Object – Oriented Design with UML and Java

Part XIX - Project Management

Version 2011

Software Development Process

A good Software Development Lifecycle (SDLC) process must be flexible enough to accommodate ***software***.

- It must be tailored for each project's individual needs.
- The SDLC is usually driven by the needs of a business.
- Control the chaos with minimal overhead.
- It should incorporate the needs of all stakeholders, including the programmers.

The Project Manager

The Project Manager should be “bilingual,” speaking the languages of business and technology. (S)he is responsible for:

- Tailoring the process to the project.
- Shielding developers from unnecessary work & distractions.
- Addressing enterprise-wide reuse & consistency issues.
- Deciding what deliverables to produce & when. Planning.
- Ensuring that the scope is well-defined.
- Setting and enforcing quality & testing standards.
- Assessing & mitigating risk.
- Managing, rather than reacting to, change and complexity.
- Expectation management & communication in general.
- Time, money, and people.

Corporations often look for P.M. qualifications such as ***Prince2*** (**PR**ojects **IN** a **C**ontrolled **E**nvironment).

Project Life Cycle: *Waterfall*

- Inception, analysis, design, code, test, deliver, maintain; in that order.
- Each phase completes before next phase begins.
- Much formality, ceremony and detailed documentation (including contracts).
- Good when requirements are stable & well defined & small.
- Relatively easy to define and manage.
- Can lead to stable software with strong traceability to the requirements.
- The schedule and budget are easy to underestimate (the devil is in the details; hindsight is 20/20).
- Relatively few opportunities to assess priorities and make adjustments.
- Can also lead to brittle software when all those last minute changes are "shoehorned in" without regard to the overall design.
- Testing and user acceptance happen late, implying late schedule risk.

Project Life Cycle: *RAD*

Rapid Application Development (**RAD**):

- Inception, no design, no modeling, code like crazy until done.
- Short-term schedule driven.
- Involves end-users.
- Code ends up full of quick fixes, hacks and kludges.
- Leads to inconsistencies everywhere.
- Leads to redundant information and redundant code.
- OK as a prototyping technique - the code gets thrown away.
- Be disciplined - throw the code away!
- A common model for organizations who believe the hype that RAD is the fastest way to custom software.
- Very difficult to maintain and enhance.

Project Life Cycle: *Scrum*

- An “**agile**” project management methodology.
- **Scrums** are daily meetings (15 minutes, standing up, strictly on time). Developers are asked 3 questions: What did you do yesterday? What do you plan to do today? What obstacles are in your way?
- The **scrum master** removes impediments and enforces process.
- Iterations are called “**sprints**” (ending with **retrospective** and demo).
- The **product owner** puts prioritized **user stories** in a **product backlog**.
- Each sprint begins with a **planning meeting**, where the **scrum team** selects the **sprint backlog**, the stories they can complete during the sprint.
- These stories are broken down into **tasks** (hand written index cards).
- Tasks are grouped into three buckets: not started, in progress, and **done**.
- The definition of done = working software, ready for the **sprint demo**.
- Scrum can be used in conjunction with a governing process such as that recommended by Prince2.

Project Life Cycle: *Scrum*

- Tasks are estimated in **story points** with **planning poker**. This technique for **relative estimation** helps prevent psychological estimate grounding.
- The team's **velocity** is calculated as story points completed per calendar day. Over time, this provides an empirical metric for translating story point estimates into calendar time. Initially, guess. It's fine to use **ideal engineering hours** instead of story points. Assume the team's **focus factor** is about 70%. Later this can be calculated empirically too.
- The **burn down chart** graphs work left against time (sprint backlog story points on the vertical axis, time on the horizontal axis). A diagonal line is drawn to show ideal progress through the sprint. The scrum master updates this graph daily, so that all can see progress (things **done**) against plan.
- If the actual line is above the ideal line, the team is behind schedule. To meet the sprint deadline, the lowest priority stories may be **descoped**. In this case, a big question in the sprint’s retrospective meeting will be: Why did we estimate this so poorly? Learn. And do better next time.

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/`

Principles behind the Agile Manifesto

“We follow these principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers work together daily.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- Working software is the primary measure of progress.

The Agile Manifesto

- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain their pace indefinitely.
- Continuous attention to technical & design excellence enhances agility.
- Simplicity --the art of maximizing the amount of work not done-- is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly."

Project Life Cycle: *Incremental & Iterative*

- Inception, elaboration, *incremental & iterative* construction...
- Risk-oriented. Each iteration addresses some number of risks. Solutions are implemented. The next iteration is planned.
- Requirements change. Risks change. Priorities change.
- Changes can always be incorporated in the next iteration.
- People often don’t know what they want until they’ve seen it.
- Each iteration produces an executable which can be demo’d (and tested).
- Avoids analysis paralysis. The first design / construction phase can proceed once some high percentage of the major functionality is specified, along with an architectural vision.
- Requires less formal documentation overhead. For optimal efficiency, document at the *appropriate* level of detail and completeness.

Incremental & Iterative

- The team focuses on 1 aspect of the system at a time, increasing efficiency.
- An iteration can be time boxed. To accomplish this, use cases must be prioritized. The lower priority ones may be dropped to achieve the date.
- Achieves morale-boosting milestones relatively frequently.
- Delivery milestones allow accurate determination of progress; they also show where interfaces need refactoring & refinement.
- “You ain’t gonna need it.” Don’t code it until you do.
- Each iteration adds new functionality which tests the generality of the infrastructure; if it is found to be too specific, it may be refactored.
- Long term developer productivity and system quality are increased when opportunities are given to fix what is awkward or broken.
- Earlier testing finds problems when they are easier / faster to fix.

Waterfall versus Scrum

- Scrum is an excellent methodology that has received wide support internationally, because it works.
- Scrum makes the state of the project’s progress quite transparent.
- Scrum empowers developers to commit to realistic schedules.
- Excellent model when requirements are poorly defined or unstable, and when the technology is risky.
- The customer benefits with more efficient development and more opportunities to adjust their plans mid-stream, based on hindsight.
- Sometime is it better to have an 80% solution that works than to have a 100% solution that doesn’t work.
- It is easier to be certain about delivery dates and budgets, but harder to be certain about what functionality will be included.

Waterfall versus Scrum

But...

- Scrum is not ideal when customers demand fixed price contracts, detailed acceptance criteria, and waterfall methodologies.
- Must be vigilant to ensure that adequate documentation is produced.
- Scrum allows feature creep. Flexibility is a double-edged sword.
- Scrum requires mature and responsible developers.
- Scrum requires user / customer involvement throughout the project to clarify and prioritize requirements.
- Scrum projects require continuous testing.
- In customer relations, scrum requires a high degree of trust (compared to thick legal contracts and detailed acceptance criteria).
- Customers often demand up front: HOW MUCH WILL IT COST?

Software Risk

- ***Business*** - Will the software do what the business needs? Will it do what the market wants? Will the requirements change before it gets deployed? Will it impact the way the company does business? If the requirements are not all defined up front, as in an iterative lifecycle, will there be intolerable feature creep?
- ***Technical*** - Does the technology you are using match the hype? Will it really work? Has the third party software you are depending on been sufficiently tested? Are you locked into a third party's architecture? What if they go out of business? Will their product be supported? Can you make custom changes? Can you afford (not) to build a 7x24 fault tolerant system with zero downtime?
- ***Skills*** - Is your staff sufficiently trained in the technology? Do they have enough experience to do a good job? Can they do an *excellent* job? Are your contractors dependable? Can you get the people you need when you need them?

Software Risk

- ***Internal*** - Are there political forces working against you?
- ***Schedule*** - Does it absolutely have to get done by a certain date? Is that date realistic? Can certain features be postponed? Is success dependent on timing? How accurate are the estimates? Will vendors deliver their products as scheduled?
- ***Financial*** - Can you meet payroll with existing funds until the software makes enough new revenue to pay for itself? Will some high-up manager pull the plug on the project's budget (see Internal risk). Will the product be priced correctly?

- If you don't have time to do it right, will you ever have time to do it all over from scratch when it doesn't work?
- Can you simultaneously dictate the delivery date, team resources, code quality, and the feature set?

Software Risk

- For high Skill & Schedule risk, get more training.
  - Mistakes such as poor design generally cost more time and money than proactive training.
- For high Schedule risk, prioritize use cases.
  - With proper prior planning, the delivery date will be met, albeit maybe with less than 100% of the desired functionality.
  - Use cases should be prioritized first by architectural risk, then by functional importance.
- For high Business & Technical risk, make a prototype.
  - Can be throwaway, or part of an iterative lifecycle.
- Do the highest risk things first.

Training

- Project failure can be due to organizational and/or technical problems. Mistakes are costly. Training mitigates this risk.
- Invest in more training - include managers!
- As the number of people on a team increases, the number of communication links increases as N squared or worse; small, skilled (trained) teams are optimal.
- Send several people to this year’s Conference on Your Technology
- Conduct design and code reviews.
- Consider eXtreme Programming (pair programming / mentoring).
- Consider having an OO “Boot Camp.”
- Lots of new tools and technology... lots of training.
- Developers can easily get frustrated when forced to use new tools or technology. More training increases timeliness, quality, and morale.

Teams

Teams should be small (about 3-7 people):

- 1 P.M.
- 1 Architect (knows the technology and the big picture).
- 1+ Senior programmer(s) with appropriate skills.
- At least one “abstractionist” & one skilled UI designer.
- The rest: smart & motivated people who can work well with others and who know how to code; other specific skills (such as a DBA) are required, depending on the architecture.

Provide plenty of mentoring w/ the more senior team members.

Include dedicated testers and writers as needed.

Production support?

Schedules

- Schedules reflect what is known at the time they are created. Foresight is not 20/20. As you “shine the light into dark corners” new complexity is found.
- The schedule (work breakdown structure) should reflect the realities of software construction and testing. Therefore, some planning should be done to determine the order that modules ought to be built.
- People tend to underestimate the effort required for complex tasks. Here is a formula to use as a guideline (max time can be quite large):

  ```
  estimate = (min + 4 * bestGuess + 2 * max) / 7
  ```
- Consider planning poker for “relative estimation” - helps to avoid “anchoring” (psychological bias) created by pessimistic developers or anxious customers.
- The macro plan must account for interruptions, sick days, etcetera.
- Fixed schedules can be met if lower priority use cases are dropped.
- Flexible schedules are realistic, and scope can be variable, if properly managed.

Best Practices

- With “quick & dirty” software, the dirtiness remains, evolving into “slow & dirty” maintenance; strive for “clean” at the expense of “quick”.
- Create an environment where the developers can stay focused as much as possible. Provide developers with “thinking oriented” office space. Optimize people.
- Use source code control & a bug tracking system.
- Design & code reviews find bugs early, find different bugs than those found with testing, enforce design, coding & unit testing standards, and allow people to learn from each other. Review the requirements & architecture too. One way to “review constantly” is to have two programmers working together at one terminal.
- Bugs happen; they take the least time to triage and fix as soon as possible after they are written. Bugs should be fixed as soon as they are found, even if that delays getting “urgent” features done.
- Write lots of unit and functional tests; test early & often; automate regression tests.
- Use ECOs (Engineering Change Orders) to help manage change.
- Develop iteratively, model visually, and use a layered component architecture.

Worst Practices

- Wishful thinking, arbitrary delivery dates, and poor management can promote unrealistic schedule pressure.
- Giving in to unrealistic schedule pressure tends to shortchange important activities, such as considered design, documentation, unit testing of code, and reviews, even when it is exactly these activities that reduce bug rates, pull in the schedule during the home stretch & increase long term quality.
- Unrealistic schedule pressure hurts morale & software quality.
- Giving powerful tools to developers without adequate training increases developer frustration while decreasing productivity.
- Jumping right into implementation without first gathering requirements and considering the design/architecture is a recipe for inflexible, buggy code (assuming of course that it ever gets deployed at all).
- There is an ongoing effort to catalog Anti-Patterns … Analysis Paralysis, Creeping Featuritis, Manager Controls Process, Toss it Over the Wall, Mythical Man Month, Death March, Design by Committee, ...

Organizational Patterns

http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns?OrganizationalPatterns

“Not only should patterns help us understand existing organizations, but they should help us build new ones. A good set of organizational patterns helps to (indirectly) generate the right process.” - Jim Coplien

Jim Coplien has authored a “Generative Development Process Pattern Language.”

Here are a few of his patterns:

- Size the project. Size the schedule. Domain expertise in roles. Phasing it in. Apprentice. Organization follows location. Organization follows market. Developer controls process. Patron. Architect controls product. Architect also implements. Review the architecture.   Engage QA. Engage customers. Scenarios define problem.   Firewalls. Gatekeeper. Prototype.

Process Patterns

There are patterns in the software development *process*.

These patterns are complementary to organizational patterns.

For example:

- Requirements, Construction, and Delivery can repeat periodically;
- Within Construction, Design, Implement and Test can repeat periodically;
- Within Design, there are smaller scale patterns: Hold Reviews, Look For Reuse, Try To Generalize, Build UI Prototype, Clarify Requirements, Plan For Testing, etc...

The RAD Life Cycle might be considered to be a Process Anti-Pattern: Hack it together as fast as possible without doing considered design.

Prior Art

- The art of project management is in communicating the right information to the right people at the right time, deciding when to move on to the next activity, and in knowing what that activity ought to be, and more...
- The goal should not be “Rapid Development” per se, but rather, efficient and smart development, artfully balancing the need for low cost and the need for high quality.
- The patterns referenced on the preceding slides, and the numerous books written on this topic will help, because the master of any discipline should certainly study the prior art.