. ABCD
3. ABCCD

Basis Tests

	Input	Expect
1.	-1	Invalid: -1
2.	0	0! = 1.
3.	1	1! = 1.

【RexBlack2007】



2.3.3 Regression and Confirmation

- Regression testing checks the effects of changes, since even small, localized, isolated changes, don't always have small, localized, or isolated effects
- Regression strategies are covered in the next section
- Confirmation testing confirms that…
  - Changes made to the system are present
  - Bug fixes introduced in the system solve the observed symptoms
- Repeatability of tests helps with regression and confirmation testing
- Automation, covered in depth later, is also helpful

2.3.4 Testing Related to Changes

- When changes are implemented
  - Tests must show that earlier faults are really repaired (retesting)
- There is risk of unwanted side effects
  - Repeating other tests (regression testing)
- Regression test: a new test of a previously tested program following modification, to ensure that defects have not been introduced or uncovered
  - May be performed at all test levels, applied to functional, nonfunctional and structural tests
- Test cases must be well documented and reusable
  - Test automation

2.3.4 Testing Related to Changes (cont.)

- How extensive?
  - 1. Rerunning all tests that have detected failures whose reasons have been fixed (defect retest, confirmation testing)
  - 2. Testing all program parts that were changed or corrected (testing of altered functionality)
  - 3. Testing of all program parts or elements that were newly integrated (testing of new functionality)
  - **4. Testing of the whole system (complete regression test)**

2.3.4 Testing Related to Changes (cont.)

- In practice, time-consuming and expensive
  - Balancing risk and cost
- Test selection strategies
  - Repeating only the high-priority tests according to the test plan
  - In the functional test, omitting certain variations (special cases)
  - Restricting the tests to certain configurations only (e.g., English product version only, only one operating system version)
  - Restricting the test to certain subsystems or test levels

Exercise: Omninet Test Types A2.1.4

- Referring back to the levels you selected in the previous exercise, list for each level the test targets or types you would include in each level.
- How does the lifecycle model you selected in a previous exercise affect the amount of regression testing?
- How does the lifecycle model you selected in a previous exercise affect the amount of confirmation testing?
- Discuss.

2 Testing throughout the life cycle

2.1 Software development models

2.2 Test levels or phases

2.3 Test types

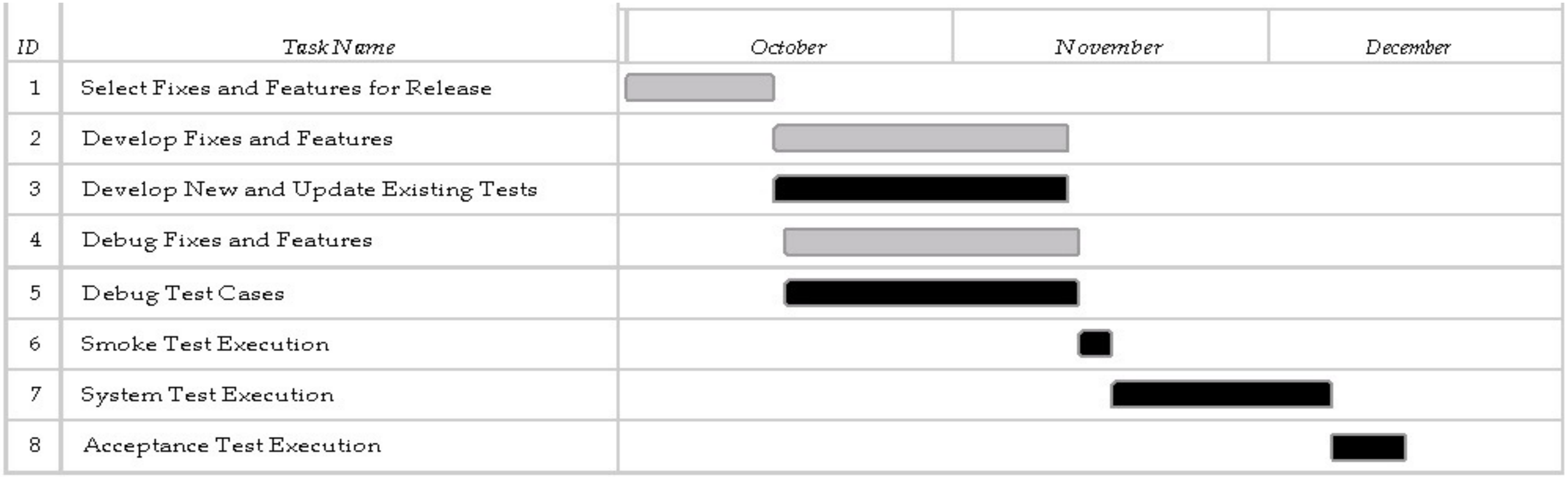
2.4 Maintenance testing

2.4 Reasons for Maintenance

- Three typical triggers for maintenance and maintenance testing
  - Modification: enhancements, bug fixes, operational environment changes, patches
  - Migration: a new supported environment
  - Retirement: end-of-life of a subsystem or entire system triggers replacement
- Maintenance testing addresses the change itself--and what wasn’t changed and shouldn’t change

2.4 Testing Maintenance Releases

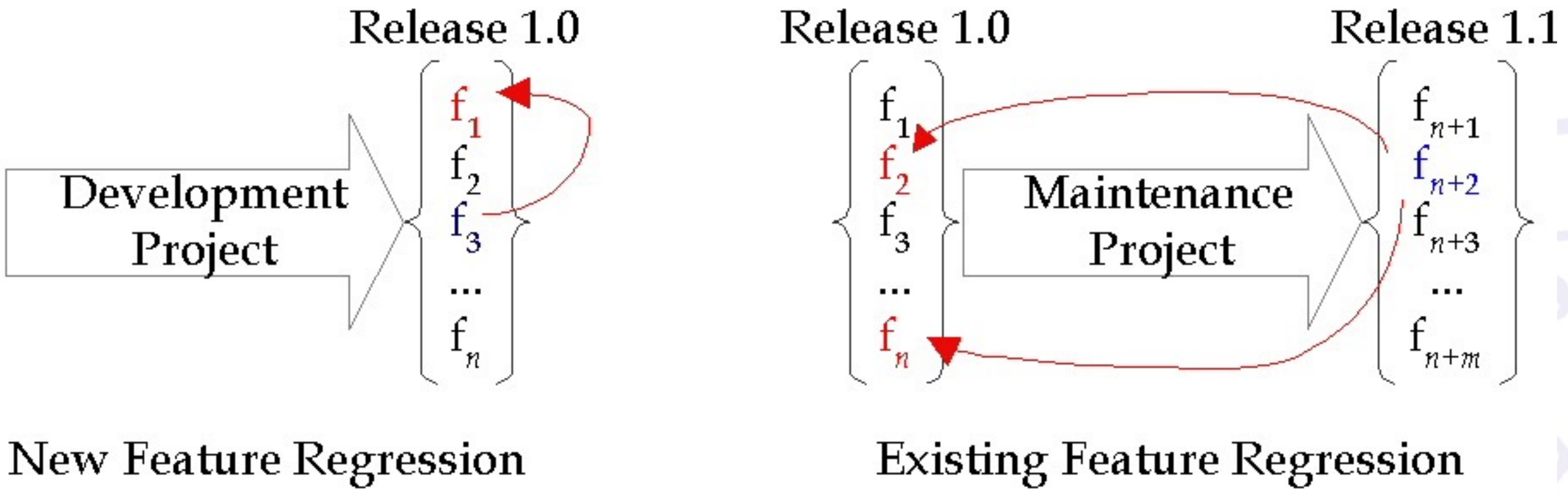
- Regression is the big risk for maintenance releases
- Some organizations try to put a major release worth of features into a short maintenance release
- Time to develop new tests is scarce
- Large project test estimation rules-of-thumb fail



ID	Task Name
1	Select Fixes and Features for Release
2	Develop Fixes and Features
3	Develop New and Update Existing Tests
4	Debug Fixes and Features
5	Debug Test Cases
6	Smoke Test Execution
7	System Test Execution
8	Acceptance Test Execution

2.4 Regression

- Regression (misbehavior of a previously correct function, attribute, or feature due to a change) types:
  - Local (fix creates new bug)
  - Exposed (fix reveals existing bug)
  - Remote (fix in one area breaks something in another area)
- Regression can affect new and existing features

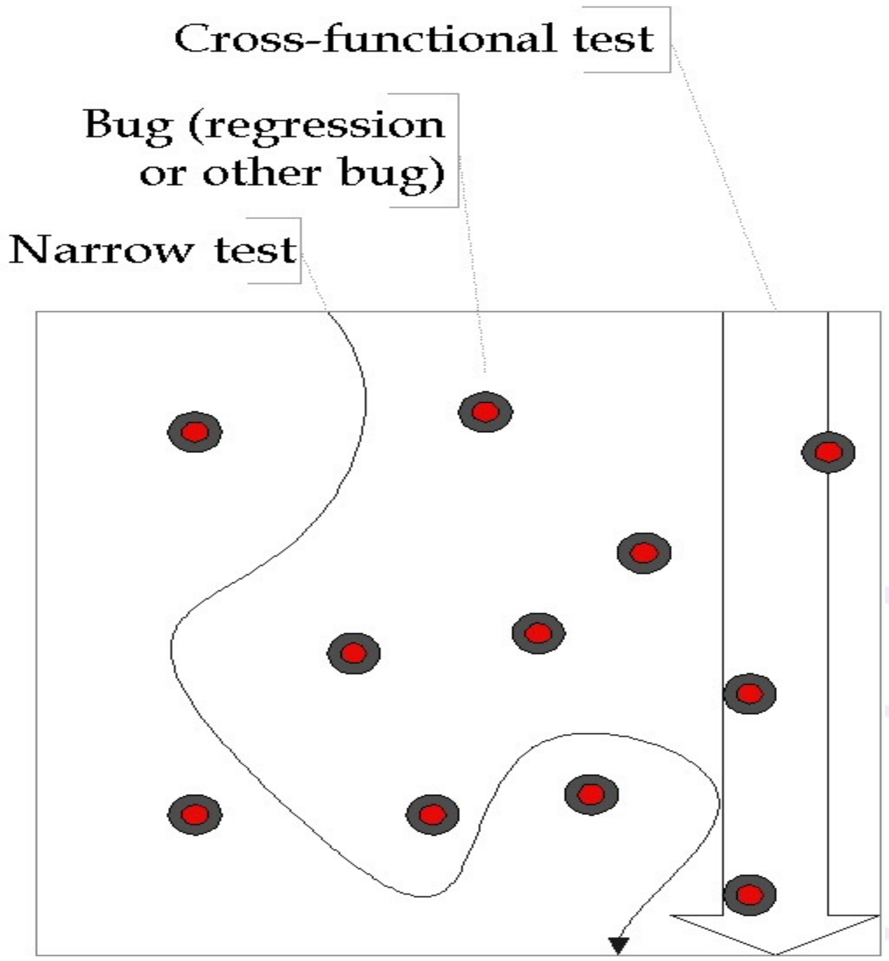


2.4 Regression Strategy 1: Repeat All Tests

- If our tests address the important quality risks, then repeating all tests should find the most important regressions
- Automation is the only practical means for large complex systems
- We'll cover automation more thoroughly in a later section

2.4 Regression Strategy 2: Repeat Some Tests

- Often, full automation is impossible
- Select some tests to repeat
  - Traceability
  - Change/impact analysis
  - Risk analysis
- Use cross-functional tests for “accidental regression testing”
- Can use code coverage to assess level of risk



2.4 Three Other Regression Strategies

- Release more slowly
  - Release every six months rather than every month
  - Partial or even full repetition can increase coverage on bigger releases
- Combine emergency patches with slower release process to allow for flexibility while keeping regression risk low
- Use customer or user testing
  - Beta for mass-market software
  - Pilot, staged or phase, parallel release for IT

Discussion: Omninet Maintenance

- Assume you can fully automate the testing of the Omninet kiosk and call center. If a post-release change is made to the call-center user interface that does not affect functionality, what would you retest?

- Assume you have not automated the testing of the Omninet kiosk and call center. If the same change is made, what would you retest? What if the change did affect functionality?

Chapter 2 Summary

2.1 Software development models

- V or incremental model ?

2.2 Test levels or phases

- Unit, Integration, System, Acceptance, Deployment, Operational etc.

2.3 Test types

-Specification Based: Functional/Non-functional

-Structure Based

2.4 Maintenance testing

-Maintenance, Regression and Confirmation

上海市嘉定区曹安公路4800号，同济大学嘉定校区软件学院