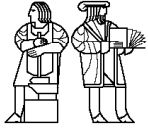


6.170 Lecture 21 Project Management and Control, part 3



John Guttag
MIT EECS

6.170 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

©Michael Ernst/John Guttag Spring 2001 Slide 2

6.170 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

©Michael Ernst/John Guttag Spring 2001 Slide 3

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

©Michael Ernst/John Guttag Spring 2001 Slide 4

6.170 Why Adding People Fails


Worker and months are not necessarily interchangeable
Start up transient for new people considerable
Unless tasks can be cleanly partitioned
What architecture is all about

©Michael Ernst/John Guttag Spring 2001 Slide 5

6.170 Architecture

An architecture describes a partitioning of the system s.t.
Work can proceed in parallel
Progress can be closely monitored
The parts combine to provide the desired functionality


©Michael Ernst/John Guttag Spring 2001 Slide 6

 **Why Parallel Activities**

Serial development requires too much elapsed time
Sacrifice efficiency for reduced latency


Ramifications
Harder to learn from mistakes
Things may not fit together
Testing a challenge

©Michael Ernst/John Guttag Spring 2001 Slide 7

 **Good Architecture Allows**

Adding and changing features
Integration of acquired components
Communication with other software
Easy customization
Ideally with no programming
Turning users into programmers is good
Software to be embedded within a larger system
Recovery from wrong decisions
About technology & markets

©Michael Ernst/John Guttag Spring 2001 Slide 8

 **Systems Architecture**

Have one and subject it to serious scrutiny
At relatively high level of abstraction
Basically lays down communication protocols


Simple is good

Flat is good
Why?

Design to accommodate changes to design
Software is malleable only by design
Corrections and changing requirements
Incrementalism not an excuse for non-design

But, know when to say NO
A good architecture rules things out

©Michael Ernst/John Guttag Spring 2001 Slide 9


 **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

©Michael Ernst/John Guttag Spring 2001 Slide 10

 **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

©Michael Ernst/John Guttag Spring 2001 Slide 11

 **Final Project Demo**

Gizmoball!

©Michael Ernst/John Guttag Spring 2001 Slide 12