Rapid Development

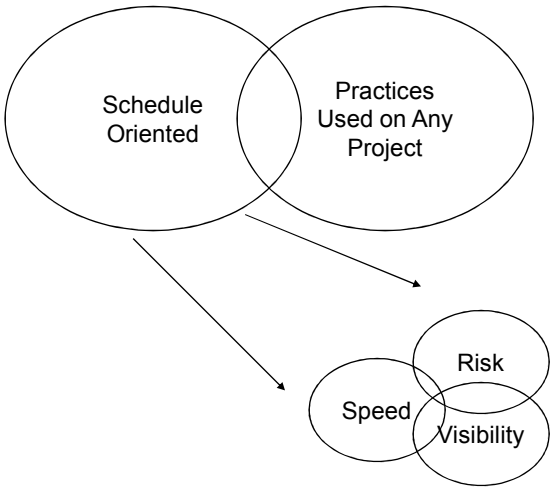
What is Rapid Development?

- Different meanings to different people
  - RAD; e.g. JAR/JAD, CASE Tools
  - Rapid Prototyping with Visual Basic
- Here Rapid Development is just a descriptive phrase
  - Contrast with “Slow Development”
  - Generic term for speedy development with short schedules, which tends to fit most projects

Attaining Rapid Development

- Two basic high-level elements
  - Choosing effective practices over ineffective ones
  - Choosing practices that are oriented specifically toward achieving your schedule objectives
- Sounds obvious?
  - But many organizations routinely choose ineffective practices

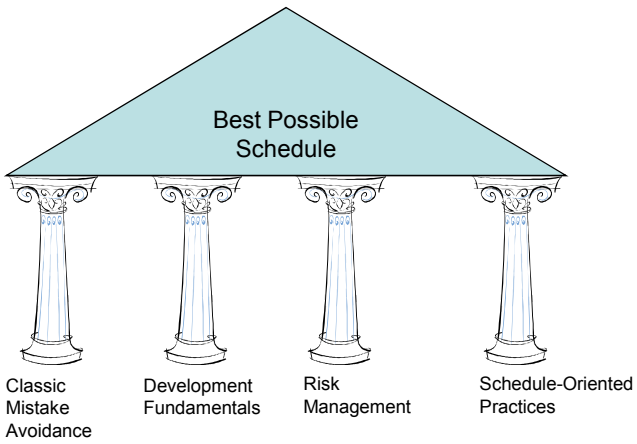
Effective Practices



General Strategy for Rapid Development

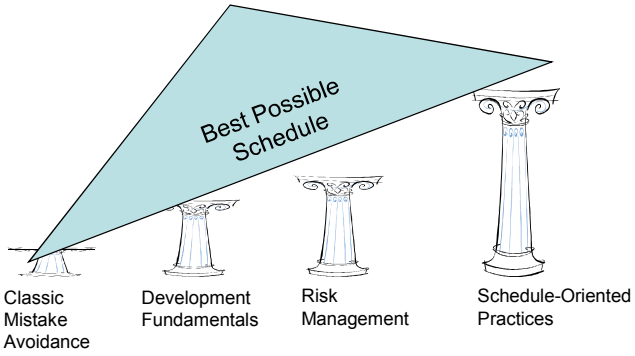
- Four part strategy
  1. Avoid classic mistakes
  2. Apply development fundamentals
  3. Manage risks to avoid catastrophic setbacks
  4. Apply schedule-oriented practices

Pillars of Rapid Development



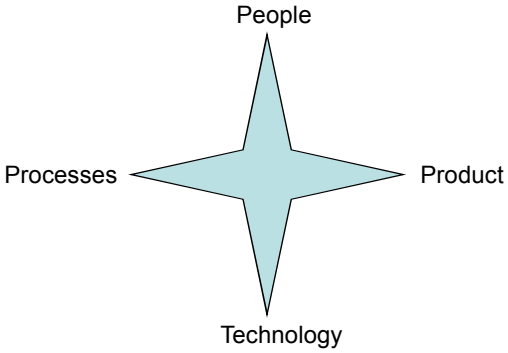
Pillars of Rapid Development

Must have the first three strategies in place or the fourth will fail



Dimensions of Development Speed

- Can leverage each of the four dimensions for maximum development speed. Can focus on all four at once.



People

- People issues have more impact on software productivity and quality than any other factor
  - Many studies since the 60's indicate productivity of individual programmers with similar experience varies by as much as a factor of 10 to 1
  - So hire the best programmers, do whatever you can to motivate the team
    - Team t-shirts? Free soda? Windows? Bonuses?
- NASA conclusion on performance of teams
  - Most effective practices leverage human potential of developers as opposed to technology

Process

- Management and technical methodologies
- Software lifecycle, quality, development fundamentals, user orientation, etc.
  - We'll say more about this in the second half of the class, touched on it already with XP

Product

- Product Size
  - Largest contributor to a development schedule – large projects take a long time
  - More features means more specs, design, testing, integration, coordination
  - Cutting the size of a program by 1/3 will typically cut the effort required by 2/3
  - Strive to develop only the most essential features
- Product Characteristics
  - Goals for performance, memory use, robustness, etc.
  - Choose battles wisely, do not insist on too many priorities at once

Technology

- Tools used for development
- Language choice
  - Low level vs. High level
  - Objects and Reuse
  - Controls and COTS software
- CASE Tools
- Project Management Tools

Which Dimensions?

- Different projects must accept limitations on the dimensions to emphasize
- Real-time fuel injection system?
  - Technology dimension likely fixed to assembly, get leverage from other dimensions
- In-House Business Program?
  - Seek gains in Technology and People, then Product and Process

Alternative Strategy – "Code Like Hell"

- Use the best possible people
- Ask for total commitment to the project
- Give near autonomy and motivation
- See that they work 60-100 hours/week until the project is finished

- Successfully used with NT and other projects!
  - But hit or miss, unrepeatable, problems with long-term motivation, generally not very pleasant

Classic Mistakes

- Usually knowledge of classic mistakes is enough to avoid them
- Using such best known practices is necessary, but not sufficient for project success
  - Still other things can go wrong, no guarantees here
- Mistakes categorized by People, Process, Technology, Product

People Mistakes

1. Undermined motivation
   - Hokey pep talk, going on vacation while the team works, etc.
2. Weak personnel
3. Uncontrolled problem employees
   - Failure to deal with problem employee is the most common complaint that team members have about their leaders
4. Heroics
   - Encourages extreme risk taking
5. Adding people to a late project
6. Noisy, crowded offices
7. Friction between developers and customers
8. Unrealistic expectations

People Mistakes

9. Lack of effective project sponsorship
10. Lack of stakeholder buy-in
11. Lack of user input
    - Get that user input early!
12. Politics placed over substance
13. Wishful Thinking
    - Not the same as being optimistic; hoping something will work with no reasonable basis for thinking it will.
    - Usually leads to big blowups at the end of a project

Process Mistakes

14. Overly optimistic schedules
15. Insufficient risk management
    - The mistakes not common enough to be classics are “risks” – must be managed
16. Contractor failure
17. Insufficient planning
18. Abandonment of planning under pressure
19. Wasted time during the fuzzy front end

Process Mistakes

20. Shortchanged upstream activities
    - Cutting out “non-essential” activities like requirements, design
21. Inadequate design
22. Shortchanged quality assurance
23. Insufficient management controls
24. Premature or overly frequent convergence
25. Omitting necessary tasks from estimates
26. Planning to catch up later
27. Code-like-hell programming

Product Mistakes

28. Requirements gold-plating
    - More requirements, complex features than needed
29. Feature creep
30. Developer gold-plating
    - Developer has to throw in the latest technologies
31. Push-me, pull-me negotiation
    - Manager approves schedule slip then adds on new tasks
32. Research-oriented development
    - Pushing the boundaries of known CS with new algorithms? Software research schedules are not nearly as predictable as software development schedules

Technology Mistakes

33. Silver-bullet syndrome
    - New technology, e.g. OOP or .NET, may not be enough to save you
34. Overestimated savings from new tools or methods
35. Switching tools in the middle of a project
36. Lack of automated source-code control
    - On average, source code changes at a rate of 10% a month and manual source code control can't keep up

Classic Mistakes

- Be aware of the classic mistakes so you do not repeat them
  - Escape from Gilligan's Island

Software Development Fundamentals

- Management
- Technical
- Quality

Management Fundamentals

- Estimation and Scheduling
  - Estimate of the size of the project
  - Estimate of the effort needed to build a product of that size
  - Estimate of the schedule based on the effort estimate
- Planning
  - Poor planning is one of most common mistakes, and source of problems more than any other problem. Examples include the following:
    - Ill-defined contract
    - Unstable problem definition
    - Inexperienced management
    - Political pressures
    - Unrealistic deadlines
  - According to Hetzel, 1993, the best projects are characterized by strong up-front planning to define tasks and schedules

Management Fundamentals

Planning a software project includes

- Estimation and scheduling
- Determining how many people to have on the project team, what technical skills are needed, when to add people and who the people will be
- Deciding how to organize the team
- Choosing which lifecycle model to use
- Managing risks
- Making strategic decisions such as how to control the product feature set and whether to buy or build pieces of the product

Management Fundamentals

- Tracking
  - Once you've planned a project, you track it to see if it is following the plan (ie. Meeting schedule, cost and quality targets)
  - Examples of management level task controls:
    - Task lists
    - Status meetings
    - Status reports
    - Milestone reviews
    - Budget reports
  - Examples of technical level tracking:
    - Technical audits
    - Technical reviews
    - Quality gates (to determine if milestones are met)

Management Fundamentals

- Tracking
  - On a typical project, project management is almost a black-box function
    - You rarely know what is going on during the project and you just have to take whatever comes out at the end!
  - On an ideal project, there is 100% visibility at all times
  - Tracking is a fundamental management activity
    - If you don't track a project you can't manage it
  - Efficient tracking allows you to detect schedule problems early, while there is a chance to fix them

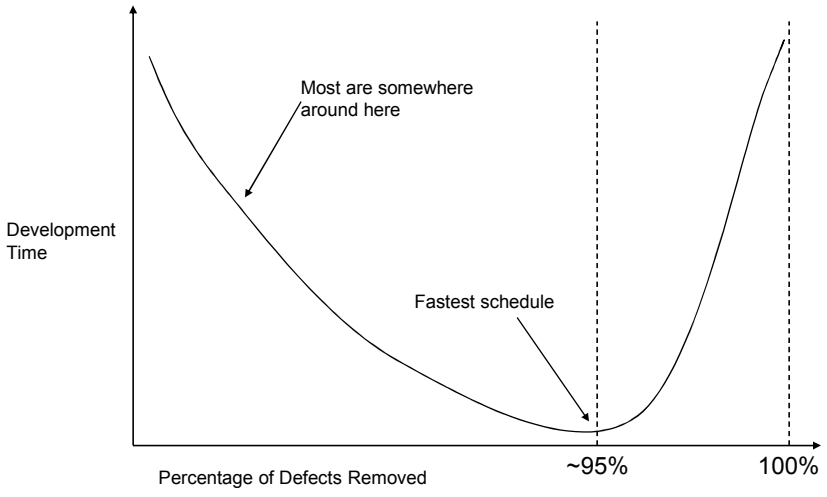
Management Fundamentals

- Measurement
  - Collection of data, in addition to cost and schedule data
  - E.g. measurements of how large the program is in lines of code, number of defects
  - Can then use the measures of size of the program to see if can maintain schedule or not

Quality Assurance Fundamentals

- Quality-assurance fundamentals provide critical support for maximum development speed – if a product has too many defects the developers spend more time fixing the software than they spend writing it!
- Some projects try to save time by reducing the time spent on quality-assurance practices such as design and code reviews.
- Other projects try to make up for lost time by compressing testing schedules – These are some of the worst decisions a person can make to maximize development speed
- This is because higher quality (i.e. lower defect rates) and reduced development time go hand in hand
  - generally as defect rate is lower in a project, the development time will be less (up to some point, since if you aim for an extremely low defect rate may slow things down) – note some projects, life-critical must have low rates

Development Time vs. Defect Rate



Quality

- In general, many organizations develop software with defect rates that give them longer schedules than necessary
- Study of 4000 projects has shown that poor quality is one of the main reasons for schedule overruns and is implicated in nearly half of all cancelled projects
- There is a cut-off for acceptability in many applications – good to develop systems that are 95% defect free by time of release (also related to speed, these systems get done fastest)
- If there is more than 5 % of your defects after your first release, there will be major problems!

Some Statistics

- Each hour spent on quality-assurance activities (e.g. design reviews) saves from 3 to 10 hours in downstream costs
- A requirements defect that is left undetected until construction or maintenance will cost 50 to 200 times as much to fix as it would have cost to fix at requirements time
- A defect that isn't detected upstream (during requirements or design) will cost from 10 to 100 times as much to fix downstream (during testing) as it would have cost to fix at its origin.
- The further from its origin that a defect is detected, the more it will cost to fix

Error Prone Modules

- A module that is responsible for a disproportionate number of defects
- On IBM's IMS Project they found that 57 percent of errors were clumped into 7% of the modules
- Boehm: 20% of the modules in a program are often responsible for 80% of the errors
- Error prone modules cost more to develop
  - If a normal module costs about $500 to $1000 per function point to develop, an error-prone module might cost $2,000 to $4,000
- Identification and redesign of error-prone modules may need to be a priority if development speed is important
  - Guideline: 10 defects per 1000 lines of code, think about redesign

Testing

- The most common QA practice is execution testing
  - Execute the program and see what it does
  - Unit testing – where developer checks his or her own code
  - System testing – an independent tester checks to see whether the system operates as expected
- Unit testing can find anywhere from 10-50% of the defects in a program
- System testing can find from 20-60% of a program's defects
- Together their cumulative defect-detection rate is often less than 60%
- Rest of errors are found by some other error-detecting method (e.g. reviews, or by end users)

Testing

- Testing is considered by some managers as the black sheep of QA practices as far as development speed is concerned
  - But without it the product may never get released properly
  - Moreover it can be the messenger that delivers bad news if the project is not implemented well
- Plan ahead for bad news – set up testing so that if there is bad news to deliver, testing can deliver it as early as possible

Technical Reviews

- Reviews used to detect defects in requirements, design, code, test cases, or other project artifacts
- Vary in level of formality and effectiveness
- Play a more critical role in development speed than testing does

- Walkthroughs
  - Any meeting at which two or more developers review technical work with the purpose of improving its quality
  - Useful for rapid development because you can detect errors well before testing
  - Can be used to detect a requirement at specification time, before any design or code
  - Can find between 30 and 70 percent of errors in a program

Technical Reviews

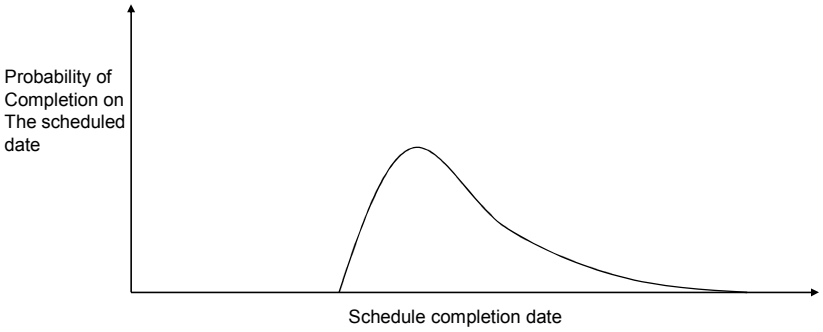
- Code Reading
  - A more formal review process than a walkthrough
  - The author of the code hands out a sources listing to two or more reviewers
  - The reviewers read the code and report any errors to the author
- Inspections
  - A formal technical review
  - Developers receive special training in inspections and play specific roles during the inspection
  - The “reviewers” examine the product before the meeting
  - The “author” summarizes the product during the meeting
  - The “scribe” records everything
  - Can find from 60-90 % of the errors in a program
  - One study- one hour spent on inspections avoided an average of 33 hours of maintenance, and inspections were up to 20 times more efficient than testing

Technical Reviews

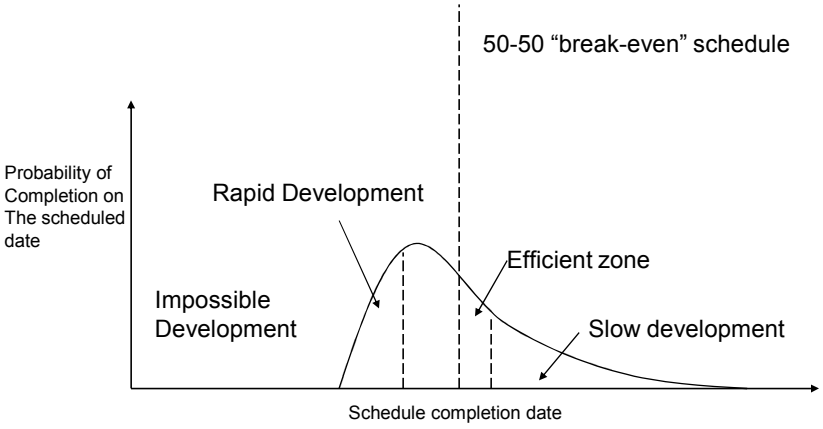
- Technical reviews are an important supplement to testing
  - They find different errors than testing
  - They find defects earlier
  - Can be more cost effective
- Testing detects only the symptom of the defect, the developer still has to find the cause by debugging
- Reviews lead to a more preventative approach
- Also provide a forum for developers to share their knowledge of best practices

Odds of Completing on Time

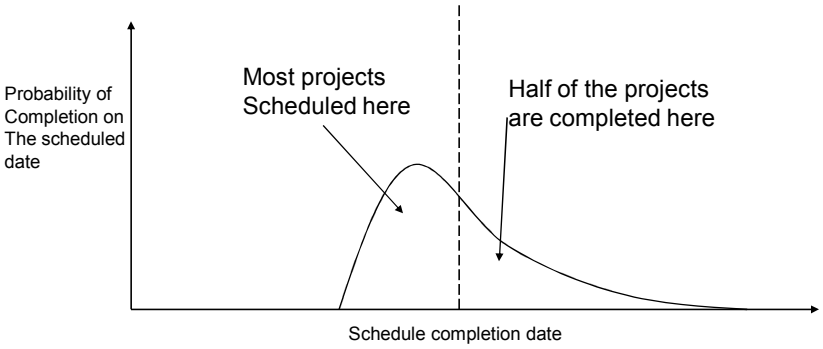
- Too many variables to set a schedule with 100% accuracy
- Probability distribution something like the curve below



Odds of Completion on Time



Odds of Completion on Time



Overcome the perception of slow development

- Eliminate wishful thinking
- Lengthen gap between planned and actual completion dates
- Keep customers informed of progress

Where the Time Goes

Activity	Small Project (2,500 lines of code)	Large Project (500,000 lines of code)
Architecture/Design	10%	30%
Detailed Design	20%	20%
Code/Debug	25%	10%
Unit Test	20%	5%
Integration	15%	20%
System Test	10%	15%

Development/Speed Trade-Offs

- Balance between Schedule, Cost, Product
- Customers “hold” one or two corners, developer can tell them what the other must be