Do I Trust It to Work?

CIS 322
Feb 21, 2017

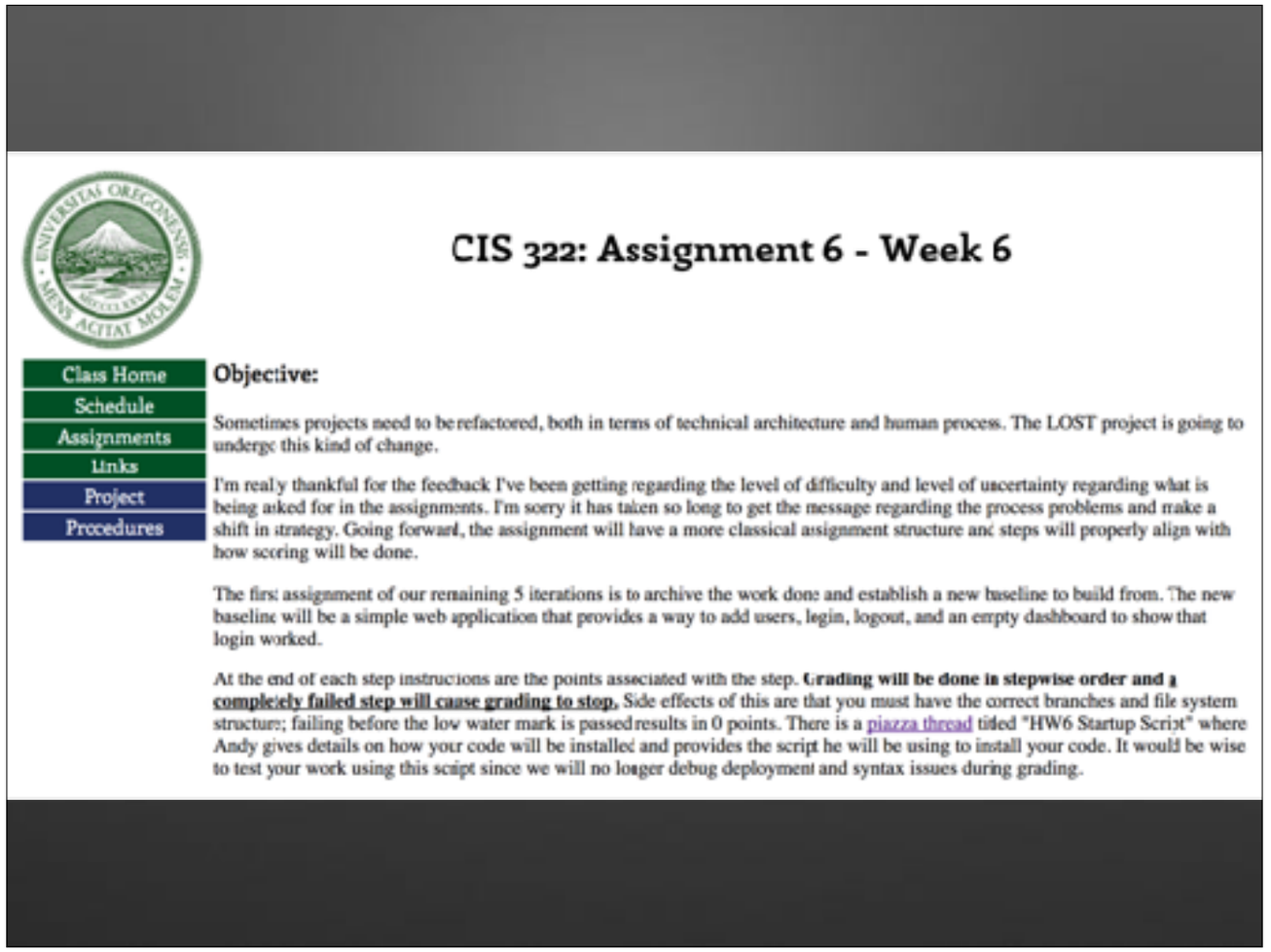
From the Reports

- I still hate all of the text editors.
- I made the assignment7 branch and technically went past the low water mark as my implementation has used apache since we started using it.
- Code all roughed out for assignment 7. Created and pushed new branch for assignment 7, updated and pushed create_tables.sql. Goal for tomorrow is to have steps 4 and 5 fully implemented.
- Off to a good start this week. Having it spelled out in steps takes some of the wrestling with approach and design structure out of each deliverable. I have completed up to Step 5, and should have the next two or three steps completed tomorrow.

Here are a few recent dailies.

The CLI text editors are a little uncomfortable… There are environments where that they are the only editor available. Fortunately, there are other file transfer options that can keep you from needing to work primarily from the shell.

There are a few students starting early. I’m really happy to see this and glad to see the questions via piazza for clarification early in the week.



I've posted my solution to assignment 6, you are free to look at how I approached the problem but do not copy and paste any of my code into your solution. Copied code will be a serious problem when grading the term project. All of your code (with the exception of the configuration parser) should be your own. If you would like to base your work on mine; become familiar with how I approach the problem. Then close all of my code and write your own solution with comments.

This week has been busy, so the grading is running a little bit late… One of the surprises we're encountering are solutions that do not appear to have been tested. This seems bizarre since we provided the test harness that is being used to checkout and install your code.

So, rather than talking about workflows today I'm pivoting the topic to testing.

Software/System Development Life Cycle

- Software and systems exist in space and time
  - Requirements change
  - Upstream/downstream systems change
- A popular visualization of SDLCs is the circle to the right.

Systems start life as some set of requirements, defining both a problem and what the shape of a solution would look like.
A system is then design to address the problem and match with the desired solution shape.
The design is implemented as an actual system.
Testing is done to verify that the system meets the requirements prior to deployment. Deploying a broken system is really expensive since it can cause the wrong decisions to be made or extensive rework to be done.
After deployment, the changing context and patches/fixes cause the system characteristics to change and other requirements to be identified… restarting the cycle.

After a system introduces a useful capability, the system can't just be turned off or replaced until there is some confidence that an acceptable replacement exists. This can lead to hack upon hack to keep systems going.

For more on SDLC and IT management, you might consider looking up ITIL.

Do I Invest In This Solution?

- What is the cost to keep doing what I'm doing?
  - Future capital and operations expenses
  - Liabilities incurred
- What is the cost if I use this new thing?
  - Comparative capital and operations expenses
  - Liabilities incurred
  - Retraining and business process changes
  - How much does it cost to try
  - How likely is failure
  - What might I lose

Both action and inaction carry a cost. To make an informed decision, a decision maker should aim to understand both the current state and the proposed future state.
When we change a system or introduce a new system we have the potential to make major impacts to the cost structure.
Liabilities include things like security risks and legal risks.

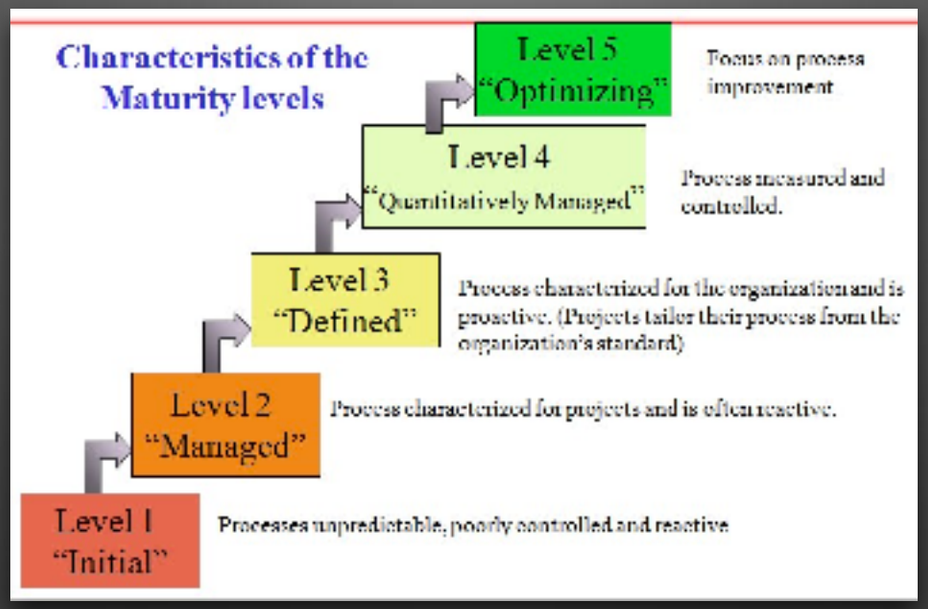
We have to overcome the uncertainty somehow… The new solution has to be enough better than what is being done to pay for itself.

Controlling Perceived Risk

- Has this ever been done before? Is it possible?
- Do we agree an what we are trying to do?
- How do we know we are making progress?
- Can we tell the difference between a working and nonworking solution?
- Do we know how it might fail? Do we have a plan for that?

https://www.flickr.com/photos/johnbullas/7788230372

Past performance
Requirements documentation, story boards, etc
Project timelines, milestone deliveries
Testing, independent verification and validation
Continuity of operations plan, disaster recovery plan



CMMI

Capability Maturity Model Integration

CMMI is a business process improvement training and appraisal program that was developed by at Carnegie Mellon University. If you get into government contracting they will care about which CMMI level your organization's projects have been appraised to.

At the lowest level, there is no defined project process. This is a huge risk since there is no visibility into wether the delivered solution will have any connection to the requirements… or even if there were requirements. Without process around development there is little ability to identify that problems are occurring or respond to them.

Moving to higher levels increases the amount of project management overhead. Documentation is more detailed and traceability between documents and project activities are required. Higher CMMI levels are expected to have lower risk since there is strong documented correspondence between documents and deliverables. Failure to specify the correct system or an inability to assemble the technical solution still cause projects to fail.

Most of the primes I worked for only achieved CMMI level 3 and only for a few of their projects.

Let's talk a bit about testing…

Computer Science

- Science is about testable hypotheses… Do you apply science to your computing?
- Test ability
  - What is needed to prove my hypotheses are true?
  - What is needed to prove my hypotheses are false?
- Tests are experiments
- For an engineered system, the outcome of every test should be known in advance

https://commons.wikimedia.org/wiki/File:Mad_scientist.svg

Proof of Correctness

- Use mathematics to reason about the solution
  - Requirements become invariants and assumptions
  - The algorithm is analyzed as a mathematical object
- Proof of correctness is extremely powerful but doesn't fit reality well…
  - Real machine violates the execution model
  - Implementation may diverge from algorithm model
  - Requirements are incompletely specified
  - State space explosion makes analysis to complex

Everybody likes CIS 313 and 315, right?

Being able to complete proof of correctness and analyze asymptotic runtime well are super important skills and extremely helpful for building good systems. Just be aware that for any sufficiently large or complex system this tool breaks.

Spot Checking

- All software testing comes down to spot checking
  - Try the system with a known input
  - Does the expected output occur
- Only proves the given test case is working as expected
  - Testing all inputs is not tractable
  - Testing cannot prove correctness
  - Mostly good at finding known unknown faults
- Effective testing involves
  - Partitioning the search space effectively
  - Applying automation where it is cost effective

Debugging Is Testing

- We don’t set out to write broken code, this code should work
- Each execution of a program is an experiment; we can’t know in advance for arbitrary programs if they will work correctly.
- A defect/bug indicates a belief we had about the way our program would behave is wrong
- A hypothesis regarding the nature of the defect leads to an experiment
  - Conduct an experiment to get more information of the system state
  - Conduct an experiment to avoid the error
- Rerun the experiment

Where Do I Get Test Cases From?

- System requirements
- Defects that have been discovered
- Boundary conditions in the implementation
- Branching conditions in the implementation
- Component boundaries
- Security guidance

The first place to go is to a list of the requirement. Ideally, each requirement is enumerated somewhere and you can write a test to verify that each requirement is met.
As problems/defects in the code are discovered, a test should be introduced to verify that the defect has not reemerged.
As you write the code, you should be able to identify things that will cause it to break. Assertions can also be useful for protecting against these.
Each branch in the code create a new possible path through the program. Ideal test coverage would use all paths.
Component boundaries are places where subsystems might we swapped out later and are common places where systems break. Does my output match the expectation of my downstream? What happens if my upstream gives me the wrong thing?
An organization with a security organization will also provide guidance. These will generally appear as requirements for the system and questionnaires to explain why your software does not have specific vulnerabilities.

Requirements Traceability

- In well specified systems, the requirements define what the system will and won't do
- Good requirements should be testable
  - What is a test that could verify the requirement is satisfied?
  - What steps are needed to conduct the test?
  - What output shows the test passed/failed?
- Mapping requirements to tests supports test driven development, passing tests indicates quantifiable progress towards completion

Test Plans

- Capture and organize a series of tests
- Generally a document to be used by non-developers
- Ideal exercises all of the intended system functionality
- Each test sequence is a test case
  - Should map to one or more requirements
  - Should include preconditions if any
  - Should document the specific steps to be done
  - Should provide the expected result of each step

Let’s try to build a test plan for assignment 7