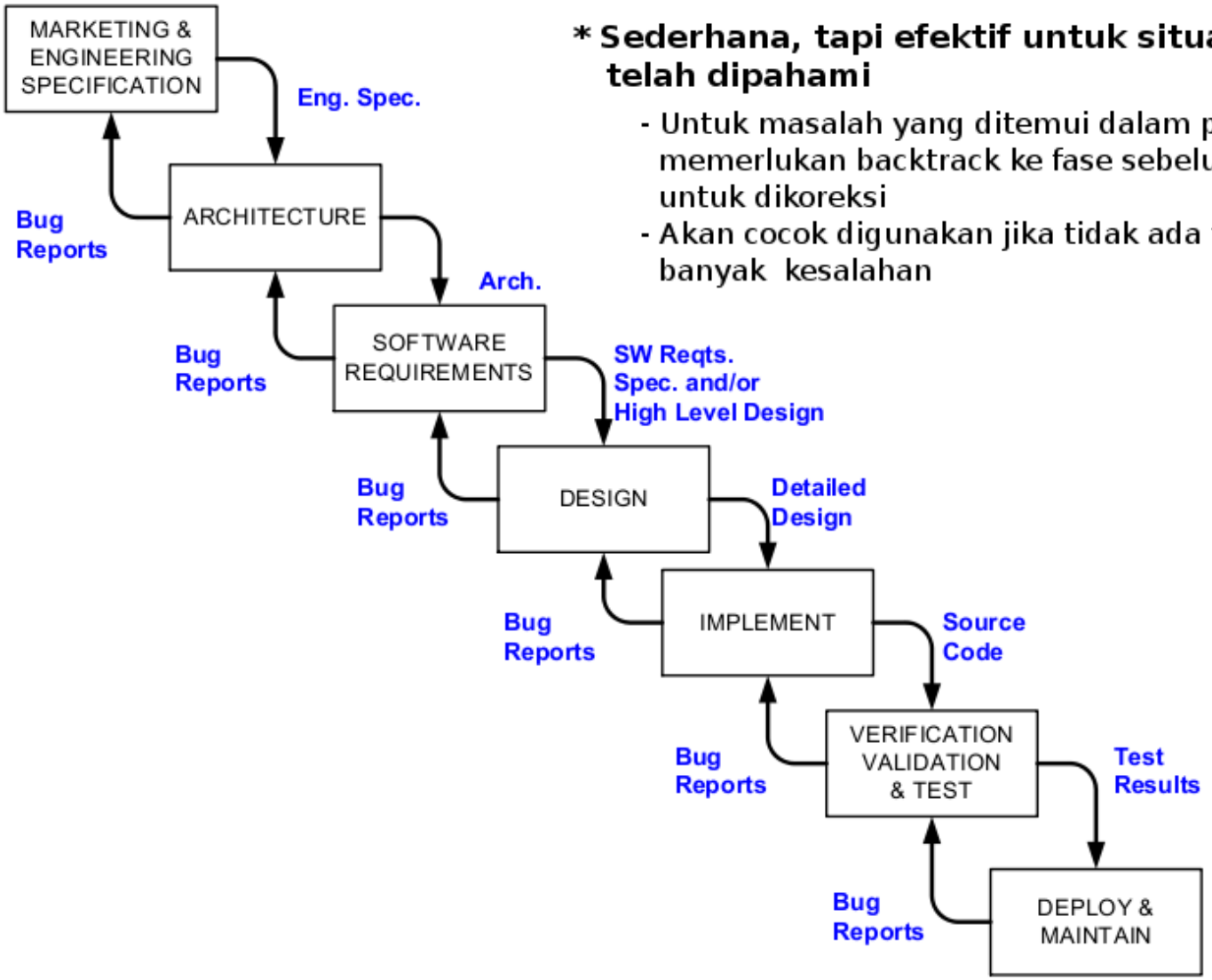
Pengujian Software Embedded

Kuliah#8 TSK-612 Sistem Embedded Terdistribusi - TA 2011/2012

Eko Didik Widianto

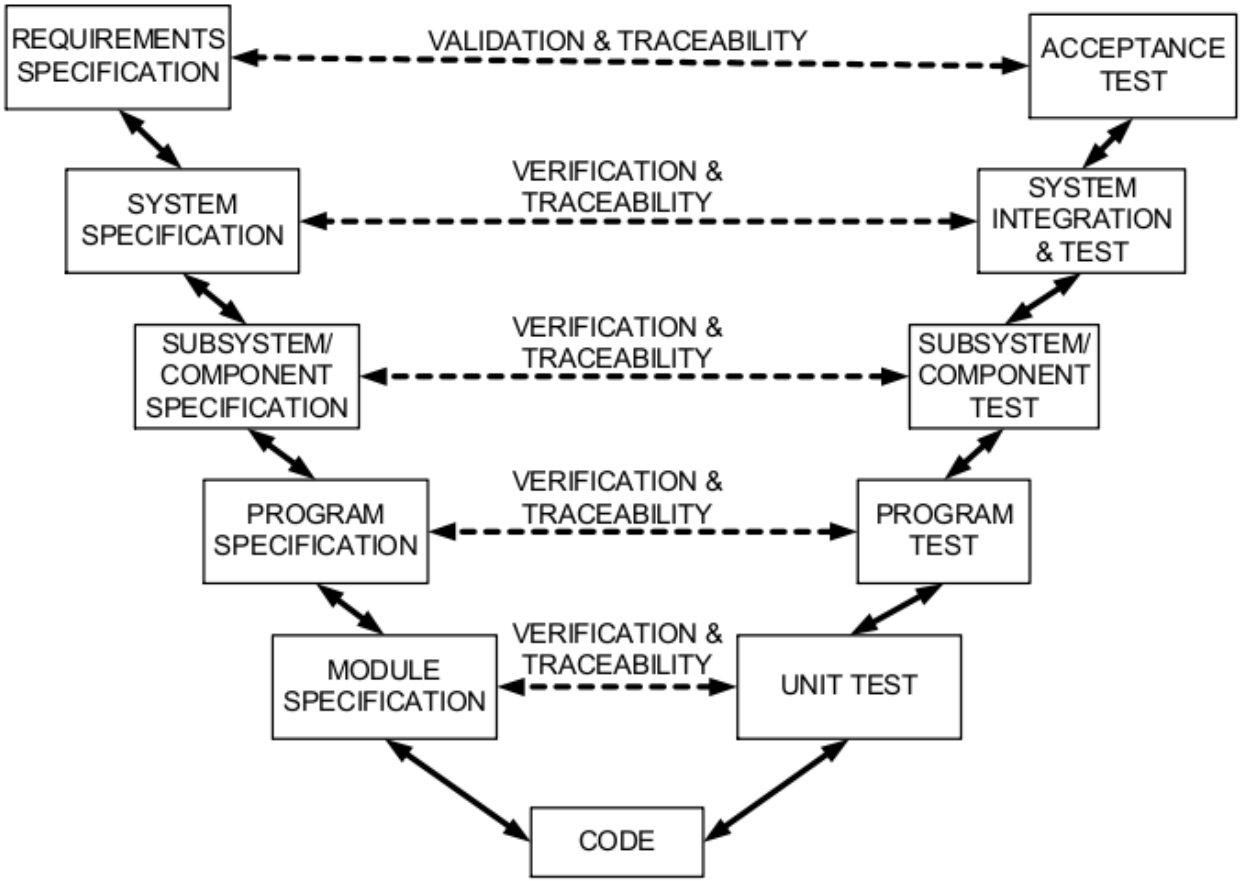
Teknik Sistem Komputer - Universitas Diponegoro

Review: Metodologi Pengembangan Waterfall



* **Sederhana, tapi efektif untuk situasi yang telah dipahami**
  - Untuk masalah yang ditemui dalam proses, memerlukan backtrack ke fase sebelumnya untuk dikoreksi
  - Akan cocok digunakan jika tidak ada terlalu banyak kesalahan

Review: Metodologi Pengembangan V-Model



Tentang Kuliah #8 Pengujian

- Testing is **an attempt to find bugs**
  - The reasons for finding bugs vary
  - Finding all bugs is impossible
- Various types of testing for **various situations**
  - Exploratory testing – guided by experience
  - White Box testing – guided by software structure
  - Black Box testing – guided by functional specifications
- Various types of testing throughout **development cycle**
  - Unit test
  - Subsystem test
  - System integration test
  - Acceptance test
  - Beta test

Kompetensi Dasar

- Setelah menyelesaikan bab ini, mahasiswa akan mampu:
  - [C2] menjelaskan dan membedakan jenis-jenis pengujian
  - [C3] menerapkan prinsip-prinsip pengujian di pengembangan sistem embedded terdistribusi
  - [C4] merancang testplan untuk satu desain sistem embedded terdistribusi

Link dan Acknowledgement

- Link
  - Website: http://didik.blog.undip.ac.id/2012/03/06/kuliah-tsk-612-sistem-embedded-terdistribusi-2011/
  - Email: didik@undip.ac.id
- Acknowledgement:
  - Materi dalam slide ini diambil dari http://www.ece.cmu.edu/~ece649/[ECE649]

Bahasan

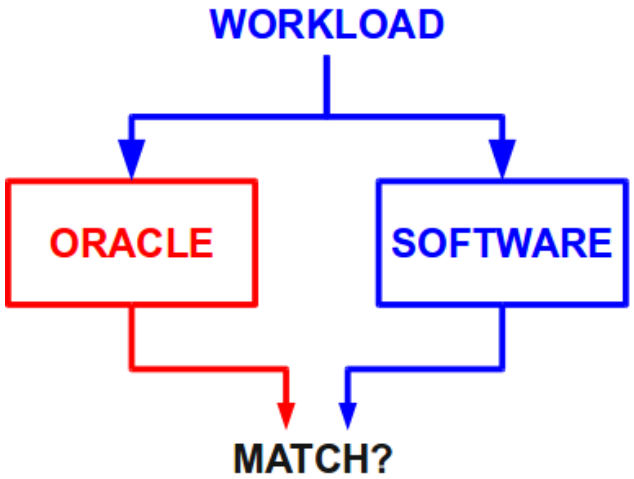
Bahasan

Definisi Pengujian

- **Testing is performing all of the following**:
  - Providing software with inputs (a “workload”)
  - Executing a piece of software
  - Monitoring software state and/or outputs for expected properties, such as:
    - Conformance to requirements
    - Preservation of invariants (e.g., never applies brakes and throttle together)
    - Match to expected output values
    - Lack of “surprises” such as system crashes or unspecified behaviors
- **General idea is attempting to find “bugs” by executing a program**
- The following are potentially useful techniques, **but are not testing**:
  - Model checking
  - Static analysis (lint; compiler error checking)
  - Design reviews of code
  - Traceability analysis

Bahasan

Istilah dalam Pengujian



- **Workload:** Inputs applied to software under test
  - Each test is performed with a specific workload
- **Behavior:** Observed outputs of software under test
  - Sometimes special outputs are added to improve observability of software (e.g., “test points” added to see internal states)
- **Oracle**: A model that perfectly predicts correct behavior
  - Matching behavior with oracle output tells if test passed or failed
  - Oracles come in many forms:
    - Human observer
    - Different version of same program (simulation)
    - Hand-created script of predicted values based on workload

Pengertian Bug

- **Simplistic answer:**
  - A “bug” is a software **defect** = incorrect software
  - A software defect is an instance in which the software violates the specification
- More realistic answer – a “bug” can be one or more of the following:
  - **Failure** to provide required behavior
  - Providing an **incorrect** behavior
  - Providing an **undocumented** behavior or behavior that is **not required**
  - **Failure** to conform to a design constraint (e.g., timing, safety invariant)
  - Omission or **defect** in requirements/specification
  - Instance in which software performs as designed, but it’s the **“wrong”** outcome
  - **Any “reasonable” complaint** from a customer
- The goal of most testing is to attempt to find bugs in this expanded sense

Jenis-jenis Pengujian

- Testing **styles**: (how to test)
  - Smoke testing
  - Exploratory testing
  - Black box testing
  - White box testing
- Testing **situations**: (where/when to test)
  - Unit test
  - Subsystem test
  - System integration test
  - Acceptance test
  - Beta test

Bahasan

Smoke Testing

- **Quick test** to see if software is operational
  - Idea comes from hardware realm – turn power on and see if smoke pours out
  - Generally simple and easy to administer
  - Makes no attempt or claim of completeness
  - Smoke test for car: turn on ignition and check:
    - Engine idles without stalling
    - Can put into forward gear and move 5 feet, then brake to a stop
    - Wheels turn left and right while stopped
- Good for catching **catastrophic errors**
  - Especially after a new build or major change
  - Exercises any built-in internal diagnosis mechanisms
- But, **not usually a thorough test**
  - More a check that many software components are “alive”

Bahasan

Exploratory Testing

- **A person exercises** the system, looking for unexpected results
  - Might or might not be using documented system behavior as a guide
  - Is especially looking for “strange” behaviors that are not specifically required nor prohibited by the requirements
- **Advantages**
  - An **experienced, thoughtful tester** can find many defects this way
  - Often, the defects found are ones that would h**ave been missed** by more rigid testing methods
- **Disadvantages**
  - Usually no documented **measurement of coverage**
  - Can leave big holes in coverage due to tester bias/blind spots
  - An **inexperienced**, non-thoughtful tester probably won’t find the important bugs

Bahasan

Black Box Testing

- Tests designed with knowledge of **behavior**
  - But without knowledge of implementation
  - Often called **“functional” testing**
- Idea is to test **what software does**, but not how function is implemented
  - Example: cruise control black box test
    - Test operation at various speeds
    - Test operation at various underspeed/overspeed amounts
    - BUT, no knowledge of whether lookup table or control equation is used
- Advantages:
  - Tests the final behavior of the software
  - Can be written independent of software design
    - Less likely to overlook same problems as design
  - Can be used to test different implementations with minimal changes
- Disadvantages:
  - Doesn’t necessarily know the **boundary cases**
    - For example, won’t know to exercise every lookup table entry
  - Can be **difficult to cover all portions** of software

Contoh Black Box Testing

- Assume you want to test a floating point square root function: **sqrt(x)**
  - sqrt(0) = 0 (boundary condition)
  - sqrt(1) = 1 (behavior changes between <1 and >1)
  - sqrt(9) = 3 (test some number greater than 1)
  - sqrt(.25) = .5 (test some number less than 1)
  - sqrt(-1) => error (test an out of range input)
  - sqrt(FLT_MAX) => ... (test maximum numeric range)
  - sqrt(FLT_EPSILON) => ...(test smallest positive number)
  - sqrt(NaN) => NaN (test for Not a Number input values)
  - Pick random positive numbers and confirm that: sqrt(x) * sqrt(x) = x
- Other types of possible results to monitor:
  - Monitor numerical accuracy/stability for floating point math
  - Check to see if software crashes on some inputs

Bahasan

White Box Testing

- Tests designed with **knowledge of software design**
  - Often called **“structural” testing**
- Idea is to exercise software, knowing how it is designed
  - Example: cruise control white box test
    - Test operation at every point in control loop lookup table
    - Tests that exercise both paths of every conditional branch statement
- **Advantages**:
  - Usually helps getting good coverage (tests are specifically **designed for coverage**)
  - Good for ensuring boundary cases and special cases get tested
- **Disadvantages**:
  - 100% coverage tests might not be good at assessing functionality for “surprise” behaviors and other testing goals
  - Tests based on design might miss bigger picture system problems
  - Tests need to be changed if implementation/algorithm changes

Contoh White Box Testing

- Assume you want to test a floating point **square root function: sqrt(x)**
  - Uses lookup table spaced at every 0.1 between zero and 10
  - Uses iterative algorithm at and above value of 10
  - Test sqrt(x) for negative numbers (expect error)
  - Test sqrt(x) for every value of x in middle of lookup table: 0.05, 0.15, ... 9.95
  - Test sqrt(x) exactly at every lookup table entry: 0, 0.1, 0.2, ... 10.0
  - Test sqrt(x) at 10.0 + FLT_EPSILON
  - Test sqrt(x) for some numbers that exercise interpolation algorithm
- **Main differences** from Black Box Testing:
  - Tests **exploit knowledge** of software design & coding details
  - Usually strives for 100% coverage of known properties
    - e.g., lookup table entries
  - Digs deepest at algorithmic discontinuities & branches
    - e.g., lookup table boundaries and center values

Bahasan

Testing Coverage

- **“Coverage” is a notion how completely testing has been done**
  - Usually a percentage (e.g., “97% branch coverage”)
- **White box testing coverage**
  (this is the usual use of the word “coverage”):
  - Percent of conditional branches where both sides of branch have been tested
  - Percent of lookup table entries used in computations
- **Black box testing coverage:**
  - Percent of requirements tested
  - Percent of documented exceptions exercised by tests
  - But, must relate to externally visible behavior or environment, not code structure
- **Important note: 100% coverage is not “100% tested”**
  - good coverage is necessary, but not sufficient to achieve good testing

What Can You Test For?

From [kaner]

- Conformance to specification
- Correctness
- Usability
- Boundary conditions
- Performance
- State transitions
- Mainstream usage (against scenarios)
- Load testing (events, memory usage, error rates)
- Error recovery
- Security
- Compatibility/configuration (equipment variations; old versions)
- Installability/serviceability

What Errors Do People Make?

From [kaner]

- Sometimes it is a good idea to look for common programming errors
  - User interface
  - Error handling
    - Incorrectly handled errors
    - Missed error checks
  - Boundary (e.g., overflow/underflow)
  - Calculation errors (wrong algorithm)
  - Control flow
  - Race conditions
  - Load conditions
  - Hardware error handling
  - Version control errors
  - Documentation
- **Remember** that test programs are programs too, and can have errors

Complete Testing Is Usually Impossible

- How many test cases to completely test the following :

```
myfunction(int a, int b, int c)
```

- **Assume 32-bit integers**:
  - $2^{32} * 2^{32} * 2^{32} = 2^{96} = 7.92 * 10^{28}$=> at 1 billion tests/second
  - Testing all combinations takes 2,510,588,971,096 years
  - Takes longer for complete tests if there are any state variables inside function
  - Takes longer if there are environmental dependencies to test as well
- This is a fundamental problem with testing – you can't test everything
  - Therefore, need some notion of how much is **"enough"** testing
- Even if you could test "everything," there is more to "everything" than just all possible input values
  - Kaner has published a list of more than 100 types of coverage (http://www.kaner.com/coverage.htm)

Bahasan

“White Box” Testing – Based on Structure

- Look at program structure & attempt to cover various aspects with tests
  - Traceability is to design & implementation
- Types of coverage:
  - All paths & states in a statechart
  - All statements
  - All branches
  - All data dependencies
  - All inputs
  - All exceptions/error conditions
  - ...
- Coverage doesn’t mean you necessarily did the “best” tests
  - But, it gives some confidence that you didn’t completely miss an area of test

“Black Box” Testing – Based On Functionality

- Look at specification & attempt coverage
  - Traceability is to specification
- Types of coverage:
  - All numbered requirements
  - All scenarios
  - Standard bag of tricks:
    - Boundary testing
    - Special value testing (e.g., string terminator characters)
  - All implicit requirements
    - Doesn’t crash
    - Reports reasonable error codes
  - All implementation techniques for approaches
    - E.g., a trig. Function could be:
      » Table lookup + interpolation
      » Polynomial approximation
      » Computed by hardware coprocessor

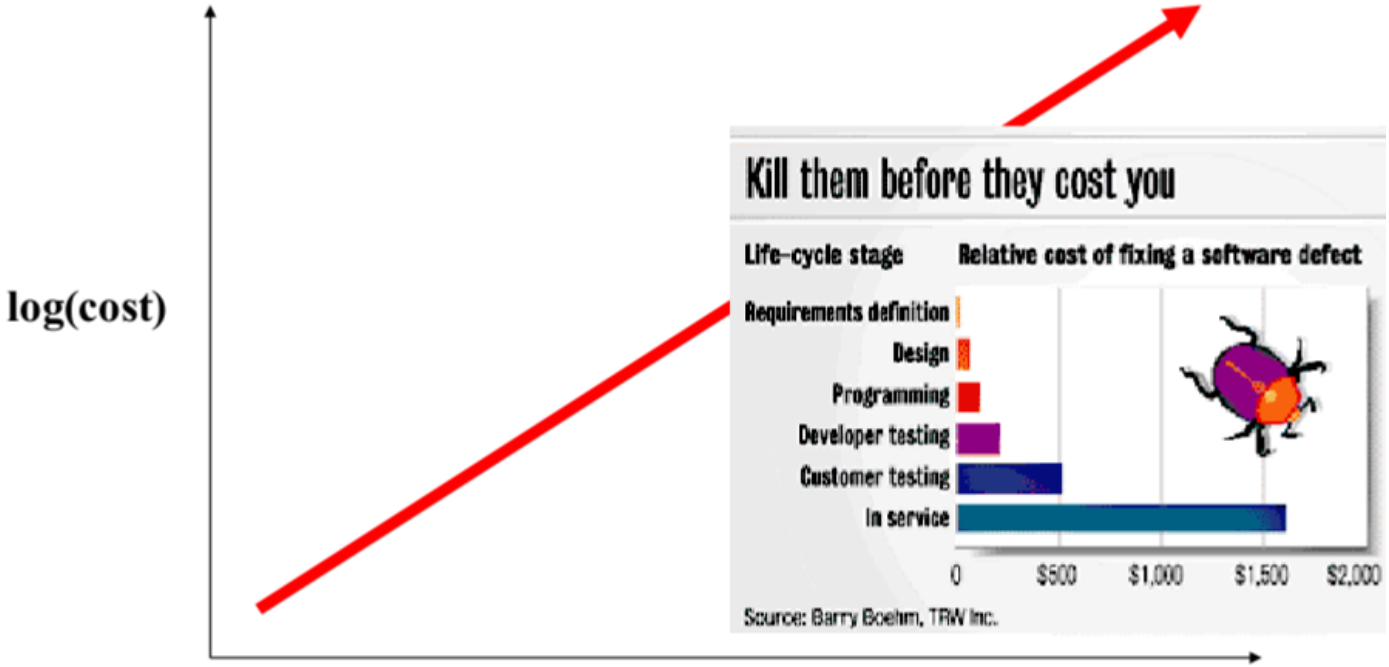
Which Is Better – Black Box or White Box?

- Both types of test have their place
- White box best for ensuring details are correct in implementation
  - Tests are designed to ensure that code conforms to design
  - Exercises different nooks and crannies of code in a way black box usually can't
  - Best way to ensure good coverage metrics
- Black box best for requirements-based testing
  - Emphasis on meeting requirements and, in general, overall behavior
  - Tests are designed to ensure that the software actually works!
- Smoke tests – good for quick checks, but not for rigorous evidence of correctness
- Exploratory tests – can be helpful in finding bugs before users find them, but does not assure coverage

Mengapa Perlu Melakukan Pengujian

- **Because someone makes us test**
  - This is an unsatisfactory reason
  - Unlikely to be productive because it is a bureaucratic imposition
  - BUT, can happen for certification (e.g., “must have 100% branch coverage”)
- Want to find bugs in program so they can be removed
  - Tests are designed to ensure that important operations work properly
  - Usually, approach is to test until we stop finding bugs
  - When “important” bugs are fixed, product is shipped
  - Most often, this is how desktop software testing works
- Want to test the hypothesis that there are no (important) bugs
  - Tests are never supposed to find an (important) defect
  - Keep testing until it seems unlikely any bugs are really there
  - If a bug is found this indicates a software development process failure!
  - In general, this is the goal of testing for safety critical systems

Cost untuk Mencari dan Memperbaiki Bug



Notes:

- Cost includes both money and lost time to market
- Higher costs late in lifecycle are when profits are low, making it worse

Kapan Melakukan Test

- **Different phases** of development cycle
  - Unit test – development
  - Subsystem test – software module integration
  - System test – system integration
  - Acceptance test – product shipment
  - Beta test – selected customer use of system
- **Different people** play roles of tester:
  - Programmer often does own testing for unit test
  - Independent testers are often involved in subsystem, system & acceptance tests
  - Customers are often involved in acceptance & beta tests

Bahasan

Unit Test

- **A "unit"** is a few lines of code to a small program
  - Usually created by a single developer
  - Usually tested by the programmer who created it
- **Purpose** of unit test:
  - Try to find all the "obvious" defects
  - Can be done before and/or after code review
- **Approaches** (mostly exploratory & white box)
  - Exploratory testing makes a lot of sense
    - Helps programmer build intuition and understand code
  - White box testing to ensure all portions of code exercised
    - Often useful to ensure 100% arc and state coverage for statecharts
  - Some black box testing as a "**sanity check**"

Bahasan

Subsystem Test

- **A "subsystem"** is a relatively complete software component (e.g., engine controller software)
  - Usually created by a team of developers
  - Usually tested by a combination of programmers and independent testers
- **Purpose** of subsystem test:
  - Try to find all the "obvious" defects
  - Can be done before and/or after code review
- **Approaches** (mostly white box; some black box)
  - White box testing is key to ensuring good coverage
  - Black box testing should at a minimum check interface behaviors against specification to avoid system integration surprises
  - Exploratory testing can be helpful, but shouldn't find a lot of problems
  - Smoke test is helpful in making sure a change in one part of subsystem doesn't completely break the entire subsystem

Bahasan

System Integration Test

- **A "system"** is a complete multi-component system (e.g., a car)
  - Often created by multiple teams organized in different groups
  - Usually tested by independent test organizations
- **Purpose** of system integration test:
  - Assume that components are mostly correct; ensure system behaves correctly
  - Find problems/holes/gaps in interface specifications that cause system problems
  - Find unexpected behavior in system
- **Approaches** (mostly black box)
  - Tends to be mostly black box testing to ensure system meets requirements
  - Want white box techniques to ensure all aspects of interfaces are tested
  - Exploratory testing can help look for strange component interactions
  - Smoke tests are often used to find version mismatches and other problems in independent components

Bahasan

Acceptance Test

- Acceptance tests ensure system provides all advertised functions
  - Testing performed **by a customer**
  - Might also involve a certification authority or independent test observer
- **Purpose** of acceptance test:
  - Does the system meet all requirements to be used by a customer?
  - Usually the last checkpoint before shipping a system
  - Might be performed on all systems to check for hardware defects/manufacturing defects, not just software design problems
  - In a mature software process, it is a quality check on the entire process
    - OUGHT to find few or no significant bugs if software has high quality
    - If bugs are found, it means you have a quality problem
- **Approaches**
  - Usually black box testing of system vs. 100% of high level product requirements

Bahasan

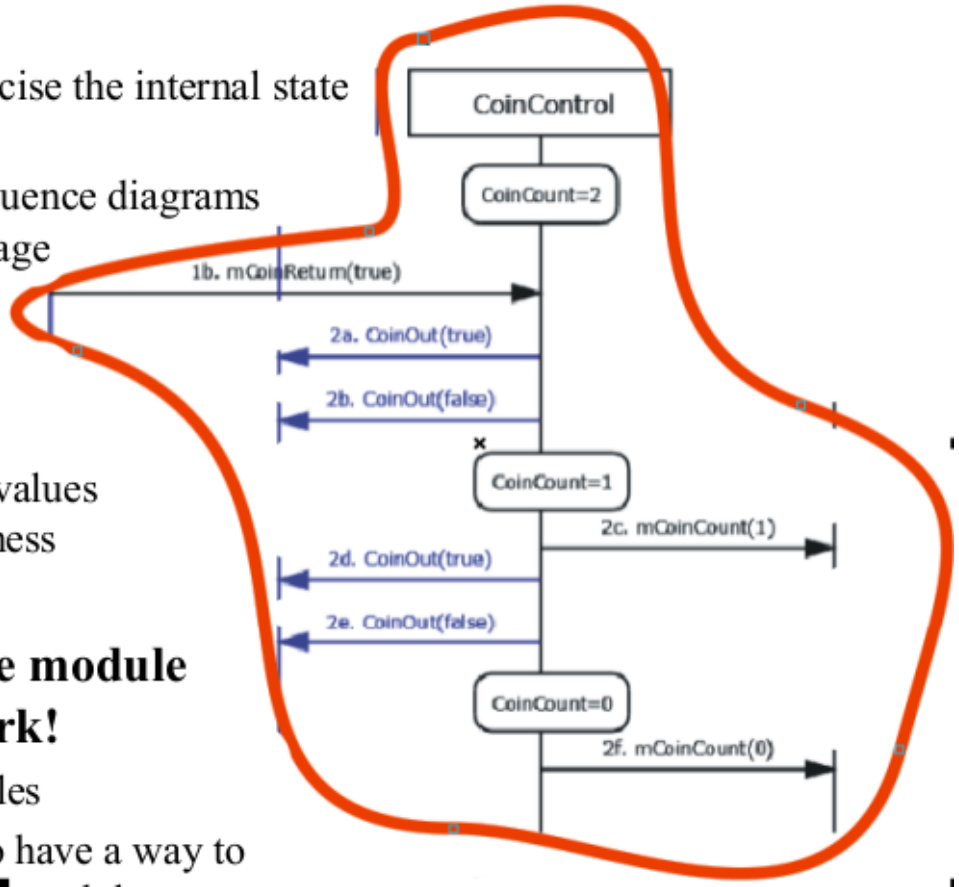
Beta Test

- A “beta version” is complete software that is “close” to done
  - Theoretically all defects have been corrected or are at least known to developers
  - Idea is to give it to sample users and see if a huge problem emerges
- Purpose of beta test:
  - See if software is good enough for a friendly, small user community (limit risk)
  - Find out if “representative” users are likely to stimulate failure modes missed by testing
- Approaches
  - This is almost all exploratory testing
    - Assumption is that different users have different usage profiles
    - Hope is that if small user community doesn’t find problems, there won’t be many important problems that slip through into full production
- Defect in beta test means you have a significant hole somewhere
  - If you have a good process, it is likely a requirements issue

Idea Behind Unit Test

- **Isolate single module and feed it direct inputs**
  - Feed in inputs that exercise the internal state machine
  - Base tests on single sequence diagrams and on statechart coverage

Test Input

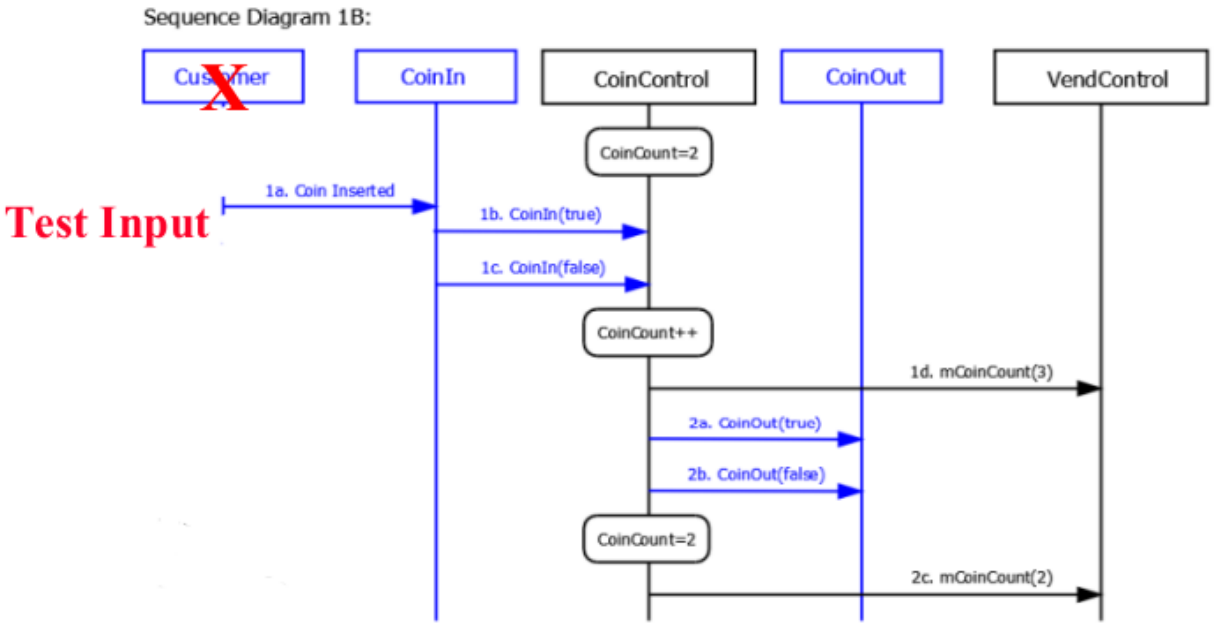


  - Monitor state machine values and outputs for correctness

- **ONLY LOAD a single module into the test framework!**
  - Plus test/monitor modules
  - Might be a good idea to have a way to set internal state in your module too

Idea Behind Integration Test

- Run all modules in a Sequence Diagram
  - Artificially set up state information to meet preconditions
  - Feed primary inputs from test harness; let rest of arcs run on their own
  - Make sure other arcs perform as expected



Acceptance Test

- Ensure system as a whole actually meets requirements
  - In simple systems, testing **all scenarios** suffices
  - In real systems, need to test **sequences of Use Cases**
  - In our system we have simulated passengers
    - So make sure passenger workload covers all reasonable usages
- If you don't have a simulated workload:
  - Go through the high level system requirements and get coverage
  - Go through all the use cases and get coverage
  - Go through all the high level scenarios and get coverage
- Traditionally this is called a "**Factory Acceptance Test**"
  - The factory (manufacturer) accepts it as a viable product to sell
  - Traceability is to functional requirements for entire system

Lisensi

Creative Common Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)

- Anda bebas:
  - untuk **Membagikan** — untuk menyalin, mendistribusikan, dan menyebarkan karya, dan
  - untuk **Remix** — untuk mengadaptasikan karya
- Di bawah persyaratan berikut:
  - **Atribusi** — Anda harus memberikan atribusi karya sesuai dengan cara-cara yang diminta oleh pembuat karya tersebut atau pihak yang mengeluarkan lisensi.
    - Cantumkan sumber asal file ini, yaitu `http://didik.blog.undip.ac.id/2012/03/06/kuliah-tsk-612-sistem-embedded-terdistribusi-2011/`
  - **Pembagian Serupa** — Jika Anda mengubah, menambah, atau membuat karya lain menggunakan karya ini, Anda hanya boleh menyebarkan karya tersebut hanya dengan lisensi yang sama, serupa, atau kompatibel.
- Lihat: Creative Commons Attribution-ShareAlike 3.0 Unported License