The Practices of Small Software Projects

Khaled El Emam

Sources

- Objectives:
  - Develop an understanding of what practices small projects follow to succeed
  - When to use these practices
- Basis of this presentation:
  - My research, development and management work, and consulting
  - Surveys and benchmarks
  - Cutter Consortium reports
  - Literature (scientific and trade)
  - Extensive set of interviews in 2003

Does Size Matter

- Small projects can be in any size company
- Benefits of small projects:
  - More likely to finish on time and be on budget
  - Can be more productive
  - Can have better quality
- Best practices for small projects:
  - Both agile and traditional methodologies work well - it depends on the context of the business
  - Driven by business needs

Overall Success Rates

- Success defined as on time and on budget with all features and functions as originally specified

Project Size	People	Time (months)	Success
< $750k	6	6	55%
$750k-$1.5M	12	9	33%
$1.5M-$3M	25	12	25%
$3M-$6M	40	18	15%
$6M-$10M	250+	24+	8%
$10M+	500+	36+	0%

Based on a study by the Standish Group.

MIS Projects

Size (FP)	Early (%)	On-Time (%)	Late (%)	Cancelled (%)
1	6	92	1	1
10	8	89	2	1
100	7	80	8	5
1,000	6	60	17	17
10,000	3	23	35	39
100,000	1	15	36	48

Data provided by Software Productivity Research.

Outsourced Projects

Size (FP)	Early (%)	On-Time (%)	Late (%)	Cancelled (%)
1	5	93	1	1
10	8	90	1	1
100	7	85	6	2
1,000	8	67	15	10
10,000	1	38	34	27
100,000	1	26	40	33

Data provided by Software Productivity Research.

Systems Projects

Size (FP)	Early (%)	On-Time (%)	Late (%)	Cancelled (%)
1	6	90	3	1
10	8	85	5	2
100	10	78	8	4
1,000	5	68	17	10
10,000	3	40	32	25
100,000	1	35	34	30

Data provided by Software Productivity Research.

Commercial Software Projects

Size (FP)	Early (%)	On-Time (%)	Late (%)	Cancelled (%)
1	9	88	2	1
10	8	85	5	2
100	7	75	12	6
1,000	5	65	20	10
10,000	3	35	37	25
100,000	1	30	34	35

Data provided by Software Productivity Research.

Military Projects

Size (FP)	Early (%)	On-Time (%)	Late (%)	Cancelled (%)
1	2	92	5	1
10	3	87	8	2
100	4	78	12	6
1,000	3	65	22	10
10,000	2	40	33	25
100,000	1	30	36	33

Data provided by Software Productivity Research.

Quality

	Small		Medium		Large	
	Avg.	**Best**	**Avg.**	**Best**	**Avg.**	**Best**
MIS	0.15	0.025	0.588	0.066	1.06	0.27
Systems	0.25	0.013	0.44	0.08	0.73	0.15
Comm.	0.25	0.013	0.495	0.08	0.79	0.21
Military	0.263	0.013	0.518	0.04	0.82	0.18

Data provided by Software Productivity Research.

Team Size

- Data on more than 2000 software projects world-wide were collected during the 2003/4 ISBSG benchmark
- There is a linear relationship between team size and system size (in FP) up to a certain size
- After that a much larger team is needed per unit of functionality delivered
- The most productive size appears to be 2-3 programmers
- Beyond 4-5 staff members additional management and support effort is required

Some Agile Principles

- Customer satisfaction through early and continuous delivery of software using short iteration cycles
- Accept and accommodate changing requirements at any time during development
- Developers work daily with business people
- Minimize documentation and maximize face to face communication
- Maximize the amount of work not done

XP Practices

- Metaphore
- Release planning
- Test-driven development
- Pair programming
- Refactoring
- Simple design
- Collective ownership
- Continuous integration
- On-site customer
- Small releases
- 40-hour work week
- Coding standards

Agile In Practice

- It would seem at first glance that agile practices are the most appropriate for small projects - but that is not necessarily the case
- Successful small projects, irrespective of whether they use agile methods or not, have some common practices
- Agile practices add value, but are not appropriate for all cases

Primary Objective of Projects

	Agile (%)	Traditional (%)
On Time	52	7
On Budget	13	18
High Quality	22	47
High Functionality	13	28

Based on a survey by the Cutter Consortium. Projects were using either agile or traditional.

Schedule Slippage

Slippage (%)	Agile (%)	Traditional (%)
0	7	4
1-10	41	20
11-25	31	29
26-50	13	24
51-75	3	12
76-100	3	9
>100	2	2

Based on a survey by the Cutter Consortium. Projects were using either agile or traditional.

Budget Overrun

Overrun (%)	Agile (%)	Traditional (%)
0	7	5
1-10	34	15
11-25	38	29
26-50	12	25
51-75	3	16
76-100	4	6
>100	2	4

Based on a survey by the Cutter Consortium. Projects were using either agile or traditional.

Perceived Quality

- The study showed that perceived quality from using both approaches are the same
- However, this is likely a function of the lower expectations from the users of agile methods (i.e., it does not necessarily mean that the quality on an absolute scale is much better or the same)

Project Size (Months)

Contents
Small
Agile
Lessons
Cases
Conclusions
End

Months	Agile (%)
0-1	6
1-3	29
3-6	39
6-9	20
9-12	6
>12	1

Based on a survey by the Cutter Consortium. Projects were using agile methods. 2003 numbers.

Project Size (Team)

Contents
Small
Agile
Lessons
Cases
Conclusions
End

Team Size	Agile (%)
0-2	7
3-10	62
11-20	17
21-50	9
51-100	3
101-200	2
>200	0

Based on a survey by the Cutter Consortium. Projects were using agile methods. 2003 numbers.

Agile Implementation in Small Projects - I

- Strong reluctance to use them on mission-critical or high-reliability systems (e.g., critical banking applications) and infrastructure projects - as one manager put it “agile methods scare me”
- It is much preferred by developers, and where developers have influence (e.g., technical leads play the role of project managers) they tended to be more successful in their implementation
- Not all of the agile practices are implemented - projects tend to be selective in their adoption based on their culture
- Few bring external experts to help them
- Few companies have adopted pair programming successfully

Agile Implementation in Small Projects - II

- Architecture is emphasized more in practice as well as the use of models & requirements documents
- When architecture is ignored (i.e., no overall architecture or not recognizing when refactoring the architecture is needed) this causes problems
- Most had some forms of coding standards - with varying degrees of rigor
- Agile methods are perceived to be good for morale and for building a good team - however the longer term value of this is questioned because esp. small companies are very fluid with layoffs and acquisitions and therefore frequent staff changes
- Perceived by senior management as “hacking”

Agile Implementation in Small Projects - III

- Experts in certain areas tend to touch the pieces of the code that they are most expert in and avoid other areas - therefore there is still a sense of individual code ownership even though the principle is of having collective ownership
- Acceptance test defects treated like stories (scheduled and prioritized by customer)
- Low end testing staff are not likely to survive easily in an agile environment

Users

- Getting a user full-time or even just often is a challenge with large multi-disciplinary teams and when the user is critical to the success of the business
- There have been cases with hostile clients (changing priorities constantly, not willing to commit, not willing to change scope) - the quality of the customer makes a big difference on whether agile methods will work or not
- User participation is known to be beneficial but only when there is uncertainty about the problem - there is evidence that if the requirements are certain users feel resentment and dissatisfaction if they spend time explaining obvious functionality
- Marketing, product management, and virtual users have sometimes been used

40 Hour Weeks

- One of the big selling points about the use of agile methods is that they alleviate the chances of burnout and allow staff to have normal lives
- Towards the end of projects/iterations as the code base increases, performance problems start to surface - this leads to a back-end loading of the projects as staff scramble
- Many of the tools typically used when implementing agile projects are not that good for sophisticated performance evaluation - especially with prevalent underfunding of QA/QC activities
- A good example is the speech recognition application
- Need to include performance (and non-functional requirements) in the initial stories and have a more developed performance evaluation function

Refactoring

- Many projects neglect refactoring - as time passes refactoring becomes more and more difficult to do
- Refactoring data is very difficult to do (e.g., change database structure in the middle of an iteration whereby there already exists data in the deployed DB)
- It is well established that a nontrivial percentage of “defect fixes” introduce new defects - this bad fix ratio would be amplified if one is constantly making changes (while the regression test suite would catch that, the amount of rework would still be significant)

Architecture

- The lack of a coherent architecture creates many difficulties with subsequent refactorings becoming more complex
- When basic (reusable) architectural services are not stable, changes can have very serious ripple effects (both import and export coupling increases ripple)
- Frequent cloning of code distributes identical defects throughout the system - defect propagation
- General recommendation is to include an architect role on the team

Legacy Systems

- With the frequent acquisitions and mergers of the last few years, companies have legacy code bases that they need to maintain with little in-house expertise
- Very difficult to start using agile methods on legacy systems
- Incrementally refactor as changes are made, but this may take a long time
- Legacy systems written in different languages and good tools to support all of these are not easily available

Test First

- A practice that is not implemented frequently but probably one of the best practices in agile methods
- There is evidence that test first programming results in more test case being written (programmers are forced to write tests first)
- With test last, programmers do minimal testing or test until they run out of time. Therefore, unit testing is spotty at best
- The quality of the test cases tends to be better as well since they are more generic and deal better with boundary conditions
- Test-first has remarkable benefits for the best (i.e., most skilled and experienced) programmers and provides little benefit for those at the low end

Pair Programming

- Effective peer pressure mechanism (focus, interrupts)
- Few companies have successfully implemented pair programming - in some cases it optional
- Perceived to be most useful when there are feature uncertainties or complex functionality
- Fewer have the setup to support it (many desks that can seat two people comfortably for extended periods and rewards structures)
- There is anecdotal evidence that pairs are more productive and produce higher quality work - but the evidence is rather weak
- In principle regular formal inspections find more unique defects than pairs - but pairs spend so much time doing reviews that that probably dilutes any advantages

Co-location

- There is good evidence that co-location improves quality and productivity
- Most useful for short projects and staff require a break (quiet time) after a few iterations
- The high bandwidth facilitates the exchange of information rapidly
- Need to have private (concentration) space

Time-Boxing

- Most projects used a “time-boxing” strategy
- This basically means that the deadlines are decided at the very beginning based on a rudimentary understanding of the system (i.e., the deadline is not really driven by an estimate of how much work is involved)
- Allow a couple of weeks slippage on schedule before raising red flags
- Once the deadline is determined, the negotiations on scope start
- This works well if progress is tracked very closely
- When developers are involved in these negotiations they quickly improve their personal estimation skills

Automation

- Where automation is used in testing, it is usually homegrown
- Homegrown tools are typically not very sophisticated (e.g., none incorporates parsers that can perform static analysis of codes to find defects)
- Perceived difficulty, even today, to have automated GUI testing tools (some say the tools are not good enough and also usability testing is typically done towards the end when the product has been developed)
- Moving from homegrown to commercial tools is a very painful process that many would rather avoid
- Performance testing tools tend to be commercial
- Some visual tests (e.g., image quality) cannot be automated
- Regression testing can be slow

Case Study - The Startup

- Different processes before and after crossing the chasm
- Funding for automation is very scarce
- Most important practices: version control, configuration management, release management
- Important to have a good architecture from the beginning - a good understanding of where system instability may be (what is likely to change)
- Quality assured through peer reviews
- Key influencers may divert the process and undermine good practices

Case Study - The Shrink-Wrap Software Company

- Requirements were defined by the product managers and marketing people
- These serve as the basis for development work
- Defect discovery and correction quotas are used as gates through a high-level process
- Features broken down into around 5 day tasks (or less)
- Tracking through a features database with estimated effort and actual effort for each task
- Project broken down into small “carts” or time-boxes
- Features are shifted among carts or renegotiated with product management as the project progresses
- Missing deadlines is not an option - each day of delay on a release can be worth millions of dollars

Case Study - The Hi-Rel Systems Company

- Build very high reliability systems for banks that work in mainframes
- ISO 9001 registered company
- At least one incident where the undocumented processes almost caused a law suite (i.e., the value of documentation is appreciated)
- If a field defect is found, the whole company stops everything else
- Use extensive testing and selective peer reviews
- Requirements volatility not a major issue
- Well defined process and everyone follows it
- Need to have a predictable process for their market segment
- Little commercial automation - most of tools used are built in-house

Case Study - The Small Contractor

- Negotiate the requirements with the client and ensure that scope is well defined - well defined procurement and contracting process that is followed
- Get the client involved through regular reviews and updates
- Good scope control ensures that requirements volatility does not become unmanageable
- Most of the engineering staff have been on the ‘dark side’
- Strong adherence to internal standards

Case Study - The Frontier Projects

- This was a series of small projects that dealt with new technology and high risk development within a larger organization
- They are the only projects that are seriously using agile methods
- Small releases and a formalized process for tracking stories
- Defects found during acceptance testing are treated as mini-stories
- Optional use of pair programming
- Constant integration and testing
- Maintain the ‘traditional’ process for core or complex functionality

The Good & Bad Practices

- User participation is a good thing, most of the time
- Users must be managed very carefully
- Test-driven development is a good thing because it forces testing to occur
- Small and frequent releases helps manage uncertainty
- Refactoring is necessary but not practiced enough
- Doing away with documentation and up-front analysis and design is a bad thing
- Doing away with specialization is a bad thing
- The evidence on pair programming is very shaky – a high risk practice that is likely to fail
- There is a real question mark on whether in the real world these practices can be scaled for long-lasting and large projects

Final Words

- Small projects do use disciplined practices - but these are geared to their own context
- They rely very strongly on good people - average or below average developers and designers would not survive in these environments
- There were mostly implicit (and in some rare cases very explicit) punishment mechanisms for sloppy work
- Continuous integration reveals sloppy work very fast
- Tools not used that often for refactoring
- All had (or wish they had) some form of architecture before embarking on a new project

Contacts

Khaled El Emam
kelemam@cheo.on.ca
kelemam@trialstat.com
(613) 797 5412