



6.170 Lecture 21

Project Management and Control



John Guttag
MIT EECS



Effort Is Not the Same as Progress


Cost is product of workers and time
Easy to track

Progress is more complicated
Hard to track

People don't like to admit lack of progress
Think they can catch up before anyone notices
Not usually possible

Design process and architecture to facilitate tracking


©Srinivas Devadas/John Guttag Spring 2002 Slide 2



How Does a Project Get to Be One Year Late?

One day at a time
It's not the hurricanes that get you
It's the termites
Tom missed a meeting
Mary's keyboard broke
The compiler wasn't updated
...

©Srinivas Devadas/John Guttag Spring 2002 Slide 3




How Does a Project Get to Be One Year Late?

One day at a time
It's not the hurricanes that get you
It's the termites
Tom missed a meeting
Mary's keyboard broke
The compiler wasn't updated
...

Remember, "It ain't over 'till it's over."
If you find yourself ahead of schedule
Don't relax
Don't add features

©Srinivas Devadas/John Guttag Spring 2002 Slide 4




Controlling Schedule

First, you must have one
Need verifiable milestones
Some non-verifiable milestones
90% of coding done
90% of debugging done
Design complete

Need 100% events
Module 100% coded
Unit testing successfully complete

Need critical path chart
Know effects of slippage
Know what to work on when

©Srinivas Devadas/John Guttag Spring 2002 Slide 5



Getting to the End

Rule of thumb for complex projects
1/3 planning (not all up front)
1/6 coding
1/4 component test and early system test
1/4 system test

When is the project over?

The project is done when
It is in users' hands
Significant fraction of resources freed up

©Srinivas Devadas/John Guttag Spring 2002 Slide 6

6.170 Dealing with Slippage

People must be held accountable
Slippage is not inevitable
Software should be on time, on budget, and on function

Four options
Add people -- rarely works
Buy components -- hard in mid-stream
Change deliverables
Change schedule

Take no small slips
One big adjustment far better than three small ones

©Srinivas Devadas/John Guttag Spring 2002 Slide 7

6.170 Temptations to Avoid

Avoid featuritis
Costs under-estimated
Effects of scale discounted
Benefits over-estimated
Swiss army knife rarely the right tool

Avoid digressions, for example,
Infrastructure
Premature tuning
Often addresses the wrong problem

Avoid quantum leaps
Occasionally great leaps forward
More often, into the abyss

©Srinivas Devadas/John Guttag Spring 2002 Slide 8

6.170 A Few More Hints

Incorporate existing software
Legacies can be good

Buy rather than build
Last year's model if possible
The Devil you know is better than the Devil you don't

Start from where you are
Clean slate an insurmountable opportunity

Reusable components should be a design goal
Mission not same as project
Build organization as well as project
Software is capital
Will not happen by accident
Need suitable reward structure

©Srinivas Devadas/John Guttag Spring 2002 Slide 9

6.170 You Have a Design, Now What?

Need to code and test the system
Key question, what to do when?
Assume incremental development model

Suppose piece of system has module dependency diagram

```
graph TD; A --> B; A --> C; A --> D; B --> E; C --> F; D --> G; E --> F; F --> G;
```

©Srinivas Devadas/John Guttag Spring 2002 Slide 10

6.170 Bottom-Up Implementation

Before implement/testing any module
Implement/test children, e.g., G, E, B, F, C, D, A

G & E tested stand-alone
Generate test data as discussed earlier
Construct drivers

Next, implement/test B, C, F, D
No longer unit testing, use lower-level modules

At each level we are testing
Whether module being tested works
Whether modules it calls behave as expected

When an error surfaces, many possible sources
Integration testing hard, irrespective of order

```
graph TD; A --> B; A --> C; A --> D; B --> E; C --> F; D --> G; E --> F; F --> G;
```


©Srinivas Devadas/John Guttag Spring 2002 Slide 11

6.170 Building Drivers

Use a person, simplest choice
Also worst choice
Errors in entering data inevitable
Errors in checking results inevitable
Tests not easily reproducible
Problem for debugging
Problem for regression testing
Test sets stay small, don't grow over time
Testing cannot be done as a background task

Incorporate drivers into automated test harness

©Srinivas Devadas/John Guttag Spring 2002 Slide 12

 **Test Harnesses**

Goals
Increase amount of testing over time
Facilitate regression testing
Reduce human time spent on testing

Take input from a file


Call module being tested

Save results (if possible)
Including performance information

Check results
At best, if correct
At worst, same as last time

Generate reports


©Srinivas Devadas/John Guttag Spring 2002 Slide 13

 **Regression Testing**

When a change is made
Make sure that things that used to work still do
Including performance

Knowing exactly when a bug is introduced important
Keep old test results
Keep versions of code that match those results
Storage is cheap

©Srinivas Devadas/John Guttag Spring 2002 Slide 14

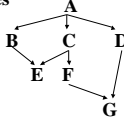
 **Top-down Testing**

Before implement/test, test all using modules
Here, must start with A


How do we run A?
Build stubs to simulate B, C, & D

Next, choose a successor module, e.g., B
Build stub for E
Drive B using A

Suppose C is next
Can we reuse stub for E?
Maybe



©Srinivas Devadas/John Guttag Spring 2002 Slide 15

 **Building Stubs**


Query a person at a console
Same drawbacks as using person as driver

Write message describing call
Name of procedure and arguments
Fine if calling program does not need result
More often than you might think

Provide canned or generated sequence of results
Very often sufficient
Generate using criteria used to generate data for unit test
May need different stubs for different callers

Provide a primitive (inefficient & incomplete) impl.
Best choice, if not too much work
Look-up table often works

©Srinivas Devadas/John Guttag Spring 2002 Slide 16

 **One Good Way to Structure Implementation**

Largely top-down
But always unit test modules

Bottom-up
When stubs too much work
Low level module that is used in lots of places
Low-level performance concerns

Depth-first, visible-first
Allows interaction with customers, like prototyping
Lowers risk of having nothing useful
Morale of customers and programmers improved
Needn't explain how much invisible work done
Better understanding of where the project is
Don't have integration hanging over heads

©Srinivas Devadas/John Guttag Spring 2002 Slide 17