FYP 2010-2011

Agile practices for surviving your Final Year Project

Gwenn Bosser
The Inevitable Truth about Every Development Project

● Planning is always wrong
  – You can't be an expert in everything you tackle. Neither can your supervisor.
  – Even if you are, the initial planning you do is usually using some sort of “I'll put the time I think it takes me and multiply it by 2 just in case” rule.

● Technological risk assessment is always incomplete
  – Especially if you are using a given SDK or technology for the first time.
The Inevitable Truth about Every Development Project

- “Human material” is fragile
- There are only 24 hours in a day and 7 days in a week

   And caffeine can only do so much

Disclaimer: there are debates about software development methods. A lot of what will be said here is subject to controversy. Talk with your supervisor.
Topics

- Software Development lifecycles and methods from an agile extremist perspective
- Some Agile Practices reviewed
- Agile Toolbox for a one-person project
  - Test Driven Development
  - Refactoring
  - Planning, re-planning, and re-planning again
- Discussion: controversy about code documentation, UML and code comments.
Software Development Lifecycles Primer

- Two main families:
  - Waterfall lifecycles: strict, V, incremental...
  - Evolutive lifecycles: spiral, in every agile methodology
Strict Waterfall

- After each step, verification and inspection
- A big inconvenient: not very resilient to specification change
- Well adapted to simple, small-scale, very contractual project
  - If you are very lucky your project falls into this category
  - Note however that such FYP projects are quite rare.
V lifecycle

- To each stage on the left branch corresponds a testing phase on the right branch
- Difficult/Heavy to backtrack to the previous steps
- Difficult to re-plan: risk to discover too late that the project was too ambitious
Pure Incremental lifecycle

- Assumes the project is easy to decompose in relatively independent functionalities
- Planing done in successive, sometimes parallel, ideally independent tasks
- Each increment is a stable but incomplete version of the software
The Agilist point of view about Waterfall-type lifecycles

• They are adapted to very contractual projects

• They are really bad at dealing with changes because of the need to backtrack to previous phases

=> Especially bad in project contexts
  – With unknown variables and technical risks
  – For which specification are not very precise initially

• Possibly useful used in micro-cycles in a more adaptable lifecycle
Spiral lifecycle

• Based on the production of successive working prototypes, approaching more and more closely the final software

• Each iteration comprises
  – Specification
  – Analysis of technical issues and critical points
  – Development of prototype
Spiral lifecycle

- At least 4 prototypes:
  - Risk analysis
  - Central application functionalities and interface prototyping
  - Definition and validation of more accurate specifications issued from the previous prototype
  - Final Prototype for validation

- There is a new estimate of planning and workload produced/refined after each stage
A few other classical lifecycles and methods

- Unified Process
- Rapid Application Development
Unified Process

• A meta Process
  - Start from a review of UML use cases
  - Development oriented by an architectural description of the project
  - Generates a great (insane the agilist would say) number of artefacts
  - Possibility to discard some artefacts on a case by case basis, but all have to be considered
  - Iterative cycles for each of the 4 phases (use cases, architectural prototype, components development, tests)
  - Numerous tools to help manage such projects and their artefacts (automation of diagram generation/maintenance...)
The Agilist point of view about the Unified Process

- In most cases, it's an elephant in a china porcelain shop.
- It's a nice security blanket for overwhelmed project managers or control-freaks clients
- If you are prone to design perfectionism, it is possibly the best way to end up involuntarily implementing the infamous anti-pattern Analysis Paralysis
- Some code generation and reverse-engineering tools associated with this method are fun to play with, and can help save time in situations where you are required to use the UML diagrams
Rapid application Development (RAD)

- Most commonly applied to the production of small applications with user interfaces: small, short, one shot projects
- A final user (e.g. client) is involved in the process
- Relies on tools for rapid prototyping (interface builders, etc), for automating “stupid” tasks.
- Short iterations, presented to the client for frequent feedback
The Agilist point of view on RAD

- Represents an important first step towards agile, adaptable, flexible methodologies
- QA approach often not strong enough for sizeable projects. This produces applications which tend to be a nightmare to maintain

  Not the best choice if you foresee to take holidays and then have to understand what you meant by that bit of code a week later

See also anti pattern Big Ball Of Mud
Agile methods

Manifesto for Agile Software Development
We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value:
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan
That is, while there is value in the items on the right, we value the items on the left more.

http://agilemanifesto.org/
A few selected Agile Precepts Discussed

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Securing functional changes, definition of priorities between functionalities. Acknowledging the fact that initial requirements are most often incomplete

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Frequent iterations, each one of them producing a demonstrable executable. Allows frequent feedback.
A few selected Agile Precepts Discussed

Continuous attention to technical excellence and good design enhances agility

- Refactoring at the heart of the methodology, readability and maintainability essential.
- Idea that good design emerges and is not pre-build

Simplicity--the art of maximizing the amount of work not done--is essential

- KISS (Keep It Small and Simple). Code Metrics exist, e.g. cyclomatic complexity, coupling...
- YAGNI (You Aren't Going to Need It)
- Test Driven Development as a discipline
A few selected Agile Precepts Discussed

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly

- Refactoring principle applied to the process itself
- Regular debriefings to adapt working habits
- You can do this with your project supervisor
Instances of Agile Methodologies

- eXtreme Programming: despite the fancy name a very constrained and regulated practice
  - http://www.extremeprogramming.org/
- Crystal Clear: designed for smaller scale projects
  - http://www.agilekiwi.com/crystal_clear.htm
- Scrum: very much in fashion, includes fancy nicknames and metaphors using bacon, eggs, chickens and pigs.
  - http://www.controlchaos.com/
Agile Toolbox for your Final Year Project

- Test Driven Development
- Refactoring
- Planning Game
- Versioning
Test Driven Development Principles

- Writing the tests, then writing the \textit{minimum} amount of \texttt{code} verifying the tests

  test writing becomes the fun part, code writing becomes less creatively entertaining

- Use your common sense, no need to test for every Integer value: concepts of test coverage, limit cases

- If there is a bug, add a test evidencing it, then fix the code

- Use tools allowing to automate the tests (JUnit, Boost.Test...)

Gwenn Bosser
Test Driven Development
Outcomes

- Tests can complete/replace documentation, comments
- Tests can complete/replace specification
  A lightweight form of formal specification
- It's normal if writing tests takes up to 80% of the total development time
  But that's ok because it becomes fun

Gwenn Bosser
Test Driven Development
Outcomes

- Forces you to improve modularity
- Incites you to not write useless code (see YAGNI)
- Often allows to not need the debugger
- Security and comfort for modifications and refactoring practices
Test Driven Development

● For more:

*Test Driven Development; by example*, Kent Beck

*Test-driven development : A practical guide*, Dave Astels
(available at your local library)
Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Martin Fowler
Defining Refactoring

- Refactoring: a change applied to a program internal structure, in order to make it more understandable and easy to modify, without changing this program observable behaviour

- To Refactor: to apply a series of refactoring

- Recipes allowing to systematise the production of good code

- Refactoring can lead to design patterns
Why Refactor

- Amelioration of the design
- Makes the program easier to understand
- Helps finding bugs
- Allows to progress faster
When to Refactor

- Before adding a functionality
- Before attempting to fix a bug
When to not Refactor

- When just everything would have to be thrown away.
  Just don't reach this point
- The night before your FYP hand-in date (productivity gain is then quite unsure)
- However, generally, the fact that you think you don't have the time to refactor is a good indication that you NEED to refactor
Refactoring should follow a method

- Identify a symptom in the list provided by Martin Fowler's book (*Refactoring: improving the design of existing code*)
- Look for the proposed refactorings
- Apply one of the recipe
- Compile, build, execute tests
- Start again

Note: some IDEs provide automatic refactorings. It works better with languages such as C# and Java than C++ because of the macros.
Selected Examples of “Code smells”

- Code duplication
- Method body taking more than half a screen in a decent resolution
- Use of switch instruction
- Data always used together
- Code you put because one day it will be useful for a future functionality (dead code)
- Long delegation chains (object1.object2.object3.object4.method)
- Lots of comments
Example for the Symptom Shotgun Surgery

- A desired change will have an impact on a big number of classes
- Effect of the refactoring:
  - Modifications will only be located in one class
  - Encapsulation of a frequent change in one class
- Refactorings in use:
  - Move Method
  - Move Field
  - Inline Class
Example: Description of the Move Field Refactoring

• Motivation: field mainly used by another class

• Recipe to follow:
  – Use *Encapsulate Field* (field becomes private with an accessor)
  – Compile and test
  – Create a new field with accessors in the target class
  – Compile the target class
  – Decide how to reference the target in the source
  – Remove all references in the source class
  – Replace all reference in the source by a call to accessors in the target
  – Compile and test
Planning Game (XP)

- Continuous planning: common point between numerous agile methodologies
- Consists in practices and metaphors allowing dynamic and flexible planning
  - Permanently questioning the planning
  - Always taking into account unexpected events/issues
  - Better risk management
  - Fine-grained decomposition of the tasks
Planning, what for?

• Planning is useful to ensure that
  – You are working on the most important tasks which are left to do (not just the ones you like better)
  – You can rapidly adapt to unexpected events/issues

• Planning is useless if just a security blanket, when you try to force all your work to fit into the initial plan, it has to be continuously revised/updated
How to play

• A good reference book:

*Planning Extreme Programming*, Kent Beck & Martin Fowler
How to play: the tools

- The problem with Gantt charts according to the Agilist:
  - It is too high level for everyday use
  - It enforces a certain level of rigidity in the planing
  - It only gives the illusion of control to project managers
  - See: http://c2.com/cgi/wiki?GanttChart

- How you would plan the agile way:
  - A sprint: is an iteration in the process
  - A product backlog: the lists of tasks left to do. By prioritising, you use it to plan for the content of the next sprint. Can be augmented if the task definition was imperfect, tasks are removed when achieved.
  - Burndown chart: a very simple visual representation of the progress of the project
  - Velocity: amount of work achieved during a sprint
An example of burndown chart
How to play

• A few tips:
  - *Not Enough Time vs. Too Much to Do:* **Prioritise**
  - **Record the time taken for each previous development** and use this to weight any future task. Since there have been issues in the past, you indirectly take into account future problems.
  - For the first planning, make a list of “use cases”/”stories” and prioritise. Add bugs to this list for the planning revisions.
  - Use **metrics** and **charts**: velocity, burndown, ratio developed functionalities/total to attest your progress.
  - **Man months are useless** they are too big, use candies to weight your tasks it is arbitrarily fine-grained. Also, when you're done with a task you can eat them. You can use hours as well but remember estimations are usually wrong.
How to play

• Some former students successful experience:
  – At the beginning of the project, the student presents an initial, approximative **product backlog**, and discusses it with the supervisor. This includes research and writing up tasks.
  – **Sprints length** are fixed to the amount of time between meetings with the supervisor.
  – During each meeting, **the content of the next sprint is discussed**, based on the current backlog and estimation of weight for each task. Priorities are discussed with the supervisor. If tasks have been added to the backlog (bugs, unexpected technical issues with SDKs...), their weighting is discussed as well
How to play

- The project diary sheet is filled in advanced of each meeting, with what scheduled tasks have been achieved, list of tasks eventually added to the backlog, velocity for the sprint, and average velocity since the start of the project. This helps to inform the definition of the next sprint.

- Why this works well:
  - With a glance at the burndown and velocity, your supervisor detects if there is a problem with your progress even if he is not a specialist, and can take steps to help you.
  - Discussing priorities with your supervisor allows to make sure that even if your project was too ambitious, you will end up with the most important things in your product (e.g. your project is on AI but you are developing your own 3D engine as a test platform...)

Gwenn Bosser
Discussion

UML diagrams, code comments, code documentation
When are they really needed?
Links and References, not Exhaustive

Planning Extreme Programming Kent Beck Martin Fowler

Extreme Programming Explained Kent Beck

Agile Project Management with Scrum Ken Schwaber

Crystal Clear A human-powered methodology for small teams Alistair Cockburn

Links and References, not Exhaustive

*Refactoring: Improving the design of existing code* Martin Fowler

*Test Driven Development; by example* Kent Beck

*Refactoring to Patterns* Joshua Kerievsky
Links and References, not Exhaustive

• Agile methods:
  – http://agilemanifesto.org/
  – http://www.agilealliance.org
  – http://www.controlchaos.com
  – http://www.xprogramming.com

• The first wiki, full of the latest discussions about software engineering topics and design patterns
Links and References, not Exhaustive

- Managing your project with Scrum, defining a backlog, maintaining your burndown chart, computing project metrics:
  - http://www.acunote.com/
  - http://pangoscrum.com/

- Code metrics tools, to help identify “code smells” or define a unit testing policy
  - http://cppncss.sourceforge.net/
  - http://cccc.sourceforge.net/