



Lecture 19

Software Project Management II



MIT EECS


MIT 6.170 Slide 1



Outline

- Build tools and Source Code Control**
- Scheduling
- Architecture
- Bug Tracking
- Implementation and testing order
- Anatomy of a Product Cycle

MIT 6.170 Slide 2



Build Tools

Modern software is very complex

Many different components are required

Example: Java compiler, C compiler, GUI builder, Device driver build tool, Installshield, Web server, Database, scripting language for build automation, parser generator, test generator, test harness

System may run on multiple devices


Each has its own build tools

Everyone needs to have the same tool set!

Wrong, missing tool can drastically hinder productivity

Hard to switch tools in mid project.

MIT 6.170 Slide 3



Source Code Control

The central repository of the code

Helps in...

- Collecting code/bug-fixes from multiple team members
- Synchronizing all the team members to current source
- Let multiple teams make progress in parallel
- Manage multiple versions, releases of the software
- Help identify regressions

Again, good tools exit

CVS, RCS


However, policies are even more important

- When to check-in, when to sync, how builds are done and when to branch and merge
- Policies need to be changed to match the state of the project

Surprisingly big issue

A large time sync in even medium sized projects

MIT 6.170 Slide 4



Scheduling

“More software projects have gone awry for lack of calendar time than for all other causes combined.”

-- Fred Brooks, *The Mythical Man-Month*


Three central questions of the software business

- 3) When will it be done?
- 2) How much will it cost?
- 1) When will it be done?

Facts

1. Estimates almost always too optimistic
2. Estimates reflect what one wishes to be true
3. We confuse effort with progress
4. Progress is poorly monitored
5. Slippage is not aggressively treated

MIT 6.170 Slide 5



Scheduling Is Crucial

Usually gets far less attention than appropriate

- Made to fit other constraints

Needed to make slippage visible

- Like a quarterly business plan
- Must be objectively checkable by outsiders

Unrealistically optimistic schedules are a disaster


- Decisions get made at the wrong time
- Decisions get made by the wrong people
- Decisions get made for the wrong reasons

The great scheduling paradox

Everything takes twice as long as you think

*Even if you **know** that it will take twice as long as you think*

MIT 6.170 Slide 6




Milestones

Milestones are critical keep the project on track
Major milestones should change many policies around
Check-in rules, build process etc.

Some typical milestones
Design complete
Interfaces complete / Feature complete
Code complete / Code freeze
Alpha release
Beta release
FCS (First Commercial Shipment) release

MIT 6.170 Slide 7




Estimates in Large Projects

Estimates don't change as activity approaches
No matter how wrong they end up being

Once activity has started
Overestimates of cost come steadily down
Underestimates do not change until near scheduled end

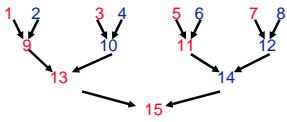
MIT 6.170 Slide 8



Optimism is the Root of Problem

People assume that all will go well
Every task will take as long as it ought to take


Consider the following schedule (arrows indicate dependences)



Suppose that on average each task takes as long as planned
Odd tasks over-run by 10 days, even under-run by 10

How close to schedule does the project finish?

MIT 6.170 Slide 9




Estimating With One's Heart

Desires of client
Can dictate scheduled completion date
Cannot dictate the actual completion date

Don't let client push you into an unrealistic plan
Have the courage to trust pessimistic estimates
This is not easy!
"Madame No" sometimes gets fired

Evaluate your team honestly
Remember that productivity differs greatly

MIT 6.170 Slide 10



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

MIT 6.170 Slide 11



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 'til it's over."
If you find yourself ahead of schedule
Don't relax
Don't add features

MIT 6.170 Slide 12




Controlling the 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

MIT 6.170 Slide 13



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?
Never?

The project is done when

- It is in users' hands
- A significant fraction of resources are freed up

MIT 6.170 Slide 14



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 – there is a startup cost
- Buy components – hard in mid-stream
- Change deliverables
- Change schedule

Take no small slips

- One big adjustment is far better than three small ones

MIT 6.170 Slide 15



Architecture


An architecture describes a partitioning of the system

It indicates dependences on, and data flow between, modules

A good architecture ensures that

- Work can proceed in parallel
- Progress can be closely monitored
- The parts combine to provide the desired functionality


MIT 6.170 Slide 16



A 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
 - About markets

MIT 6.170 Slide 17



System Architecture

- Have one and subject it to serious scrutiny**
 - At relatively high level of abstraction
 - Basically lays down communication protocols
- Reusable components should be a design goal**
 - Organizational mission is not the same as the project
 - Build your organization as well as the project
 - Software is capital
 - This will not happen by accident
- Simple is good**
- Flat is good**
- Know when to say no**
 - A good architecture rules things out

MIT 6.170 Slide 18

6.170 Temptations to Avoid

Avoid featuritis

- Costs under-estimated
- Effects of scale discounted
- Benefits over-estimated
- A swiss army knife is rarely the right tool

Avoid digressions

- Infrastructure
- Premature tuning
- Often addresses the wrong problem

Avoid quantum leaps

- Occasionally, great leaps forward
- More often, into the abyss

MIT 6.170 Slide 19


6.170 Bug Tracking

Helps in...

- Tracking and fixing bugs
- Identifying problem areas and managing tem
- Communicating between team members
- Track regressions and repeated bugs


Any medium to large size project requires bug tracking software

Good tools exist



MIT 6.170 Slide 20

6.170 Bug Tracking



MIT 6.170 Slide 21

6.170 Bug Tracking

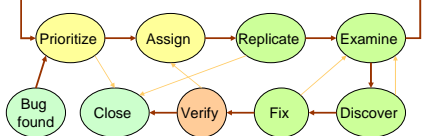
Bugs are different from features!

Need to configure the bug tracking system to match the project

- Many make the system too complex to be useful

A good process is key to managing bugs

- Need an explicit policy that everyone knows and follows
- Keep tweaking it as needed



MIT 6.170 Slide 22

6.170 How to code and test your design

You have a design and architecture

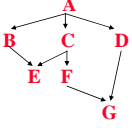
- Need to code and test the system

Key question, what to do when?

- We'll assume an incremental development model

Suppose the system has this module dependency diagram

- In what order should you address the pieces?



MIT 6.170 Slide 23

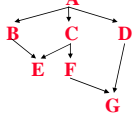
6.170 Bottom-Up Implementation

Before implementing/testing any module

- implement/test its children
- For example: G, E, B, F, C, D, A

G & E are tested stand-alone

- Generate test data as discussed earlier
- Construct drivers



Next, implement/test B, F, C, 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 is hard, irrespective of order

MIT 6.170 Slide 24

Building Drivers

Use a person

- Simplest** choice, but also **worst** choice
- Errors in entering data are inevitable
- Errors in checking results are 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

Better alternative: Automated drivers in a test harness

MIT 6.170 Slide 25

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, is correct
- At worst, same as last time

Generate reports

MIT 6.170 Slide 26

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 is important

- Keep old test results
- Keep versions of code that match those results
- Storage is cheap

MIT 6.170 Slide 27

Top-down Testing

Before implement/test, test all clients (using modules)

Here, we start with A

How do we run A?

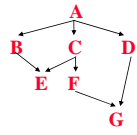
- Build stubs to simulate B, C, & D

Next, choose a successor module, e.g., B

- Build stub for E
- Drive B using A

Suppose C is next

- Can we reuse the stub for E?



MIT 6.170 Slide 28

Implementing a Stub

Query a person at a console

- Same drawbacks as using a person as a driver

Print a message describing the call

- Name of procedure and arguments
- Fine if calling program does not need result
- More common 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) implementation

- Best choice, if not too much work
- Look-up table often works

MIT 6.170 Slide 29

Comparing Top-down and Bottom-up


Criteria (next 5 slides)

- What kinds of errors are caught when?
- How much integration is done at a time?
- Distribution of testing time?
- Amount of work?
- What is working when (during the process)?

Neither dominates

- Useful to understand advantages/disadvantages of each
- Helps you to design an appropriate mixed strategy

MIT 6.170 Slide 30




Catching Errors

Top-down tests global decisions first
 E.g., what system does
 Most devastating place to be wrong
 Good to find early

Bottom-up uncovers efficiency problems earlier
 Constraints often propagate downward
 You may discover they can't be met at lower levels

MIT 6.170 Slide 31




Amount of Integration at Each Step

Less is better

Top-down adds one module at a time
 When error detected either
 Lower-level module doesn't meet specification
 Higher-level module tested with bad stub

Bottom-up adds one module at a time
 Connect it to multiple modules
 Thus integrating more modules at each step
 More places to look for error

MIT 6.170 Slide 32




Distribution of Testing Time

Integration is what takes the time

Bottom-up gets harder as you proceed
 You may have tested 90% of code
 But you still have far more than 10% of the work left
 Makes prediction difficult

Top-down more evenly distributed
 Better predictions
 Uses more machine time
 In business environments this can be an issue

MIT 6.170 Slide 33



Amount of Work

Always need test harness


Top-down
 Build stubs but not drivers

Bottom-up
 Build drivers but not stubs

Stubs usually more work than drivers
 Particularly true for data abstractions

On average, top-down requires more non-deliverable code
 Not necessarily bad

MIT 6.170 Slide 34




What Components Work During the Development Process

Bottom-up involves lots of invisible activity
 90% of code written and debugged
 Yet little that can be demonstrated

Top-down depth-first
 Earlier completion of useful partial versions

MIT 6.170 Slide 35



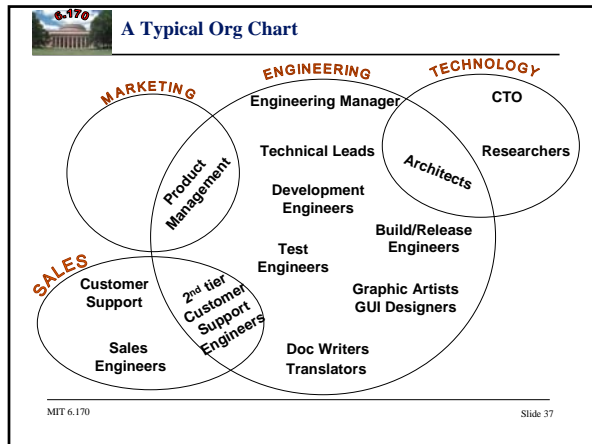
One Good Way to Structure an Implementation

Largely top-down
 But always unit test modules

Bottom-up
 When stubs are 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 your head

MIT 6.170 Slide 36



Conception phase

Product Management

- Analysis of the company's strengths
- Analysis of the current product line and trends
- Customer Survey
- Market Survey
- Technology ideas from the CTO office

Product Idea

Marketing Requirement Document (MRD)

MIT 6.170 Slide 38

Planning phase

Architect

- Take the MRD
- Figure-out how to divide the work into components
- Identify possible hard/showstopper problems
- Get a researcher/engineer to do a Proof-of-Concept (PoC)
- Identify the high-level tasks

Development Leads for each Component

- Define the tasks and project the work required
- Define the interfaces to the rest of the components

Test Lead

- Create a test plan

Engineering Manager

- Assign Resources / People
- Identify a timeline and a release plan

→ Leads to an Engineering Requirements Document (ERD)

MIT 6.170 Slide 39

Development Phase

Multiple teams work in parallel.

...and everyone attend a lot of meetings

- Project status meeting
 - All hands on a small project
 - team leaders on a large one
- Component interface design/review meetings
 - Architects, developers and testers involved
- User interface, workflow design/review meetings
 - Product mgmt., developers, testers, graphics designers, tech writers
- Code reviews
 - A few members of the team
- Defect tracking/triage meeting
 - Project manager, test manager and a few (very unhappy) developers
- Release management meetings
 - Project manager, test manager, dev. leads, release lead, product mgmt.

MIT 6.170 Slide 40

Release Engineering Phase

Feature Freeze

- Stop adding features
- Stop changes to the interfaces without mgmt signoff
- Increase the bar for check-ins
- Start scrubbing the bug database

Code Freeze

- Mainly testing activity
- Bug triage and critical bug fixes
- Very stringent check-in rules

Beta Release

- Sign-off from product mgmt. and the test manager

Post Beta

- Only critical/showstopper bug fixes
- Continuous testing

FCS (First Commercial Shipment)

MIT 6.170 Slide 41