Session 15: Life-Cycle frameworks and more alternatives

Origins

Life cycle models (methodologies)
activity (procedures) driven
results (deliverables) driven

Reactions against LCM
requirements (ESD) preserving emerging requirements ("agile")
UML & others

Origins (~1963)

- A senior programmer would
  1. confer with a problem sponsor (high-level executive, functional manager, engineer, physicist, etc. who's paying for the project) to learn nature of problem to be solved.
  2. code & test rough implementation (might take a few hours or a week), show results to the problem sponsor.
  3. discuss what needs to be changed, refined, or added
  4. return to step 2 until sponsor is satisfied

  *What's wrong with that?*
  *What do we call that approach today?*

Drawbacks of incremental approach

- We often spent a lot of effort (time & money) before we knew whether the project is worth doing or even possible.
- We wasted effort on ideas that later got scrapped.
- Neither the problem sponsor nor the programmer had experience and knowledge about how to define clear & complete specifications.
- Resulting software not always a suitable basis for future change & enhancement.

A word that knowledgeable I.T. professionals avoid: *Waterfall*

- Term is used pejoratively by people who condemn the phased life-cycle (SDLC) and propose to replace it with their own project methodology.
- As far as we know, it was *never* used by any expert, textbook, or knowledgeable practitioner who favored or promoted a phased life-cycle.
**Waterfall** (continued)

- It implies an extremely rigid framework in which:
  - a. Once a phase is done and accepted, you can't change any of its results ("frozen specifications")
    
    *Water can't flow uphill!*
  - b. Before the current phase is finished, you may not work on any activity in the next phase.

- We know of no serious book, course, or packaged SDLC methodology that advocates such slavish inflexibility.

---

**Under a sensible phased Life Cycle (SDLC)**

a. You can indeed reconsider and revise material approved from an earlier phase.

  *How?*

b. You can indeed perform tasks from a later phase before you've finished the current phase.

  *Why?*

  c. *What are the risks? the benefits?*

---

**Under a sensible phased Life Cycle (SDLC)**

a. You can indeed reconsider and revise material approved from an earlier phase.

  *How?*

b. You can indeed perform tasks from a later phase before you've finished the current phase.

  *Why?*

  - Compress the schedule (a gamble)
  - You had a great idea and couldn't wait!

---

**Another word some I.T. professionals avoid: Agile**

- Term has been pre-empted by promoters of extreme programming, incremental development, emergent specifications, user stories, etc.

- It implies that the established (non-agile) approaches were stodgy, rigid, sluggish, etc.

- But isn't the very nature of most software development agile?

  - a. Practically everything a competent programmer does is *agile* in the usual dictionary sense of the word.
**Agile** (continued)

- The most extreme agilists want to set aside virtually all systematic discipline in their professional activities.
- More moderate agilists support disciplined approaches *in principle*, but may still want to ignore standards, procedures, conventions, guidelines, etc. that they find inconvenient at the moment.

---

**What is the "agile" approach to software development?**

- A generalization of "Extreme Programming" (XP) popularized around 1999 by Kent Beck, Ron Jeffries, et al
- A major project they were engaged to do for Chrysler Corp
  

  was the research/demonstration effort that led to publicizing XP and "agile" and generated widespread interest.

---

**One common "agile" misconception**

- Many presentations listing the various components of the agile approach cite unit testing as an important agile innovation!
- But programmers have been doing unit testing for decades before we heard of agile. (The ones who didn't *should* have.)

  *What was the one major exception?*

---

**The Chrysler project**

- A series of books, mostly Addison-Wesley, explained how this *modern, enlightened, and manageable approach* could revolutionize development of complex software, making it predictable and economical.
- Messrs. Beck, Jeffries, et al have built a strong reputation on their books & lectures. They produced the *Agile Manifesto* as a focal point.
- On the other hand *some* people in Chrysler considered the 7-year project a huge fiasco and colossal waste of time and money.

  *What was the true story?*
The Agile Manifesto

"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 & interactions over processes & 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."

Kent Beck    Mike Beedle    Arie van Bennekum
Alistair Cockburn    Ward Cunningham    Martin Fowler
James Grenning    Jim Highsmith    Andrew Hunt
Ron Jeffries    Jon Kern    Brian Marick
Robert C. Martin    Steve Mellor    Ken Schwaber
Jeff Sutherland    Dave Thomas

Some "Agile" flaws & shortcomings

1. Doesn't support projects that may lead to purchasing an application software product.
2. Provides little basis for determining whether a project can be justified (ROI).
3. Incremental deployment may conflict with transition/conversion from existing application.
4. Fragmented functionality (YAGNI)

Let's examine each of them.

Agile methodology doesn't support projects that may lead to purchasing an application software product.

How often do we encounter such projects?

In order to evaluate and eventually choose an application software product, we have to know exactly what it must do for us.

That means
- we need complete requirements; (may refer to vendor-documentation for report layouts, etc.)
- we don't need executable code fragments
- so there's no need for iterative development.

Agile methodology provides little basis for determining whether a project can be justified (ROI).

Agilists condemn the phased life cycle because "initial cost and time estimates are often inaccurate"!

But with iterative approach, there are no rigorous initial cost & time estimates.

So management has no basis to assess project justification: Return on what investment?
**Agile methodology may conflict with transition/conversion from existing application.**
- Delivering *working code* every "sprint" is nice, but whom will it actually work for?
- If we're developing a completely new application, then fragments of the eventual functionality may be useful to the end users.
- But if we're replacing an existing system, users may not be able to reconcile them.
  - Suppose the new application allocates stock from a different warehouse than the old system did?
  - Suppose it posts customer payments to a different database than the one now used to trigger dunning letters?
  - etc., etc., etc.

**Agile methodology leads to fragmented functionality (YAGNI)**
- Purpose of YAGNI ("you aren't going to need it") is to avoid wasting effort on fancy features that users may never use.
- But that's sometimes misinterpreted to justify omitting functionality that's essential to the very nature of a module (a function or a class).

What happens when a programmer contributes such a module to an organization's re-usable component library?

**Agile summary**
- We've shown that some aspects of so-called "agile" methodologies are in conflict with some aspects of enlightened application system development.

**Does that mean that we should avoid agile practices?**
- No! Many techniques promoted by some agilists are simply well-established good practice.
  - Strong example: thorough unit testing.
  - Weaker example: pair programming
  - Crackpot example: emergent specifications

**What an organization should do**
- Striving for CMM level 3 or better
  - Establish a repository for information about the organization's methodology.
  - Provide orientation courses supportive of that methodology.
  - Provide a Quality-Assurance / Review function to help the technical staff comply with the methodology.

**What's a methodology?**
**What's a repository?**
**Your organization's methodology should**
- Be drawn from informed judgment of the best practices.
  - Draw upon experts and fads when appropriate.
- Welcome and depend upon contributions from your professional staff.
- Avoid **guruism**
  - What's that?

**Guruism**
- Uncritical acceptance of one or more specific "experts" as the final authority.
  - "You can't do that; it's not *Kepner-Tregoe!*"  
    - *Yourdon*  
    - *Three amigos*  
    - *Agile manifesto*
- It's fine to respect well-known writers, lecturers, and consultants, but a good manager makes **well-informed choices**.

**Life-cycle concept summary**
- We agree *(do we really?)* that a phased life-cycle (not necessarily our exact example 7-phase model) is a sound basis for most non-trivial projects.
- But it **must** have a well-defined point that produces an ESD, and must not mix the **what** and the **how** of a proposed new application system.
  - Unfortunately the UP fails. *(How?)*
- What about packaged life-cycle methodologies that you can buy?
  - Some are results oriented
  - Some are activity oriented

**UML: where does it fit?**
- Acronym stands for **Uniform Modeling Language**.
  - What's uniform about it?
  - Where did it come from?
- Does it augment or replace other S.A. artifacts including:
  - Data dictionary?
  - Output specifications?
  - Data flow diagrams?
  - Entity-relationship diagrams
  - Algorithm / procedure specifications?
- **Details in COMP 320**
**Vital questions about UML for the Project Manager**

- Will the two audiences (*Who are they?*) be able to understand the ESD in full unambiguous detail?
- Why is it essential that they should?

**"Dinosaur" tools & techniques**

- Compared with other scientific and engineering disciplines computer technology is quite new.
- Therefore, some practitioners assume that any programming language, methodology, tool, or practice that has been around for 20 years or more must be obsolete. That sometimes leads to:
  - near-automatic embrace of anything new.
  - contemptuous rejection of anything from an earlier era as "dinosaur" approaches.
- Q: So, are 25-year-old I.T. tools & techniques hopelessly obsolete and worthless today?
- A: Some of them are and some of them aren't!

**Think!**

---

**Project failure as justification for a radical new approach**

- Presentations on some new approach often begin by asserting that an impartial survey shows Sixty percent (or some other appalling number) of projects following the "traditional approach" fail.
- Therefore, we must do something radically different in order to have a good chance of success!
- **What's wrong with that argument?**

---

**What does "failure" mean for a software development project?**

- Based on our example 7-phase life cycle:
  - a. Suppose we do phases 1 & 2 and then determine the project is either impossible or too expensive. The project is terminated.
  - b. Suppose phase 5 encounters very serious cost & time overruns based on faulty design and erroneous or incomplete user requirements. We eventually get the application installed and operational with some features missing or deferred.
- Which is the failure?