Software Development Best Practices

Part 2

Outsourcing

- Paying an outside organization to develop a project or parts of a project instead of developing it in-house
- Presumably the outsourcing organization has more expertise in the particular application area
- Can potentially save development cost and time

Outsourcing Benefits

- Reuse
  - Commercial outsourcing companies can achieve economies of scale where an individual organization cannot
- Staffing flexibility
  - Outsourcing organization might be able to devote more developers
- Experience
  - Presumably has more experience if the area is new to you
- Better requirements specification
  - Forces careful requirements in order to craft contract than otherwise may be developed
- Reduced feature creep
  - Since paying for functions and need specific requirements, feature creep can be controlled

Using Outsourcing

- Requires more skillful management
  - Develop a management plan including risk management
    - How to select a vendor
    - Negotiate contract
    - Develop requirements
    - Handle requirements changes
    - Track vendor progress
    - Monitor quality
    - Validate software meets requirements
  - Make communication with the vendor a priority
    - Loss of visibility a high risk
  - Will still need to use some of your own technical resources

Offshore Outsourcing

- Offshore companies offer considerably lower costs – could be 35% or more
- Consider communication challenges
  - Time issues, Language issues
- Language issues
  - Problem if code documented in Russian or Chinese?
- Travel expenses

Outsourcing Summary

- Efficacy
  - Potential reduction from nominal schedule: Excellent
  - Improvement in progress visibility: None
  - Effect on schedule risk: Increased Risk
  - Chance of first-time success: Good
  - Chance of long-term success: Very Good
- Major Risks
  - Transfer of expertise outside the organization
  - Loss of control over future development
  - Compromise of confidential information
  - Loss of progress visibility and control
- Major Interactions
  - Tradeoff between control/visibility for development speed

Productivity Environments

- Creating an environment that fosters productivity
  - Wrong environment can prevent the extraction of working software from the brains of developers
  - Flow Time
    - A “flow state” is a state of total immersion in a problem that facilitates understanding and generation of solutions
    - DeMarco 1987 : Developers need 15 minutes or more to enter a state of flow, can’t be constantly interrupted
  - Hygiene Factors
    - Inadequate office facilities can seriously erode motivation and productivity
    - More than adequate facilities does not increase motivation and productivity

Using Productivity Environments

- At least 80 square feet of floor space per developer
- At least 15 square feet of desk space capable of holding books
- Some means of stopping phone interruptions
- Some means of stopping in-person interruptions
- Some means of shutting out unwanted noise
- At least 15 feet of bookshelf space
- View of external window
- Access to whiteboard, bulletin board space
- Convenient access to
  - team members, printer, copy machine, conference room, common office supplies

Logitech Study

- Survey of 1003 US office workers
- Rated office as “C+”
- 46 percent of women and 32 percent of men said their emotional state was closely tied to the condition of their workspace
- 7 percent said their desk was a safety hazard
- 6 percent were embarrassed by their space
- 9 percent wouldn't want their mother to see where they work

- Lack of privacy was the top annoyance cited by those surveyed. Other irksome features mentioned by many included "not enough shelves to put things", "no window" and "too much clutter."

Programmer Competition Results

Factor	**Top 25%**	**Bottom 25%**
Dedicated floor space	78 sq ft	46 sq ft
Acceptably quiet	57% yes	29% yes
Acceptably private	62% yes	19% yes
Silenceable phone	52% yes	10% yes
Calls can be diverted to voicemail or other person	76% yes	19% yes
Frequent needless interruptions	38% yes	76% yes
Workspace makes developers feel appreciated	57% yes	29% yes

Productivity Environments Summary

- Efficacy
  - Potential reduction from nominal schedule: Good
  - Improvement in progress visibility: None
  - Effect on schedule risk: None
  - Chance of first-time success: Good
  - Chance of long-term success: Very Good
- Major Risks
  - Status-oriented office improvements instead of productivity-oriented improvements
  - Transition downtime
  - Political repercussions of preferential treatment
- Major Interactions
  - Trades small increase in cost for large increase in productivity

Rapid Development Languages (RDL)

- “Power Tools” for developers
- If building a dog house, it will probably be much faster to use a power saw, belt sander, paint sprayer, nail gun, etc. than hand tools
  - But higher chance of going to the hospital
  - More intricate quality can be performed by hand tools
- Examples
  - Visual Basic, Delphi, Microsoft Access, DreamWeaver
  - Allow developer to code at a higher level of abstraction than they could with traditional languages

Approximations

- Size in Lines of Code

Function Points	Fortran	Cobol	C	C++	Pascal	VB
1	110	90	125	50	90	30
100	11,000	9,000	12,500	5,000	9,000	3,000
500	55,000	45,000	62,500	25,000	45,000	15,000
1,000	110,000	90,000	125,000	50,000	90,000	30,000
5,000	550,000	450,000	625,000	250,000	450,000	150,000

Managing Risks of RDL's

- Silver bullet syndrome
  - Unlikely using a new language will reduce end-to-end time by 25% as vendor may claim
  - Classic mistake to overestimate savings
- RDL not suited to some projects
  - May not have functionality, require too much setup, etc.
- Failure to scale up to large projects
  - RDL's frequently lack features to support large projects
  - Same features that are convenient on small projects can cause problems on large ones
    - Weak data typing
    - Poor support for modularity
    - Weak debugging
    - Weak ability to call routines in other languages
- May encourage sloppy programming practices
  - Doesn't mean you don't have to design anymore

RDL Summary

- Efficacy
  - Potential reduction from nominal schedule: Good
  - Improvement in progress visibility: None
  - Effect on schedule risk: Increased Risk
  - Chance of first-time success: Good
  - Chance of long-term success: Very Good
- Major Risks
  - Silver-bullet syndrome and overestimated savings
  - Failure to scale up to large projects
  - Encouragement of sloppy programming practices
- Major Tradeoffs
  - Trades some design and implementation flexibility for reduced implementation time

Requirements Scrubbing

- Requirements specifications drawn up
  - Minimal specification seeks minimum requirements
- Requirements scrubbing
  - Carefully examine specs for unnecessary or overly complex requirements, which are then removed
  - Product size the largest contributor to project's cost and duration – by eliminating these requirements the schedule is shortened

Requirements Scrubbing Summary

- Efficacy
  - Potential reduction from nominal schedule: Very Good
  - Improvement in progress visibility: None
  - Effect on schedule risk: Decreased Risk
  - Chance of first-time success: Very Good
  - Chance of long-term success: Excellent
- Major Risks
  - Elimination of requirements that are later reinstated

Reuse

- Planned Reuse
  - Long-term strategy to build a library of frequently used components
  - Allows new programs to be assembled quickly from existing components, e.g. ActiveX Controls
- Opportunistic Reuse
  - Could be used opportunistically as a short-term practice by salvaging code for a new program from existing programs
  - Less savings than long-term planned reuse
- Can also apply to designs, data, documentation, specs, plans, etc.

Opportunistic Reuse

- Opportunity arises if you discover an existing system has something in common with a new system to build
- Adapt or Salvage?
  - Adapt old system to the new one
  - Design new system from scratch but salvage components from the old one
  - Usually Salvage works best – requires you to understand only small pieces of the old program in isolation

Opportunistic Reuse

- Overestimated Savings
  - Easy to overestimate potential effort and schedule savings
  - Takes time to figure out what can be reused
  - Takes time to modify old parts to fit into the new
- Experiences
  - French military: 37% improvement in productivity via reuse
    - Credited success to information hiding, modularity
  - NASA
    - 35% code salvage using functional design
    - 70% code salvage using OO based design
  - Can be done at individual developer level, not managerial

Planned Reuse

- Doesn't help on first project, but should on subsequent ones
- Requires more planning
  - Survey software to identify components that occur frequently
  - Generally requires survey outside own small group, but across many groups or whole organization
  - Needs management commitment, long-term commitment to succeed
  - Measure productivity to see if it is paying off
  - May require evaluation of architectures being used

Planned Reuse

- Focus on domain-specific components
  - E.g. reusable financial component, file-transfer component, messaging component
- Create small, sharp components
  - Easier to use than large, bulky, general components
- Focus on information hiding, encapsulation
- Focus on quality not size

Reuse Risks

- Wasted Effort
  - Creating reusable components costs 2-3 times as much as creating a 1-off component
  - Wasted effort if it is not reused, ideally three times
    - If not going to be reused three times, might not be worth the effort
    - Might even make a 1-off first, then if it comes up again make the reusable component
- Shifting Technology
  - If technology changes before it can be reused, it will probably not be reused
- Overestimated Savings
  - Reuse savings generally overestimated; still other costs to write code, modify, understand
- Bugs
  - Bugs in a reused component proliferate the problem
  - Bug might not appear in original project, but appear in new project

Reuse Summary

- Efficacy
  - Potential reduction from nominal schedule: Excellent
  - Improvement in progress visibility: None
  - Effect on schedule risk: Decreased Risk
  - Chance of first-time success: Poor
  - Chance of long-term success: Very Good
- Major Risks
  - Wasted effort if the components prepared for reuse are not selected carefully
- Major Interactions
  - Coordinate with using productivity tools
  - Must have foundation of S/W development fundamentals

Signing Up

- Technique that can lead to extraordinary levels of motivation
- Shackleton's advertisement for explorers:
  - *MEN WANTED for Hazardous Journey, Small Wages, Bitter Cold, Long Months of Complete Darkness, Constant Danger, Safe Return Doubtful, Honor and Recognition in Case of Success*
  - Drew 5000 applications from which 27 were selected
- Leader or manager asks potential team members to "sign up" to make a commitment to seeing the project through to success

Using Signing Up

- Frame a challenge and a vision
  - Key to motivation is a clear vision and extraordinary accomplishment
  - Project completion alone not enough
  - Ex:
    - First to put an astronaut on the moon
    - Design and build a totally new piece of software
    - Be the first team in the organization to develop a complete product in 8 months
    - Create a package that places #1 in PC Magazine Rankings

Using Signing Up

- Give people a choice
  - Doesn't work if people don't have a choice of whether or not to sign up
  - Can limit pool
  - Must be done up-front at start of project or upon coming across a crisis, doesn't work in the middle of a project
- Small teams
  - Works best with small teams with identity, not at the level of a large organization

Unequivocal Commitment

- Members must commit to get the job done no matter what
- Kerr's report
  - Team focused 8 hour day on project only, sweeping aside normal responsibilities
  - At high point, worked until midnight with a half-hour break for pizza and beer
- Microsoft Windows NT
  - Meant foregoing everything: evenings, weekends, holidays, normal sleeping hours
  - When not sleeping, were working
  - One team member answered email from the hospital while his wife was in labor
  - Cots kept in offices, many would go several days without going home

Unequivocal Commitment

- But not all organizations require extraordinary overtime
- IBM
  - Part of the commitment can be not to work any overtime
  - More severe constraints can lead to radically productive solutions that normally considered

Sign Up Risks

- Increased inefficiency
  - Teams have a tendency to work hard, not work smart, may make more mistakes
- Decreased status visibility
  - Less insight into true progress as developers focus on the work alone
- Loss of control
  - Signed-up team takes on a life of its own, can be hard to make it change direction without taking away empowerment
- Smaller talent pool
  - Not everyone wants to sign up
- Burnout
  - Long hours can take a heavy toll

Signing Up Summary

- Efficacy
  - Potential reduction from nominal schedule: Very Good
  - Improvement in progress visibility: None
  - Effect on schedule risk: Increased Risk
  - Chance of first-time success: Fair
  - Chance of long-term success: Good
- Major Risks
  - Increased inefficiency
  - Reduced status visibility and control
  - Smaller talent pool for project
  - Burnout
- Major Tradeoffs
  - Trades possible decreases in visibility, control, and efficiency for major increase in motivation

Lifecycle Models

- Incremental Development w/Staged Delivery
- Throwaway Prototyping

Theory-W Management

- Project management framework for reconciling competing interests among stakeholders
- Ex:
  - Customers: Quick schedule, low budget
  - Boss: No overruns, no surprises
  - Developers: Interesting work, home life
  - End-Users: Lots of features, user-friendly, fast
  - Maintainers: No defects, good documentation

Theory-W

- Goal of Theory-W is to make a winner of all the stakeholders
- All stakeholders explicitly express what is necessary in order to “win”
- Everyone realizes everyone else’s win conditions
- Improves schedule savings in improved efficiency of working relationships, improved progress visibility, reduced risk

Steps in Theory-W

- 1. Establish a set of win-win preconditions before starting the project
  - Understand how stakeholders want to win
  - Establish reasonable expectations on parts of all stakeholders
  - Match people's tasks to their win conditions
  - Provide an environment that supports the project's goals
- 2. Structure a win-win software process
  - Realistic plan
  - Identify and manage win-lose and lose-lose risks
  - Keep people involved
- 3. Structure a win-win software product
  - Match product to end users' and maintainer's win conditions

Theory-W Summary

- Efficacy
  - Potential reduction from nominal schedule: None
  - Improvement in progress visibility: Very Good
  - Effect on schedule risk: Decreased Risk
  - Chance of first-time success: Excellent
  - Chance of long-term success: Excellent
- Major Risks
  - None
- Major Tradeoffs
  - Effective with schedule negotiations

Timebox Development

- Have you noticed an increase in productivity the day before flying off for vacation?
  - Get laundry done, wrap up work, pay bills, quick shower, less goofing off, etc.
  - Could do this every day, but priorities push these down
- Timebox
  - Fixed deadline for milestones
  - Refine product to fit schedule deadlines instead of redefining the schedule to fit the project

Timebox Benefits

- Emphasizes priority of the schedule
  - Schedule is absolutely fixed
  - Stresses it is of utmost importance
- Avoids the 90-90 problem
  - Where the last 10% takes longer than the first 90%
- Clarifies feature priorities
  - Tight time constraints focus attention on the top of the priority list
- Limits developer gold-plating
- Controls feature creep
  - Generally a function of time
- Helps motivate developers

Using Timebox Development

- End users must be willing to sacrifice features for schedule
- Generally uses prototyping
  - Grows like an onion with essential features at the core
  - Other features in outer layers
  - Lots of user involvement
- Timeboxes usually last 60-120 days
  - Shorter periods not sufficient to develop significant systems

Entrance Criteria

- Prioritized list of features
- Realistic schedule estimate
  - Requires some experience
- Right kind of project
  - Best for in-house business software
  - Project that can be built with rapid development languages, CASE tools
- Sufficient end-user involvement

Timebox Risks

- Attempting to timebox unsuitable work products
  - Not good for project planning, requirements analysis, or design
    - Too many downstream implications
- Sacrificing quality instead of features
  - Customer must be committed to cutting features instead of quality
  - Hard to work on a tight schedule, high quality, and all features
  - If quality suffers, the schedule will suffer too
- True timeboxing
  - Software accepted or thrown away at the deadline
  - Makes it clear the quality must be acceptable

Timebox Summary

- Efficacy
  - Potential reduction from nominal schedule: Excellent
  - Improvement in progress visibility: None
  - Effect on schedule risk: Decreased Risk
  - Chance of first-time success: Good
  - Chance of long-term success: Excellent
- Major Risks
  - Sacrificing quality instead of features
  - Attempting to timebox unsuitable work products
- Major Tradeoffs
  - Trades feature-set control for development-time control

Tools Group

- Set up a group that's responsible for gathering intelligence about, evaluation, coordinating the use of, and disseminating new tools within an organization
- Allows for some trial/error in one group instead of many groups
- Promotes the use of software tools among the organization

Tools Group Summary

- Efficacy
  - Potential reduction from nominal schedule: Good
  - Improvement in progress visibility: None
  - Effect on schedule risk: Decreased Risk
  - Chance of first-time success: Good
  - Chance of long-term success: Very Good
- Major Risks
  - Bureaucratic overcontrol of information about and deployment of tools

Top-10 Risks

- A list consisting of the 10 most serious risks ranked from 1 to 10
- Each risk has a status and plan to address the risk
- Updated weekly
- Raises awareness of risks and contributes to timely resolution of them

Top-10 Summary

- Efficacy
  - Potential reduction from nominal schedule: None
  - Improvement in progress visibility: Very Good
  - Effect on schedule risk: Decreased Risk
  - Chance of first-time success: Excellent
  - Chance of long-term success: Excellent
- Major Risks
  - None

User Interface Prototyping

- User Interface is developed quickly to explore the design and system requirements
- Often a special-purpose prototyping language used (e.g. VB)
- Thrown away or evolved into final product

UI Prototyping Benefits

- Reduced risk
  - Find bad interfaces early
  - Best suited to business software where end users are available, but possible with commercial products as well
- Smaller systems
  - Unexpectedly, features that developers think users want are not always the same as the features that users actually want
  - Features that users want but work poorly in a live system are also weeded out
  - Users get a better understanding of the system and request fewer changes
- Less complex systems
  - End-users help focus on more usable, less complex systems
- Improved visibility

Using UI Prototyping

- Throwaway or Evolve
  - Discussed previously, usually throwaway better but harder to do
- Prototyping languages useful
  - Hollywood Façade
    - Smoke, Mirrors, Hidden man behind the curtain
    - Enforced throwaway idea
- End-User Involvement throughout the lifecycle
  - Careful, users may not know what they're looking at
  - 2 second canned printout example

UI Prototyping Summary

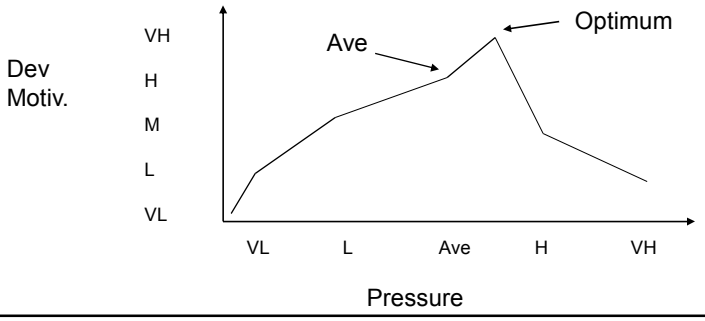
- Efficacy
  - Potential reduction from nominal schedule: Good
  - Improvement in progress visibility: Fair
  - Effect on schedule risk: Decreased Risk
  - Chance of first-time success: Excellent
  - Chance of long-term success: Excellent
- Major Risks
  - Prototype polishing

Voluntary Overtime

- Provide developers with meaningful work and motivation so they will want to work more than required
- Extra hours can provide direct productivity boost
- Care must be taken to avoid excessive, mandatory overtime

Using Voluntary Overtime

- Use developer-pull instead of management-push
  - Motivation research shows that increasing the driving force first increases performance, but excessive force drives it down
  - Pressing programmer for rapid bug elimination may be the worst strategy, but it is the most common



Using Voluntary Overtime

- Developers are naturally self-motivated, so OK to ask for a little overtime, but not too hard
- Motivate
  - Achievement of something significant
  - Possibility for growth
  - Work itself
  - Personal Life
  - Technical supervision opportunity

Using Voluntary Overtime

- Don't make it mandatory
  - Produces less total output
  - Average developer already working close to maximum level of motivation
  - Pushing developers when already motivated causes a decline in motivation
    - Decline over entire work hours, not just overtime hours
- Ask for overtime you can actually get
  - Boddie, author of Crunch Mode: 60-100 hours a week for a few weeks at a time
  - Maguire: Start doing too many personal tasks at work with that many hours, people working 12 hour days really only getting 8 hours of work done
  - Compromise, 50 hours a week?
  - Beware of burnout

Voluntary Overtime Summary

- Efficacy
  - Potential reduction from nominal schedule: Good
  - Improvement in progress visibility: None
  - Effect on schedule risk: Increased Risk
  - Chance of first-time success: Fair
  - Chance of long-term success: Good
- Major Risks
  - Schedule penalties resulting from excessive schedule pressure and excessive overtime
  - Reduced capacity to respond to emergency need for still more hours
- Major Tradeoffs
  - Requires sincere and nonmanipulative motivational practices
  - Usually required for Miniature Milestones, Timebox, Sign Up