Testing Strategies / Methods

BY EVAN OLDS
CPT S 422

Priorities

1. Above all else – solid knowledge of how code works
    1. Know all the features of language you're using
    2. Know what's going on beneath your code – things that are done automatically for you
        1. Standard library stuff
        2. Compiler-hidden stuff like destructor calls and deep vs. shallow copies
2. Design the software itself as something that's testable
    1. If it's not flexible enough, you may not even be able to test the necessary components
    2. Much more on this later

Priorities

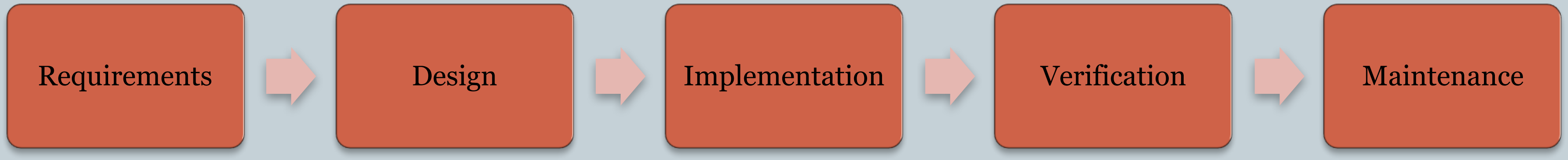
3. Understanding of testing
   - strategies/methods (← you are here)
   - tools
   - frameworks
   - flexibility
   - much much more

About Software Development Models

- Many different models/philosophies for software design
- Some are common enough to be recognized by most industry workers, but even the common ones might be defined slightly differently by different people or companies
- Get general idea of models along with strengths and weaknesses

Waterfall Software Development Model

- Linear process
- Wikipedia:



Requirements

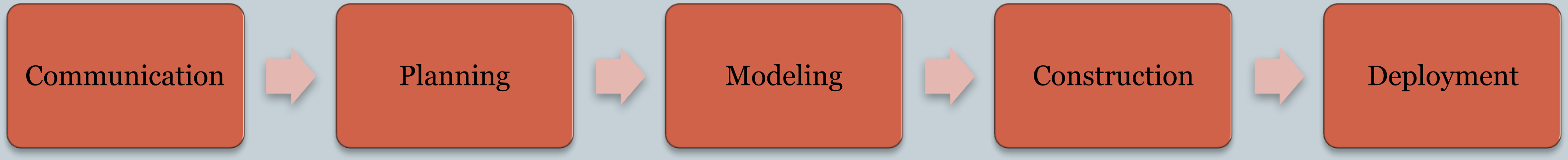
Design

Implementation

Verification

Maintenance

- Pressman book:



Communication

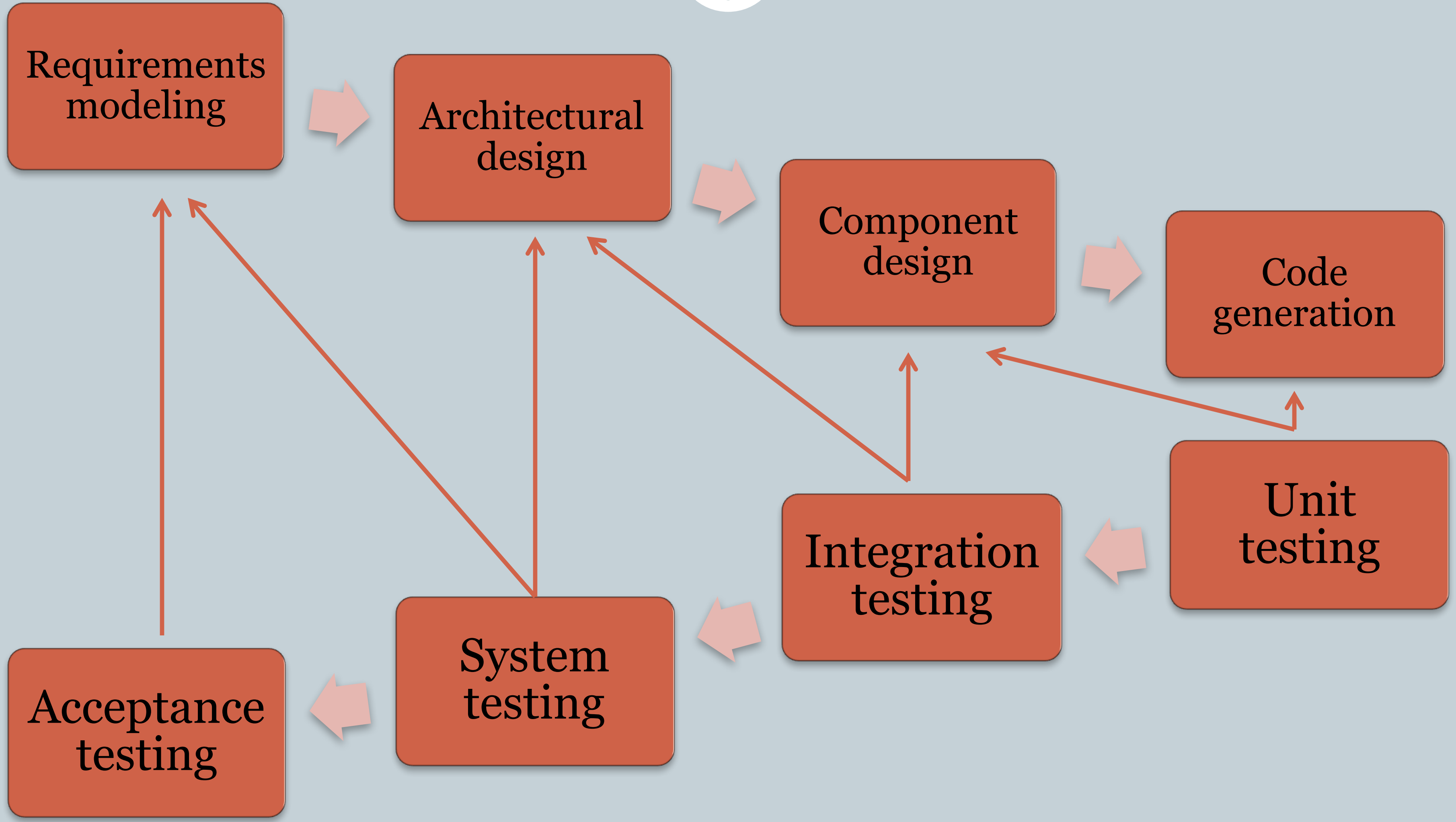
Planning

Modeling

Construction

Deployment

The V-Model


Requirements modeling
Architectural design
Component design
Code generation
Unit testing
Integration testing
System testing
Acceptance testing

Requirements Modeling

- A model of how your system is going to be used
- ATM example
  - User interaction
    - Take out money
      - (sub-steps)
    - Deposit money
      - (sub-steps)
  - Security control
    - Pulls commands from separate security modules which take priority and can shutdown the machine or prevent it from processing user commands

Architecture and Component Design

- Description of major components and how they work to achieve the requirements
- Architecture – define the components in the system and what they need to do
  - What are the components?
- Component design – define how the components are actually designed and how they do what they do
  - How do the components work?

Architecture and Component Design

- ATM example
  - User interaction
    - Card read component reads card from user
    - Keypad to get pin
    - Network connection to contact bank backend
  - Security control
    - Pulls commands from separate security modules which take priority and can shutdown the machine or prevent it from processing user commands
      - Anti-skimming module – prevents user from entering card into fake reader
      - Break-in/damage detection module – if system is damaged don't process user requests and send message to security administrator
      - Module that takes remote command from administrator/law enforcement who view situations through camera

Architecture and Component Design

- Remember we are still not at coding yet
- Identification of individual components not the only important thing
- Identifying interactions, interfaces, and dependencies between components
- Transition to the next step by thinking about how construction of these components can be split up among development team

Code Generation

- Write code to implement components
- In some cases auto-code generation from previously generated specifications
  - Computer-aided software engineering (CASE)
  - Build model of software in computer, auto-generate code from it
  - Considered by many not to be realistic in most software engineering scenarios

Unit Testing

- Remember that this is for testing “individual units” of source code. This could be functions, entire classes, or entire modules/components.
  - But should match up with the components defined in earlier phases
- Building a unit test
  - Set of inputs and expect outputs needed for tests
  - Unit tests not only testing the code to see if it crashes or not, should be testing to see if it matches the component specification
  - In this sense you are also testing the component specification too

Integration Testing

- “If each component works individually, why wouldn’t they work when you put them all together?”
- Communication between components (interfacing) is a point of potential problems
- Test the combination of components together, although it doesn’t necessarily have to be everything put together at once
  - Consider writing additional tests to just check how two specific components (that have already been tested individually with unit tests) work together
- By the end – everything together at once

System Testing

- If you’ve tested the components individually and pieced them together to make a whole and tested that too, then what’s left?

System Testing

- If you’ve tested the components individually and pieced them together to make a whole and tested that too, then what’s left?
- Let’s answer that question by introducing a few others:
  - How many different computers did you run your tests on?
  - How many different hardware variations did you have among such computers?
  - What was the state of the test machines with respect to operating system version, updated drivers, and other similar factors?

System Testing

- Still need to make sure that your app will actually work when it’s run outside of the pristine testing environment.
- Recovery testing
- Security testing
- Stress testing
- Performance testing
- Deployment testing

Recovery Testing

- Aspects of the system suddenly stop functioning – how does your software respond?
- If you have file IO read/write tests that worked fine in a test environment with admin. access and a non-full hard disk, what happens when the system denies access to files due to system settings/privileges?
  - If you fail to open a configuration file for reading, does your app stay running in a valid state or does it crash?
- Your app transfers data just fine over the internet when you have a valid connection. What happens when you pull the Ethernet cable out in the middle of a transfer?

System Testing

- Security testing (more on this in coming weeks)
- Stress testing
  - Your server software needs to be able to service 10 internet-based requests per second. What happens when you have 1000 requests incoming all at the same time?
  - The average project file for your app is 100 kb in memory. What happens when you open a project file that's 100 **M**b?
- Performance testing – measure runtime performance
- Deployment testing – install your app on some random desktop machine. Does it install and work correctly?

Acceptance Testing

- Now back to the beginning of your software process to ask the question: does the software match the original specification?
- Does your software do what it’s supposed to do for the user?
- This is likely black-box testing

Black-Box / Functional Testing

- Recall from an earlier lecture: black-box testing is testing without
  - knowledge of the inner workings of the software
  - access to the software’s source code
- Can still do this type of testing *with* code and automation, but test the app as though it’s being used by a person
- Make sure it does what it’s supposed to do for the user
  - “How it’s implemented” shouldn’t be relevant here

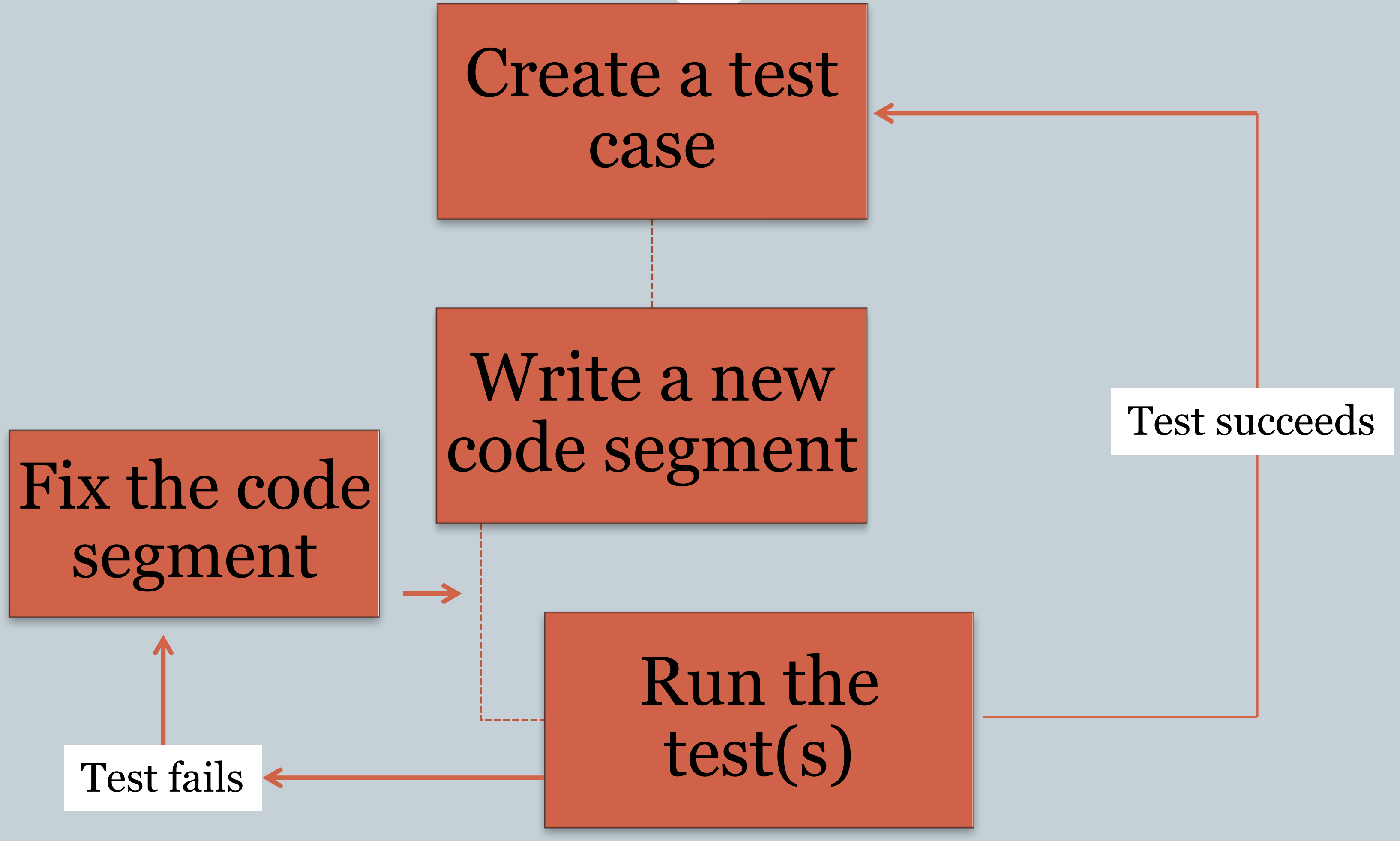
Stepping Away from the V-Model

- The previously mentioned stuff came from the V-model phases or were tangents off of some of those phases
- Many of those individual phases appear in other software development philosophies/methodologies. These other methodologies just may swap the order around or view some pieces as child or parents phases of the other.
- But now consider a software development philosophy that actually *starts* with testing.

Test-Driven Development

- In a nutshell: implement the tests before implementing the actual software components
  1. Implement test cases based on requirements
  2. Write code for a component that will satisfy the test
  3. If pass, move on to next test case, else fix the component code
- “No [product] code is written until a test exists to exercise it.”

Test-Driven Development



Create a test case

Write a new code segment

Run the test(s)

Test succeeds

Test fails

Fix the code segment

The Reality in Software Development

- Requirements can change a lot
- Thinking that you can just plan everything out on paper and execute that plan
  - Might work
  - Might not work
- Think about this: ever had a plan to build something, thought you had planned everything out and then in the middle of development hit a road block that you hadn't planned for?
- Maybe better to rephrase for this audience: ever thought you could finish a homework assignment the night before it's due only to find out you couldn't?
  - People are notoriously bad at planning

The Reality in Software Development

- Even if your plan was solid to implement a set of requested features, what if that set of features changes by customer request?
- Thinking of development as a single linear process with one final release at the end may not serve you well (even if you DO have one final release at the end)
- Shorter, repeated develop-and-release cycles may serve you better

Agile Development

- Manifesto for Agile Software Development written by notable software developers in 2001:
- “We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
  - Individuals and interactions over processes and tools
  - Working software over comprehensive documentation
  - Customer collaboration over contract negotiation
  - Responding to change over following a plan
- That is, while there is value in the items on the right, we value the items on the left more.”

Extreme Programming

- An agile process that uses the following development cycle (repeated over small amounts of time)

1. Plan
2. Design
3. Code
4. Test
5. if (reached incremental goal) then release
6. Goto 1

Extreme Programming

- A note on why we do a bunch of small weekly assignments to build one big project (web server)
- Often not effective (but done in plenty of other classes I'm sure) is this: work on it a lot and have nothing work each week throughout the semester, only get it working by the time you have to do a final demo/submission
- Better option: have incremental changes ensuring that the majority of time your code/project is actually at least partially working

Back to Agile: Agility Principles

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with preference to the shorter timescale.

Agility Principles

4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. **Working software is the primary measure of progress.**

Agility Principles

8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Which Development Philosophy to Choose?

- Up to you
  - (?)

Which Development Philosophy to Choose?

- Up to ~~you~~ your boss
- Different companies do things different ways and you'll likely just go along with whatever your employer dictates
- Worth noting that any of these philosophies could potentially work.
  - Like so many things it's not the plan itself but how well the plan is executed.
  - Would rather have you understand the philosophies and tradeoffs than just say "this one is the best" (although you have seen that I certainly do have my opinion)

References

- “Software Engineering: A Practitioner’s Approach” by Roger S. Pressman and Bruce R. Maxim
- Wikipedia
- http://msdn.microsoft.com/en-us/library/fda2bad5.aspx