Abstract

How do you teach software project management to 4th year engineering students, when there is nothing to manage, and the largest project they’ve ever experienced was with two buddies last term? In this paper, we present our experience over seven years teaching this topic alternatively to industrial practitioners, and to undergraduate and graduate students in an academic environment, both in Canada and in the Netherlands. The approach is based on a conceptual model of software development that takes into account the common aspects across a vast spectrum of software projects (“the frog”): intent, product, work, people, time, uncertainty, quality cost and value, and the variability across this spectrum (“the octopus”): size, criticality, business model, governance, team distribution, culture, etc.. This conceptual model is used throughout to (1) structure the course, (2) introduce issues, techniques, practices, and analyze them from a critical perspective: what would the frog say? what would the octopus say? (3) map other models, frameworks, or standards in this field: PMBOK, ISO 12207, RUP, Agile and lean approaches, ACM/IEEE SE 2004 curriculum. Rather than delivering to the students a canned set of recipes, the objective is to allow them to reason about the strategies, techniques, practices and tools that are most applicable to a given set of circumstances. The approach is complemented by small simulation games used to illustrate a few aspects and to trigger discussion (what happened, how realistic is this, how would you do differently?), or short videos of practices used to initiate a debate in class on a given practice.

1 Introduction

From the RUP®, “software project management (SPM) is the art of balancing competing objectives, managing risks, and overcoming constraints to deliver a software product that meets the need of the customers (the ones that pay the bill) and the end users.” [1] (p.113) The mere fact that relatively few software projects are 100% successful is an indicator of the difficulty of the task. Most project management frameworks, and in particular the model underlying the PMBOK (Project Management Body of Knowledge) [2], have a C2 (command-and-control) philosophy: develop a plan, execute this plan, control the execution for divergence to the plan, apply corrective measures to get back to the plan. This had led historically to a overemphasis on planning, scheduling, process definition and process improvement, not much on collaboration.

“Software processes are software, too” said Lee Osterweil in 1987 [3], but this paradigm did not work too well because of the imperfect machines we use to “run” this process: teams of human beings, because of the very fluid and volatile nature of the environment in which most software product are targeted nowadays, and also because of the difficulty with defining work breakdown structures (WBS) in software, a concept upon which many traditional project management techniques and tools rely. The classical model of project management has been criticized: “the underlying theory of project management is obsolete” [4], and in the specific case of software project management, seriously put into question by the agile movement [5, 6, 7]. New approaches now take into consideration the fluidity of the environment, the iterative nature of the work, and the communication and collaboration between individuals.

But how do you teach this? Many approaches are very practical, hands-on, and based on successful recipes, or well-established tools. “This is how to do planning, this is how to do scheduling, this is how to organize a team, this is how to allocate tasks, etc.”

There is a whole course at UBC organized around the 1,001 ways to use Microsoft Project®.

While there is some value for our students to master a certain number of tools and techniques, there are serious limitations to this approach. The main limitation is the vast spectrum of type of software development projects, in size, in criticality, in...
complexity, and the nice tools and techniques may not all work well in all these different contexts. Many software project management approaches fall into the same trap as the obsolete ones they claim to successfully replace: a narrow-minded dogmatism [8]. “This is the ‘agile way’, made simple for you”, and often presented in absence of any context [9], for some “average” project. Another more subtle limitation is that the recipe book approach does not allow much for critical thinking, reflection, adaptation and innovation.

In this paper, we offer more neutral conceptual model of software development projects, more encompassing and possibly more unifying than the model underlying that of the PMBOK, for example. Our model addresses both the common aspects (the “frog” part of the model) and the very wide variability (the “octopus” part). We then show how the model is used in building a software project management (SPM) course, to structure it, and to articulate the use and value of this or that approach technique or tool, find the limits of their applicability in various situations. Our conceptual model has also been mapped to other “theories” and frameworks.

2 A Conceptual Model of Software Project

The conceptual model is organized in two orthogonal parts: the frog and the octopus.

2.1 The Frog

This part of the model of software projects is organized around 8 concepts that are universal across all software projects, and their relationships (cf. fig. 1). Indeed:

- **Intent**: The concept of Intent denotes what the project is trying to achieve. The Intent defines the scope of the project, the intentions and hopes of the key stakeholders, the objectives. While we think of the intent as “the requirements” or “the specification”, in practice Intent may take many diverse forms: a set of tests that the product must pass contributes to defining Intent. A set of software problem reports (i.e., defects) that must be dealt with also indirectly defines Intent. A set of acceptance tests also define Intent. Various constraints, implicit or explicit, internal or external to the project will affect Intent. And one constant of software projects is that they are under pressure of a stream of change requests which modify the Intent.

- **Product**: The concept of Product denotes the outcome of the project, what has been achieved. This is the actual software, accompanied with any other artefacts that are needed to make it a complete product: an installer, a set of data, the user’s guide, some training material, etc. Why aren’t Intent and Product more or less equivalent? Why do we need to distinguish them in our model? Intent precedes Product: Intent is an abstraction, a virtuality that sketches the reality that the project is set to achieve. Even when the product is “done” there may be discrepancies between the Intent and the Product; the Intent may have evolved in the meantime, or the Product has come short in some ways of the original Intent. These discrepancies between Intent and Product are the key drivers for the project; they are the imbalance that makes it run. Imagine the relationship between Intent and Product as a bungee cord: the further apart and the more energy the project will expend to bring them closer.

- **Work**: The concept of Work denotes the activities, tasks, steps that need to be accomplished in order to turn Intent into Product. They often come defined by a process, or a method, which attempts to describe a systematic way to build a product; some elements of Work are defined “on-the-fly” in an ad hoc fashion. In many cases, a Work item produces or refines some artefact: a document, a model, an idea, a piece of code, a report, formal or informal. Some of these artefacts are only useful internally to the project, as stepping-stones, and do not appear in any form in the final product; they are not “deliverables”.

- **People**: The concept of People is important in modern software project management because they are the
main “engine” behind Work elements. Software development is an intellectual activity that is very ‘human-intensive’. Most of the work elements are done by human beings, and only little of this work can be automated. So the availability and the competence of the people are keys to get all the work done. And nowadays most of the cost of software development is associated with people, for most cases.

2.1.2 Three fundamental attributes
Each of these four core concepts has three attributes: Time, Quality and Risk/uncertainty.

5. Time
The concept of Time is orthogonal to our fundamental quadruple [Intent, Work, People, Product]. Often we will use the phrase life cycle to denote what happens with a project over time. It is tempting to define a project linearly relative to time in 5 main steps: 1) define completely the Intent, 2) derive from the Intent all the Work that needs to be accomplished, 3) allocate work to People, and 4) People build the Product, 5) which acceptance testing will show that it matches exactly the original Intent. This has been tried again and again, but with very meagre successes in software development for a range of reasons. In reality, we define Intent gradually, and it tends to evolve throughout the project under various pressures and demands for changes. We can therefore only define part of the Work at any point in time, and allocate it to People, who will therefore only build part of a Product. This partial product will influence back the Intent, through user feedback, or problem reports. It will also influence how people will conduct the work in the future. Other chunks of Intent are then carved out, more Work defined, and the Product will evolve until it reaches a deliverable stage. All modern software development approaches are iterative and incremental, and they define a project as a sequence over time:

{ [Intent₁, Work₁, People₁, Product₁],  
  [Intent₂, Work₂, People₂, Product₂],  
  ...  
  [Intentₙ, Workₙ, Peopleₙ, Productₙ] }  
where Productₙ is the final ‘deliverable’.

6. Quality
The concept of Quality is also an orthogonal notion to our fundamental quadruple [Intent, Work, People, Product]. We can see quality as an attribute of each of them. Quality of the Intent denotes how good we are at defining and planning a Product. Quality of the Work denotes the quality of the process we use to develop software and all the intermediate artefacts. Quality of the People denotes the competence and diligence and dedication of the staff assigned to the project, and finally quality of the product is a measure of how close to the expectation of the stakeholders the delivered product is. These four aspects of quality may evolve over time, following the sequence we described above, and hopefully their quality increases over time (inasmuch as quality is quantifiable).

7. Risk and uncertainty
The fundamental concept of Risk denotes the uncertainty that is associated to each of the four fundamental concepts at some point in time: uncertainty in the Intent, because the domain is new, for example, uncertainty in the Work to be performed, because the process is unclear, risks associated with the people and therefore uncertainties in the final Product. Similarly to Quality above, these uncertainties evolve over time: the risks will be mitigated, unknowns will become known, but new risks keep emerging.

8. Value and 9. Cost
Finally, Value is associated with Intent and Product: we need to assign expected value to the Intent to guide development of the Product over time, while the Cost of the development is associated with the Work and the People. As software is essentially an intellectual, human-intensive activity, costs are directly derived of the cost of the People associated with the project and the Work they do: what, how much, for how long. This is a characteristic of software development not shared with other engineering disciplines, such as civil engineering.

Altogether a software development Project is all the work that people have to accomplished over time to realize in a product some specific intent, at some level of quality, delivering value to the business at a given cost.

2.2 Factors shaping software development projects
There are 2 sets of factors that make up the context: factors that apply at the level of whole organization/company, and factors that apply at the level of the individual project (cf. fig. 2). In small organizations, with few software development projects, this distinction does not apply, and all factors are on the same level [10].

2.2.1 Organization-level factors
The organization-level factors (environment conditions) do influence heavily the project-level factors, which in turn should drive the process and practices that should be used.
1. **Business domain**
For what domain of activity is this organization developing software? Web-based systems, aerospace embedded systems, small hand-held instrumentation?

2. **Number of instances**
How many instances of the software system (large or small) will be actually deployed? Are you building one system, a dozen, a thousand, or millions?

3. **Maturity of organization**
How long has that organization been developing software? How mature are the processes (and the people) relative to software development?

4. **Level of innovation**
How innovative is the organization? Creators or early adopters of new ideas and technologies? Or treading on very traditional grounds?

5. **Culture**
In which culture are the projects immersed, both national culture and corporate culture? What are the systems of values, beliefs and behaviours that will impact, support or interplay with the software development practices? Ethics is part of it.

### 2.2.2 Project-level dimensions - The octopus

1. **Size**
The overall size of the system under development is by far the greatest factor, as it will drive in turn the size of the team, the number of teams, the needs for communication and coordination between teams, the impact of changes, etc. Number of person-months, or size of the code, or development budget are all possible proxies for the size.

2. **Stable architecture**
Is there an implicit, obvious, de facto architecture already in place at the start of the project? Most projects are not novel enough to require a lot of architectural effort. They follow commonly accepted patterns in their respective domain. Many of the key architectural decisions are done in the first few days, by choice of middleware, operating system, languages, etc. In chapter 10, we will revisit the concept of software architecture and its relationship with software project management.

3. **Business model (finance)**
What is the money flow? Are you developing an internal system to support your internal processes, or a commercial product, or a bespoke system on contract for a customer, or maybe a component of a large system involving many different parties? Are you contributing to a free-libre open-source (FLOSS) project? How the various participants are ultimately compensated for their effort will also shape the development and management process.

4. **Team distribution**
Linked sometimes to the size of the project, how many teams are involved and are not collocated? This increases the need for more explicit communication and coordination of decisions, as well as more stable interfaces between teams, and between the software components that they are responsible for.

5. **Rate of change**
Though agile methods are “embracing changes”, not all domains and system experience a very rapid pace of change in their environment: Intent volatility. How stable is your business environment and how much risks and uncertainties are you facing?

6. **Age of system**
Are we looking at the evolution of a large legacy system, bringing in turn many hidden assumptions regarding the architecture, or the creation of a new system with fewer constraints, a ‘greenfield’ project?

7. **Criticality**
How many people die or are hurt if the system fails? Documentation needs increase dramatically to satisfy external agencies who will want to make sure the safety of the public is assured.

8. **Governance**
How are projects started, terminated? Who decide what happens when things go wrong? How is success or failure defined? Who manages the software project manager? Are there external rules and regulations imposed to the product?

![Figure 2. Variability across software development projects (the octopus)](image)
This model of process variability is similar to Scott Ambler’s Agile Scaling Model [11], or the parameters of Alistair Cockburn’s Crystal methods [12].

3 Structure of the course

The course is then organized mainly around the elements of the conceptual model. It does not follow the waterfall lifecycle, nor Deming’s PDCA (Plan Do Check Act), nor the PMBOK higher level structure as our model does not assume a specific sequence of activities, and can accommodate any form of lifecycle [13]. The sequence of lessons is:

1. Introduction
2. A model of software development (the frog)
3. Contexts of software development (the octopus)
4. Process and management frameworks & standards
5. Managing risk and uncertainty
6. Managing time: macro-level - lifecycle, estimation
7. Managing time: micro-level - scheduling
8. Managing quality
9. Managing objectives and scope - intent
10. Managing complexity
11. Managing changes
12. Managing software assets
13. Managing software products
14. Managing people: individual level
15. Managing people: team level
16. Managing external stakeholders
17. Managing the software process - work
18. Software development governance
19. Large projects, programs, and project portfolios
20. Conclusion

4 Analysis of practices, techniques & tools

As we progress through the course, we can introduce techniques and tools used to manage the project. as we introduce them we can articulate what they achieve, functionally, for the software project manager:

- Visualize the Intent
- Map People onto Work
- Show responsibility of People onto Product
- Assess the Quality of the Intent
- Assess the Risk associated with Intent

Question: “What would the frog say?”

We can take for example the practice of Daily Stand Up meetings from Scrum [13]. It is mostly focussed around people and the mental models they have of both where the project is going, and how it is going there. Is does not scale well beyond a dozen people, and is made very difficult in distributed environment, especially over multiple time-zones. Business model and Cultural issues may hinder it (subcontracting, power distance, “losing face”).

Similarly, we can analyze a Gantt chart. It shows primarily how Work is spread over Time. In some forms, it can also indicate the mapping of People onto Work (task allocation). Supported by tools, it scales well, but its applicability is limited by the constant changes and the absence of clear, definite work breakdown structures (WBS) in software development.

Going further, we can describe a Kanban board, [14] and then use our framework to systematically compare it with a Gantt chart, both in terms of functionality, how it is achieved (the frog view), and in terms of applicability to various contexts (the octopus view).

5 Lessons learned

When teaching SPM to people who have experience of software development, as project manager or any other role, we can draw form this experience to analyse, evaluate, contrast, discuss the role and value of various techniques. But when there is no real experience to draw from, the students see this as a mere collection of recipes to follow, with few opportunities to put them in practice. Contrasting “old” or “traditional” (meaning often: bad) approaches to “new” or “agile and lean” approaches proved also to be vain, as students had not experienced the “old”, and as the old approaches may still be valid in some context, and good to know in general. The development of a conceptual model was therefore key in that it allowed to analyse and reason about any software project management techniques, old or new. To our surprise, the conceptual model proved to be also as useful when teaching SPM to seasoned practitioners, in particular in explaining why the new, agile or lean approaches bring value, and what are the limits of their validity. And the entity People is more prominent and integrated in this model than other models.

A large part of learning comes from doing, from experiencing it first hand. In academic environments, this is often achieved by term or capstone projects. This does not work too well for software project management for several reasons: 1) scale: you cannot
have projects large enough and long enough to experience much, not everyone on the team can be project manager, and such projects often turn into massive and heroic amount of coding, with little focus on management. In an industrial environment, the timeframe usually does not allow for any form of projects.

We found the analysis of cases and simulation games to be a useful complement. We have developed 2 simple games for this purpose:

- **Hard Choices** [15] developed with the Software Engineering Institute introduces risk management, decisions making and dealing with the consequences, such as delays or technical debt.
- **Mission to Mars** [16] is a release planning game, where players plan, run and re-plan again and again the release of a project over several iterations, dealing with issues of productivity, defects, dependencies, etc.

We show situations on movies and then ask the students to analyse and critique. We have used extracts from existing movies, such as the hilarious Meetings, bloody meetings (with John Cleese) [17], or shorter "skits", such as our Daily stand-up from hell [18].

As a result of using explicitly the conceptual model, the course can now be like a Socratic debate, (elenchus) and an opportunity for practising critical thinking. Based on this theory, to answer a question or solve a problem within a certain context, we can build arguments and we can make inference and judgments. We can derive more assignments and exam questions!

Finally the conceptual models maps to other SPM models [1, 2, 13], and in particular this was useful to build software engineering programs and their syllabus [19, 20] (see more detailed mappings in [21] table 1).

Note: the material for the course described here is available to qualified instructors under a Creative Common Attribution-Non commercial 2.5 License

References

[18] The daily stand up meeting from hell, (video 2.5') 2010 Vancouver: KESL. At www.youtube.com/watch?v=VjNxQ-a-x2M