Software Project Management

Vladan Devedzic

FON - School of Business Administration, University of Belgrade, Yugoslavia

In order to organize and manage a software development project successfully, one must combine specific knowledge, skills, efforts, experience, capabilities, and even intuition. They are all necessary in order to be able answer questions such as: What artifacts to manage and control during software development? How to organize the development team? What are the indicators and measures of the product's quality? How to employ a certain set of development practices? How to transition a software development organization to a new modeling and/or development paradigm? How to create and maintain a good relationship with the customers and end-users? What remedial actions to take when something goes wrong in the course of the project? What are the heuristics that can help managers in conducting the software development process?

The manager of a software development project should answer the above questions in the context of the project itself. However, there is a vast amount of knowledge the manager should possess that transcends the boundaries of any specific project.

The purpose of this chapter is to provide an extended overview of many important issues around which such knowledge should be structured. The introductory section merely introduces the issues and the context within which the other sections discuss them. Each of the remaining sections covers one of the issues in more detail. The idea has been to provide a balanced coverage of the issues from both the manager's and the developer's perspectives.

Keywords. Software development process, management practices, metrics, organizational aspects, standards.

Introduction

Software development is a complex process involving such activities as domain analysis, requirements specification, communication with the customers and end-users, designing and producing different artifacts, adopting new paradigms and technologies, evaluating and testing software products, installing and

maintaining the application at the end-user's site, providing customer support, organizing end-user's training, envisioning potential upgrades and negotiating about them with the customers, and many more.

In order to keep everything under control, eliminate delays, always stay within the budget, and prevent project runaways, i.e. situations in which cost and time exceed what was planned, software project managers must exercise control and guidance over the development team throughout the project's lifecycle [1]. In doing so, they apply a number of tools of both economic and managerial nature. The first category of tools includes budgeting, periodic budget monitoring, user chargeback mechanism, continuous cost/benefit analysis, and budget deviation analysis. The managerial toolbox includes both long-range and short-term planning, schedule monitoring, feasibility analysis, software quality assurance, organizing project steering committees, and the like.

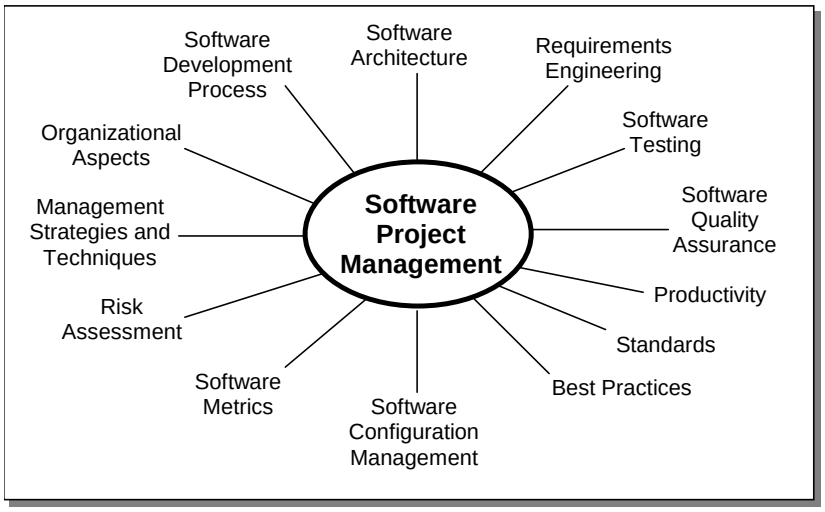

Figure 1 - Some important issues of software project management

All of these activities and tools help manage a number of important issues in the process of software development. Figure 1 illustrates some of the issues, but definitely not all of them. The issues shown in Figure 1 have been selected for an extended overview in the remainder of this chapter based on the following criteria:

- their priority in the concerns of most software project managers, according to the managers themselves - this is evident from the case studies, interviews, and reports of many software project managers and consultants in software industry worldwide (see, for example, [12], [20], and [49]);
- the frequency of their appearing as topics in the relevant, industry-oriented software engineering journals and magazines, such as *IEEE*

Computer, *IEEE Software*, and *Communications of the ACM*, during the last decade;
- their importance as identified by relevant committees, associations, and consortia of software developers (see, for example, [26]).

The chapter does not address the economic aspects of software project management, such as budgeting, negotiating, outsourcing, and contracts. The goal is to consider some of the important managerial issues specific to *software* development, not those that appear in other kinds of development projects as well.

Software Development Process

One of the primary duties of the manager of a software development project is to ensure that all of the project activities follow a certain predefined *process*, i.e. that the activities are organized as a series of actions conducing to a desirable end [33]. The activities are usually organized in distinct *phases*, and the process specifies what artifacts should be developed and delivered in each phase. For a software development team, conforming to a certain process means complying with an appropriate *order* of actions or operations. For the project manager, the process provides means for control and guidance of the individual team members and the team as a whole, as it offers criteria for tracing and evaluation of the project's deliverables and activities.

Software development process encompasses many different tasks, such as domain analysis and development planning, requirements specification, software design, implementation and testing, as well as software maintenance. Hence it is no surprise at all that a number of software development processes exist. Generally, processes vary with the project's goals (such as time to market, minimum cost, higher quality and customer satisfaction), available resources (e.g., the company's size, the number, knowledge, and experience of people -- both engineers and support personnel -- and hardware resources), and application domain.

However, every software developer and manager should note that processes are *very* important. It is absolutely necessary to follow a certain predefined process in software development. It helps developers understand, evaluate, control, learn, communicate, improve, predict, and certify their work. Since processes vary with the project's size, goals, and resources, as well as the level at which they are applied (e.g., the organization level, the team level, or the individual level), it is always important to define, measure, analyze, assess, compare, document, and change different processes.

There are several well-known examples of software development processes. Each process relies on a certain *model* of software development. The first well-established and well-documented software development process has followed the *waterfall model*. One of its variants is shown in Figure 2. The model assumes

that the process of software development proceeds through several phases in a more-or-less linear manner. The phases indicated in Figure 2 are supposed to be relatively independent. There is not much feedback and returning to previous phases other than the one directly preceding the phase in focus. In other words, once a certain phase is finished it is considered closed, and the work proceeds with the next phase. Many developers have criticized the waterfall model for its rigidity in that sense, and for its failure to comply with the reality of ever-changing requirements and technology. However, the waterfall model is at least partially present in most of the other models as well, simply because of its natural order of phases in software development.

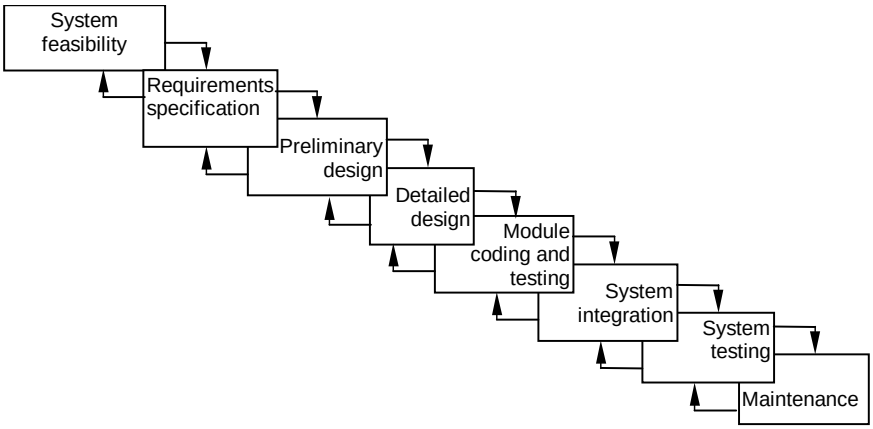

Figure 2 - The waterfall model of software development (based on [5])

There have been many attempts to overcome the limitations of the waterfall model. Two common points in all such attempts are introduction of *iterations* in software development activities and *incremental* development. Iterative and incremental software development means going through the same activities more than once, throughout the product's lifecycle, each time producing new deliverables and/or improving the old ones. The main advantage of working in that way is that each individual developer works on a small ``work packet'' at any given moment, which is much easier to control.

A classical example of iterative and incremental models is the *spiral model* [9], sketched in Figure 3. In the spiral model, there are five core tasks: planning and design (largely corresponding to the classical analysis phase), approval (requirements specification), realization (design and implementation), revision (testing and modification), and evaluation (integration and system-level testing). The process iterates through these tasks, getting closer and closer to the end by adding increments (e.g., new functions, new design, new modules, new or improved testing procedures, new or improved parts of the user interface, new integration and testing certificates, and so on) to the product in each iteration. The spiral model underlies many processes, such as DBWA (Design By

Walking Around) [50], and PADRE (Plan-Approve-Do-Review-Evaluate) [41]. The DBWA process combines the spiral model with multiple design views, flexible structuring of development teams, and dynamic changes in modes of working (e.g., working individually, working in pairs, or working in small teams), in order to improve the process efficiency and parallelism. The PADRE process uses the spiral model at multiple levels - the project level, the phase level, and the individual software module level - thus creating the ``spiral in a spiral in a spiral'' effect.

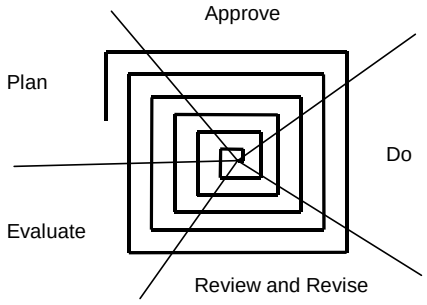

Figure 3 - The spiral model of software development (after [9] and [41])

The *JNIDS model* (*Joint National Intelligence Development Staff*) [5] is similar to the spiral model in that it is also iterative and incremental. There are, however, six tasks in the JNIDS model: requirements analysis, team orchestration (i.e. the team-building stages, ``forming, storming, norming, and performing''), design, coding, integration, and system implementation (delivery and maintenance). The model prescribes to iterate through all six tasks in every phase of software development. There are five phases (requirements identification, prototype development, the breadth of system functionality, system functionality refinement, and transition). They differ in the amount of time and effort they dedicate to each specific task. The first phase focuses most on requirements analysis, the second one focuses most on team orchestration, and so on. The last phase is concentrated most on integration and maintenance. Hence on the time axis the shift of the focus of attention in different phases generates a waterfall-like shape if the six tasks are put on the ordinal axis. However, an important difference between the classical waterfall and JNIDS models is that in the JNIDS model developers conduct their activities through *all* tasks in *each* phase.

The *Unified Process* for object-oriented software development [24], Figure 4, has recently become very popular. It is also iterative and incremental, just like the spiral and JNIDS models. All of its iterations go through five core workflows (tasks) shown in Figure 4, and are grouped in four phases - inception (resulting in a global vision of the software product), elaboration (detailed analysis and design of the baseline architecture), construction (building the system's initial capability), and transition (product release). Just like in the JNIDS model, Figure 4 shows ``fuzzified'' traces of the waterfall model in the Unified Process. The

process is architecture-centric, meaning that its main deliverable is an executable architecture (the system), described by a set of models generated during the system development (use-case model, analysis model, design model, deployment model, implementation model, and test model). The models are represented using the standard UML diagrams [13]. The Unified Process is also use-case oriented, which means that generic scenarios of how the user or external applications use the system or its subsystems bind all the workflows and drive the iterations.

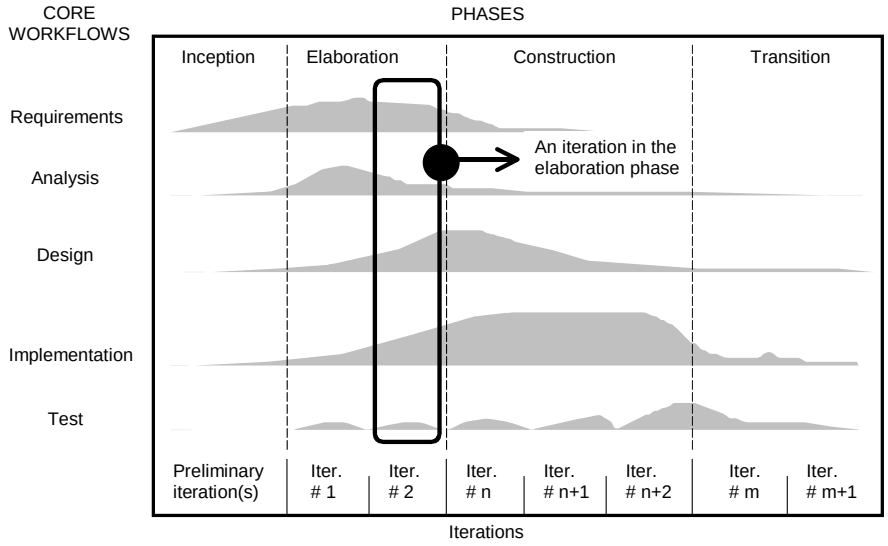

Figure 4 - Core workflows, phases, and iterations in the Unified Process of object-oriented software development (after [24])

Being iterative, the Unified Process reduces additional development costs generated by unexpected situations (usually just a single iteration of work is lost). Iterating through all core workflows in every iteration, the process is compliant with the reality of ever changing and incomplete user requirements. The Unified Process is also risk-driven - it enforces examining areas of highest risk in every phase and every iteration, as well as doing the most critical tasks first. Hence it minimizes the risk of project runaways. Managers can easily adapt the Unified Process to different application types, project sizes, development teams, and levels of competence.

Because of the importance of the Unified Process for software project management today, comments on some other issues from the Unified Process perspective are included in the following chapters.

Requirements Engineering

Requirements engineering is the discipline of gathering, analyzing, and formally specifying the user's needs, in order to use them as analysis components when developing a software system [18], [44]. Requirements *must* be oriented towards the user's real needs, not towards the development team and the project managers.

Almost all software development processes one way or another stress requirements analysis and specification as one of their core workflows. The reasons are simple. It is necessary to manage requirements as well as possible because a small change to requirements can profoundly affect the project's cost and schedule, since their definition underlies all design and implementation [40]. Unfortunately, in most practical projects it is not possible to freeze the requirements at the beginning of the project and not to change them. Requirements develop over time, and their development is a learning process, rather than a gathering one. The intended result of this process is a structured but evolving set of agreed, well understood, and carefully documented requirements [25]. This implies the need for requirements *traceability*, i.e. the ability to describe and follow the life of a requirement, in both a forward and backward direction, ideally through the whole system's life cycle.

The importance of constantly involving the users in the process of requirements analysis and specifications cannot be overemphasized. Only the users know their domain properly, and for that reason they should certainly participate in defining the system's functions, designing them, and evaluating their implementation and testing. The users should also participate in creating, verifying, and updating the requirements specification document for the project. The users should share with the developers the responsibility for the requirements' completeness and consistency. It is the project managers' duty to establish and maintain good relations with the users throughout the development process, as well as to consult them whenever the project gets stuck due to the development team's lack of domain understanding.

It is essential to make as explicit as possible all the requirements that reflect the user's work and the tasks that the software system under development is supposed to automate. Any situation in which users can find themselves when doing their job is the context that must be taken into account through requirements engineering. It is equally important not to concentrate on a single user's task, but to cover communication between users when the task requires collaboration.

There is a wide spectrum of techniques for requirements engineering. Whatever technique is applied, it is always desirable to involve the user to increase the correctness of the requirements specification. Some of the techniques are:

- structured interviews and questionnaires that the user fills in (inquiry-based requirements gathering);

- diagram-based requirements analysis (using multiple diagrams to sketch relevant parts of the user's work process and describe the requirements graphically);
- using metaphors of the user's work process (e.g., the office metaphor, or the agent/agency metaphor);
- scenario analysis (scenario is a typical sequence of activities characterizing the user's work process, hence it reflects what the user will do with the system and helps define the test procedures);
- using special-purpose software tools for requirements gathering (some of them can be simulation-based);
- requirements completeness and consistency checks (some of them can be automated, others must be performed manually);
- using special-purpose requirements-specification languages in order to describe requirements more formally and hence provide more automated requirements tracing;
- prototype system development, in order to make the requirements clear and to establish better mutual understanding with the users;
- analyzing videotaped user's work process.

When managing software development according to the Unified Process, requirements are captured mostly through the use cases and use-case diagrams. A use case can be described as a specific way of using the system from a user's (actor's) perspective [13]. Use-case diagrams graphically depict system behavior (use cases), i.e. a high level view of how the system is used as viewed from an outsider's (actor's) perspective [24].

Software Architecture

Software architecture encompasses specification and design of the application's global structure, leaving the details aside [42]. It is related to the general software organization in terms of its *components* and *connectors*. Components are things like modules, compilation units, objects, and files. Connectors define interactions among components through procedure calls, parameters of initialization, instructions for the linker, and so on.

Defining the architecture of a software system involves the choice of *architectural style*. Each architectural style defines a family of software systems organized in a similar way, the corresponding vocabulary of components and connectors, constraints in using the components and connectors in building the system according to that style, and the way the overall system behavior depends on the behavior of its components. Examples of software architectural styles include layered architectures, pipeline architecture, object-oriented architecture, event-based architecture, repository-based architecture, component-based systems (CBS) architectures, process-control architectures, real-time architectures, and various heterogeneous and Internet-based architectures.

As soon as the initial set of requirements is gathered, the project manager should direct the chief architect and some other engineers to define the initial software architecture of the system to be developed. Software architecture definition does not get sealed after the project begins. On the contrary, it is an evolving activity that continues through all the phases of the product's lifecycle. It interweaves with requirements specification, domain analysis, study of possibilities for reuse, and even design.

The Unified Process of software development treats the problem of software architecture definition and evolution as its central activity [24]. Software architecture is described by different views of the system being built, and these views are in turn represented by different UML models and diagrams.

One of the ultimate goals of the Unified Process is to produce and continuously evolve the system's *executable* architecture. Even in the project's early phases a limited version of the executable architecture must be developed and demonstrated. At the project's closure, it is important for the system architecture to support all of the use-cases specified through requirements engineering. However, in the early phases the architects should capture just a rough outline of the architecture. It is not viable to base that outline on all of the use cases. For that reason, architects first select the general architectural style in a use-case-independent way, and then refine it taking into account only a small subset of all the use-cases for the system. That small subset usually contains no more than 5-10% of the whole set, but all the use-cases in the subset represent the key functions of the system under development. As the use cases are further specified and they mature, more of the architecture is discovered, and more details are introduced. This process continues until the architecture stabilizes.

Selecting an architectural style and evolving the software architecture is far from being simple, because it involves many issues other than just the system's overall structure. Project managers must be aware of such issues. The issues include the platform the system is to run on (e.g., the hardware architecture, operating system, database management system, and network protocols), global control structures, data communication, synchronization, and access protocols, reusable building blocks available, deployment considerations, legacy systems, the choice among multiple design alternatives, assignment of functions to modules and subsystems, the system's functionality, scalability, reliability, and usability, its comprehension and resilience to changes, and also its esthetical considerations.

For that reason, the architecture of a software system is considered a product in its own right, along with the main software product to be delivered to the customers. It is the development team that benefits most from the software architecture as a product.

Organizational Aspects

Software project management always involves various organizational aspects, such as creating and staffing development teams, assigning roles to the team members, modalities of software development, leadership considerations, interpersonal communication at work, staff training and embracing new technologies, organization's culture, social and ethical issues, and so on [3], [4], [12], [16]. Organizational aspects of software development are crucial for all successful projects. They are neither about hardware nor about software - they are about ``peopleware'', i.e. about using, coordinating, and managing human resources in an organization effectively. In the context of fast-paced and extremely fluid dynamics of software industry, the key to success or failure of software project is the way it is organized and managed.

The foundation of all organizational aspects in software development are the general principles shown in Figure 5, as suggested in [16]. Experience shows that the four principles in the corners of the square in Figure 5 constitute the roots of most software organizations' cultures, hence it is important for project managers to fully understand their meanings and importance. The principles are not mutually exclusive, and they all have advantages and disadvantages. Adopting any one of them in a new software organization does not guarantee success and does not necessarily result in projects' failure.

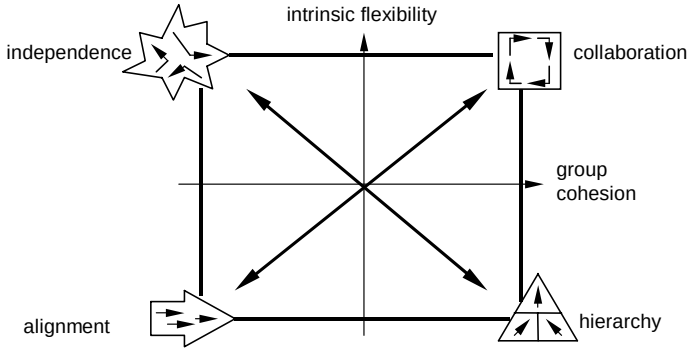

Figure 5 - General principles of software organizations (after [16])

The principle of *hierarchy* implies a strict pyramid of leadership, roles, duties, and tasks in the organization, with strict adherence to the organization's internal rules. Drifting away from the predefined overall course of the organization is interpreted as a lack of loyality that may lead to the organization's instability, and is not tolerated.

On the opposite end of the same ``dimension'' is the principle of *independence*. It relies on the individuals' initiative and individuality in doing their jobs, directing their work, and to an extent even in decision making. Organizations that adopt this principle are usually open to innovation and changes, new technologies, and creative autonomy of their members.

Collaboration is the principle of intensive teamwork, in which invention is combined with stability, and individuality with collective interests through open discussion about all the problems that come along as the project advances. Roles and tasks are assigned flexibly, and there is a high degree of adaptivity among the team members due to the intensive information exchange between them.

Finally, the principle of *alignment* suggests referring each individual in an organization to the organization's common vision of collective goals and adopted technology. Each individual is expected to support the adopted uniform approach to software development and comply with the work of others in the organization.

Of course, there are many points in between the two extremes along each dimension; hence most software organizations can be actually represented as points somewhere in the square of Figure 5. It is also important to note in Figure 5 how the cohesion of the development team increases with the hierarchy and collaboration principles, as well as how the flexibility of the organization increases with independence and collaboration principles.

The four principles can affect a number of decisions in improving a software organization, development processes, and project management. For example, creating the development team for a software development project, staffing it, assigning roles to the team members, as well as selecting the project leader(s) can all vary to a notable extent depending on the organization's dominant principle. If the principle of hierarchy is the dominant one, well-suited team members are those that prefer precisely defined tasks, roles, and guidance, and the team leader should be a person of high authority who sets precise criteria and expects results. Under the principle of independence, a free-style, informal, charismatic leader is much more likely to create and lead the team successfully. The team members in that case should be independent individuals, who don't need guidance, and are always ready for initiative and open to changes. Similar suggestions exist for applying the principles of collaboration and alignment.

Put in other words, it is an appropriate *professional culture* that a software development organization must grow in order to be able to manage all of its organizational aspects successfully, both on the long-term and day-to-day bases [3]. A culture is established in an organization when its software engineers internalize the organization’s professional values and common processes. Engineers should define these common processes from the practices they trust. Developers adopt and adhere to a professional discipline that orients them to add, modify, improve, or improvise such practices in order to achieve project objectives. Once people in the organization have internalized common practices, they transmit the culture through their behaviors, artifacts, and mentoring of others. An organization that lacks repeatable management or development practices does not have a professional culture. It is the responsibility of the organization's executives to enforce the above common principles or core beliefs that will help create, staff, orient and support development teams in practical projects.

There are efforts devoted to the institutionalization of repeatable common practices in software development organizations. Software Engineering Institute's *Capability Maturity Model* (*CMM*) [36], [45] is the most widely known among such efforts. CMM provides a reference for determining the maturity level of an organization's software processes and deals with the establishment of management structures that are to facilitate the development of the required professional culture. According to CMM, an organization can be at any one of five predefined maturity levels, in terms of its commitment to systematically perform the specific set of key practices that have been defined for a given level. At Level 1, the organization is typically chaotic and unmanaged, does not practice any uniform approach to the process of software development, budgets are always overrun, the products are unstable, and never meet the user's expectations. At Level 2, processes in the organization are controlled and maintained in detail, hence are repeatable. At Level 3, the organization contains a coherent, integrated set of well-defined software engineering and management processes, characterized by readiness criteria, completion criteria, inputs, outputs, and verification mechanisms. At Level 4, the organization formulates and strictly applies explicit assessment and feedback mechanisms, and uses them to measure efficiency and effectiveness in software development. Finally, the highest point in evolution is Level 5, at which the organization is capable of using the outcome data from its own processes for further self-improvement.

Some of the practices an organization may follow when establishing its professional discipline and culture are the following ([1], [3], [4], [12], [27], [20], [21], and [49]):

- flexible rotation of key roles and tasks (e.g., system design, integration, testing, code inspection) in the project team;
- maintaining a catalogue that specifies all key roles and responsibilities;
- the role of the project leader(s) is focused mostly on project management and system design, and much less on coding and testing;
- the project leader should have a small board of project advisors who set the project goals, prepare status reports on intermediate stages of development, care about resource management, monitor the project dynamics and stability, and suggest how to better staff the project;
- decision making should be based on consensus among the team members whenever possible; failing to allow for that may cause lacking of the sense of contribution to the project ultimate goal among those team members whose voice is ``never heard'';
- project manager's awareness of the ``staffing profile'' corresponding to the development process being applied is very important; for example, in object-oriented development processes it is necessary to have a chief architect throughout the development project; however, the number and efforts of system analysts and application engineers necessary to complete

the tasks is rather small in the early phases, and increases significantly in the later phases of the project;

- planning transition to another development process or another technology should include an appropriate training of the development team members, mentoring and guidance among the team members in embracing the new technology, careful selection of new programming languages and tools, and strategic decisions related to software reuse;
- effective interpersonal communication is essential to a project's success, hence facilitating communication between team members should be a top priority for the project manager; this may include careful pairing of staff members and assigning a mediating duty to some other members according to their experience and technical background;
- every software project manager should pay adequate attention to a number of ethical issues that always affect the project's success, the development team, and the team's relations to the customers; such issues include acting in the customer's and public interest, maintaining integrity and independence in professional judgement, adhering constantly to the highest professional standards possible, ensuring lifelong learning and supporting colleagues, and so on.

Management Strategies and Techniques

Software development is an extremely dynamic and fluid business, and it is difficult to plan everything at the beginning of a project. Therefore, efficient management of software projects must be based on some explicit, strategic goals and organization's interests. There are a number of useful lines to follow in that sense. Some of them are [3], [23]:

- balancing the need for structure and process control in software development with the need for flexibility, informality, and more effective communication processes;
- establishing software measurement programs and and enfocing accountability for completion of software development milestones;
- *making management objectives and product vision clear to the development team members* (this is very importnt in practice, because far too often developers are in total ignorance of the broader strategy of a company, and the tactical decisions made by management to advance this strategy seem to them arbitrary and even hostile);
- identifying the most critical issues of the project and stressing the need to allocate most development resources, time, and efforts to such issues;
- organizing more visible and formal management processes for reviewing and approving potential product enhancements;
- emphasizing management approaches that facilitate flexibility and creativity within clearly defined boundaries;

- keeping-up with technological developments by enforcing life-long learning, training, courses, and seminars;
- developing more globally focused, culturally sensitive management capabilities;
- involving end-users in the development process in order to constantly provide advice on using the product in the real world, thus eliminating the customer-developer gap;
- promoting orientation towards strategic business partnerships.

There is also a large number of managerial techniques that help monitor software product development on the day-to-day basis and make tactical decisions relevant to the development process [1], [3], [23], [41]:

- maintaining *progress charts* that show the percentage of completion for each module, at any given moment of product development;
- keeping track of all relevant facts about the product (e.g., previous versions, delivery dates, current version), the development process (the problems encountered, resulting delays, and the reasons why they have occurred), and discarded design alternatives in the *external group memory* (it is usually a special-purpose project-management software, or a site on the organization's server or Intranet, and sometimes even a site on the Internet); the external group memory can also serve as a board for discussion on all the relevant ideas that arise in the course of the project;
- estimating time and effort needed for each designer to complete a short-term task, e.g. an iteration in an iterative development process; for that purpose, each designer may be required to initially fill-up and constantly update a *planning sheet* that contains both the designer's original estimates and actual measures (in days) of how long does it take to complete each activity for the task (activities may include analysis, design, coding, testing, and so on);
- emphasizing progress review mechanisms across the development effort;
- applying mechanisms of recognizing, rewarding, and leveraging extraordinary efforts and/or hyperproductivity, as an avenue to promote and retain key technological leaders;
- adapting the software development process to the characteristics of the product being developed;
- increasing parallelism in product development by reducing linear, sequential activities, encouraging relevant communication and social interactions among the team members, and changing the work modes when necessary;
- insisting on creating multiple design views, such as structural, functional, object-oriented, event-based, and data-flow; although sometimes redundant, multiple design views help cover design from multiple perspectives and make it more complete and more efficient;

- enforcing the feedback mechanism in the development process, in order to detect inconsistencies in design as early as possible and reduce the costs of fixing them.

Risk Assessment

In order to prevent project runaways, meet deadlines, stay within the project's budget, and simultaneously maintain the product's high quality standards, it is essential to timely identify and periodically evaluate certain critical factors. Such factors include [1], [10], [29], [30]:

- estimating the project's size in the early phases - the project's size affects how the deadlines will be set up, and is positively correlated with monetary expense and risk;
- setting up the deadlines realistically - as a result, the necessary time to establish the rhythm of the project, prevent delays, and enter a steady state in which the effort is equally distributed from the beginning of the project, without putting an extra workload to the team members at the end of the project phases;
- collecting and studying reports on other similar projects - this provides the possibility of learning from the other projects' and other teams' experiences; in that sense, a process data base is essential for an organization that wants to go higher than Level 2 on the CMM level ladder; engineering management depends on measurements, and their proper use, and this data base is to be regarded as an organizational asset, and it is to be properly managed;
- top management commitment - if top management does not play a strong, active role in the project from initiation through implementation, then all other risks and issues may be impossible to address in a timely manner;
- failure to gain user commitment - when the users are actively involved in the requirements determination process, it creates a sense of ownership, thereby minimizing the risk that the end-user expectations will not be met and that the system will be rejected;
- timeliness of additional user requirements - it is essential to have the users involved in the development process from the beginning to the end; however, it is highly preferable to have the requirements frozen at a certain point in development;
- familiarity with technology - the higher the organization’s experience with application languages, technology databases, hardware, and operating systems, the lower the risk in the project;
- insufficient/inappropriate staffing - the risk of failing to provide adequate staffing throughout the project can be mitigated by using disciplined development processes and methodologies to break the project down into manageable chunks, and developing contingency plans;

- the degree of structure in the project's outputs - it is negatively correlated with the risk in the project;

In the context of the Unified Process of software development, it is adopted that one can never fully eliminate risks; at best, one can manage them [12], [24]. For that reason, the Unified Process stresses the need to drive software development as an architecture-centric activity. Architecture-centric approach forces the risk factors to emerge early in the development process and make the process simultaneously risk-driven - when the risk factors are identified early, managers can take steps to mitigate them. Experienced software project managers recommend to maintain a running list of project's top ten risk factors and use that list to drive each release [12].

Software Metrics

Measurement is a key factor for managing and improving software development. The purpose of the measurement process in software projects is to define and operate a context-specific set of metrics, and to describe the required guidelines and procedures for data collection and analysis [32]. Software measurement generates quantitative descriptions of key processes and products, enabling us to understand behavior and result [37]. Such descriptions can indicate the effort needed to complete the project, the product's quality, estimated schedules and time-to-market, rework effort, estimated project costs, and distribution of resources and costs by project phases. Software measurement makes possible to compare the project a development team is currently working on, to similar projects in terms of budget, costs, productivity, quality, staffing, development processes, and technology used.

In order to operate a metrics program during a software development project, the project manager must enforce continuous measurement of relevant factors. These factors depend on the overall management goals of the measurement process. In that sense, one can differentiate between the following kinds of software metrics [2], [11], [15], [19], [28], [31], [32], [47]:

- *metrics for project size and team productivity* – typical and most widely used representatives of this kind of metrics are *source lines of code (SLOC)* and *function points*; the SLOC metric can be converted relatively accurately and easily into the number of programer-months needed to complete the project; function points are dimensionless numbers that indicate the application's functionality from the user's perspective, and can also be easily converted into the effort needed to complete the project or one of its parts;
- *metrics for schedules* – these include the number of tasks completed on time, the number of tasks not completed on time, the number of tasks with changed schedules, and the number of postponed tasks;

- *metrics for requirements specification* – the number of requests for change (RFC) in specification, the number of new requirements, and the RFC diagram (showing the dynamics of RFC over time);
- *metrics for software testing* – these metrics are used to track the percentage of SLOC covered by the testing process; increasing that percentage reduces the number of errors to be discovered by the users and increases the product's quality;
- *metrics for software quality* – they typically show the fault density (the number of errors per 1 KSLOC) and fault arrival and closing rates; as a rule of thumb, the product's quality is satisfactory if the fault density is lower than 0.25;
- *metrics for project risk* – they measure confidence in the product's ready-to-deployment date (typically an S-shaped curve over time).

The most widely used metrics models include COCOMO [11], which is based on measuring SLOC, function points analysis [2], [19], [28], [47], GQM (Goal-Question-Metrics, based on systematic translation of the company's goals into the measurement process goals, and refinement by defining the concrete measurements to perform in order to support the goals) [32], and Chidamber-Kemerer's metrics suite for object-oriented software projects (specifying metrics for the number of methods per class, depth of inheritance tree, number of children, etc.) [15].

Productivity

Generally, productivity is an output divided by the effort required to produce that output. In software development, the output is a completed software development project. In order to consider a software company's productivity, it is necessary to somehow translate that output into a meaningful measurement. Ideally, software project managers should base output measurement on a combination of a project's size, functionality, and quality [34]. However, such a measurement doesn't yet exist.

On the other hand, various databases of software projects from different business sectors are available nowadays. They make possible to select some projects that closely resemble specific projects in a software development organization, and use the selected projects as a reference for measuring the productivity of the organization's completed projects or estimating the productivity of the organization's new projects.

Alternatively, a company can compare their software development productivity to that of similar projects analytically, i.e. by using some empirical benchmarking equations. Such equations typically take the values of some key productivity factors and use them to calculate productivity in function points per hour. The values are discrete (1 – very low, 2 – low, 3 – average, 4 – high, 5 – very high). The key productivity factors include customer participation, staff

availability, the use of standards in software development, the use of tools, requirements volatility, the application's logical complexity, staff's experience with the tools, and so on. There are also dozens of factors other then the key ones that also influence productivity. Here are two examples of productivity-calculation (benchmarking) equations [34]:

$$\textit{Productivity}\ [\text{fp/hr}] = 0.1072 \times (\textit{number of different languages used})^{-0.4627} \times (\textit{staff availability})^{0.6651}$$

$$\textit{Productivity}\ [\text{fp/hr}] = 0.2127 \times (\textit{sum of inquiries})^{0.1493} \times (\textit{customer participation})^{-0.3950}$$

It is important to stress that the equations like these two depend on the customer's business sector (e.g., banking, insurance, manufacturing, wholesale-and-retail, and public administration). The first equation from above is used to calculate productivity in developing software for wholesale-and-retail business, while the second one corresponds to software development for the public administration sector. The numbers and the kinds of variables (factors) in equations are different for different business sectors.

Statistical analysis shows that software development productivity's variance with respect to some significant variables (considered individually) is different. For example, the productivity variance across companies is rather high - 45%, w.r.t. different operating system it is 19%, and w.r.t. different DBMS tools and hardware platforms it is 13%. Significant variables are different for different business sectors, each individual variable's effect on productivity is either positive or negative, and each variable accounts for a certain percentage in total productivity variance. For example, requirements volatility is a significant variable in most business sectors; its effect on productivity is always negative, and it accounts for 10% to 19% of total productivity variance across different business sectors.

Apart from using equations such as the two above for benchmarking purposes, software project managers can use them to determine the likely impact on productivity of changes in a key factor.

Software Testing

In spite of the fact that in every software development project the product undergoes testing, delivered software always contains residual defects. Software testing is a difficult, time-consuming process. It requires specific skills from software testers, skills that only partially overlap with those of software developers. Apart from mastering coding, testers must also possess a great deal of knowledge of formal languages, graph theory and algorithms [26], [48].

Typically, software testing proceeds in four phases [48]:

- modeling the software's environment

- selecting test scenarios
- running and evaluating test scenarios
- measuring testing progress

In the first phase, the tester's task is to simulate the interaction between the application and its environment, be it the user or the other applications, taking into account all possible inputs and outputs that can cross the application's boundaries. The hardest part here is the fact that in many cases the interactions can go through numerous different file formats, communication protocols, GUIs, and file systems. The other hard part is the unpredictability of the user's actions – the software under test must account for that.

Since the number of possible test scenarios is usually extremely large, testers should select those scenarios that cover all code statements and all significant representatives of external events. Before running the selected scenarios, it is necessary to convert them into executable form (often as code) in order to simulate typical interactions between the system and the external world. Applying test scenarios manually is labor-intensive and error-prone. For that reason, testers try to automate the test scenarios as much as possible. In many environments, automated application of inputs through code that simulates users is possible, and tools are available to help.

Measuring testing progress is difficult, simply because it is not just counting the numbers of bugs found. As stated in the section on software metrics, specific metrics for software testing are used to measure the *coverage* of the tests applied (in terms of running all lines of the source code, forcing all the internal data to be initialized and used, applying all test scenarios, exploring all the inputs, and checking for functional completeness). Note also that software reliability engineering can greatly help - the *Cleanroom methodology* developed at IBM [35] has been particularly useful in improving software quality and providing a quantitative measure for the quality of a software product at its release. The Cleanroom approach provides for the transition of process technology to the project staff and integrates several proven software-engineering practices into one methodology [46]. The testing strategy of the Cleanroom methodology can be best described as random sample based on usage model that predicts field reliability, rather than a futile attempt for coverage and little insight on field reliability.

If the Unified Process is used to manage software development, software testing is performed in every iteration [24]. Test scenarios are defined from use cases, and comprise both functionality and performance testing. The advantage of this incremental and iterative approach to software testing is that in each iteration the testers test just some of the application. Moreover, the tests performed in early phases usually discover such bugs and faults that would cause more severe instability in the project's rhythm if they were discovered in later phases. In every iteration, tests also check whether the current iteration has jeopardized some of the previously built and tested architecture. If the project's size is large, it is impractical to manually run all the test cases, so the use of

automated testing tools is recommended. Project managers should adopt the practice of enforcing thorough testing in every iteration, and not allowing the next iteration to begin before all the tests planned in the current iteration are completed. The entire project is considered completed only when all the UML models *and* all the tests are completed and delivered.

Software Quality Assurance

The goals of software quality assurance (SQA) are monitoring the software and its development process, ensuring compliance with standards and procedures, and ensuring that product, process, and standards defects are visible to management [26].

Quality is the operational behavior of a product required by its users [8]. It comprises a set of product characteristics, both external and internal. External quality characteristics are related to how the product works in its environment (e.g., usability and reliability). Internal quality characteristics reflect how the product is developed (characteristics such as structural complexity, size, test coverage, and fault rates). Important factors affecting product's quality characteristics are process maturity level of the company that has developed the software product, its development environment (such as the design methodology and CASE tools used), and the development team's skill and experience.

It is desirable for a software development organization to plan and control product quality *during* development. Projects managers cannot allow the luxury of going back and adding quality - by the time a quality problem is detected, it is probably too late to fix it [39]. For that reason, it is necessary to establish procedures and expectations for high levels of quality before any other development begins. Also, hiring developers proven to develop high-quality code, staffing the project accordingly, and enforcing peer-level code reviews and external reviews must be top priority of every software project management.

Planning and controlling software product quality during development requires [8]:

- establishing targets for the external quality characteristics;
- pursuing those targets during development by defining and monitoring targets for internal quality characteristics - this can be done using conventional software measures of size, fault rates, change rates, structure, test coverage, and so on, taken early in product development;
- establishing relationships between internal and external quality characteristics, using experience from similar past software development projects;
- identifying and setting targets for internal quality characteristics.

In practice, all this can be done by first defining a *quality model* (in terms of measurable quality characteristics; it can be an international standard like ISO 9126, or a company-specific model), and then applying a *quality process*.

Quality process includes quality specification (establishing the software product's quality requirements), planning (deciding on a suitable development process and setting target values for measurable internal quality characteristics), control (monitoring progress throughout development using internal software measures associated with deliverables and activities related to each major review point in development), and evaluation (measuring the actual values of the external quality characteristics and comparing each actual value with its target value). Maintaining and using a database of past projects helps perform each step in the process more successfully.

Software Configuration Management

The configuration of a software system is the function and/or physical characteristics of hardware, firmware, software or a combination thereof as set forth in technical documentation and achieved in a product [14]. It can also be thought of as a collection of specific versions of hardware, firmware, or software items combined according to specific build procedures to accomplish a particular purpose [26].

Software configuration management (SCM) comprises a set of technical, managerial, and administrative activities related to identifying the configuration of a software system at distinct points in time for the purpose of systematically controlling changes to the configuration, recording and reporting change processing and implementation status, verifying compliance with specified requirements, and maintaining the integrity and traceability of the configuration throughout the system life cycle [6]. Responsibilities of each software project manager related to SCM include enforcing the practice of SCM activities for the project, distributing the activities to the relevant individuals, and managing and administrating the results of these activities.

Figure 6 illusrtates the activities encompassed by SCM. Since SCM is a supporting lifecycle process to software product development and maintenance, a successful SCM implementation requires careful management and planning. These are typically performed by the project manager or another designated individual, who does it in close relation with SQA activities. Management and planning activities cover all the other sets of activities shown in Figure 6, establish all the relevant SCM policies, and result in recording/updating the Software Configuration Management Plan (SCMP) for the project. The SCMP is typically subject to SQA review and audit.

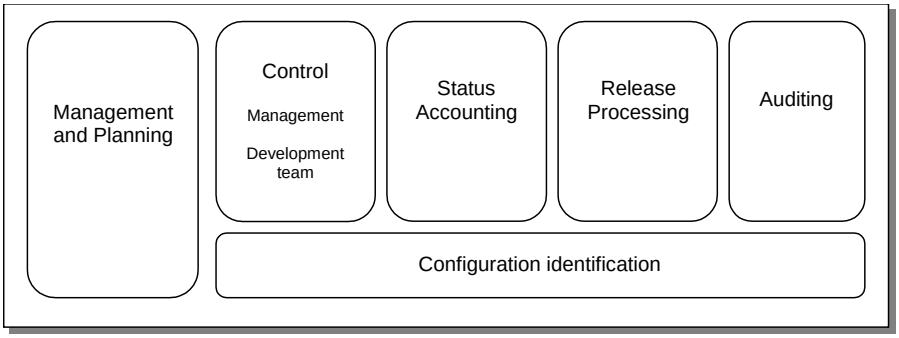

Figure 6 – Software configuration management activities (after [26])

Configuration identification activities provide the basis for other SCM activities. These activities enumerate the configuration items to be controlled (such as plans, specifications, source and executable code, code libraries, data and data dictionaries, testing materials, software tools, and documentation for installation, maintenance, operations and software use), establish identification schemes for the items and their versions, and establish the tools and techniques to be used in acquiring and managing controlled items.

SCM control activities involve both managers and developers. Managers make decisions on whether some changes in configuration should be made or not, and authorize the changes. Developers perform change activities (code management) in a coordinated manner. Status reports are generated that account for each change in the configuration, and can be of use to various parties in the project, including managers, developers, testers, SQA team members, and maintenance engineers. The information obtained by status accounting can also serve as a basis for various measurements, such as the number of change requests per software configuration item and the average time needed to implement a change request. Release processing activities support customers and the maintenance team. They are related to identification, packaging and delivery of the elements of a product (such as the software, its documentation, release notes, and configuration data), as well as product version management (versions for different platforms or versions with varying capabilities). The software configuration auditing activity determines the extent to which an item satisfies the required functional and physical characteristics. Its ultimate goal is to evaluate the conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures [26].

Standards

There are two major aspects of the term ``standards" in software development. One of them is that of using widely accepted standards under the assumption that they embody ``the common body of knowledge and accepted state of industry

best practice" [43]. Such standards include universally recognized control frameworks for software process control and improvement. Some examples include ISO 9000, ISO 12207, TickIt, and CMM. They provide a basis for defining systematic activities, roles, and tasks that can be carried out in software development, independent of individual projects, companies, or designers. Furthermore, they make possible to understand and manage all of the diverse forms of software activity from the standpoint of a single framework. Software project managers should understand and apply these standards and frameworks as points of reference in software development, in order to ensure that quality is being designed and built into the products.

The ISO 9000 series of standards provide a generic model for the Quality Management System (QMS) of a supplier organization that is involved in design and development activities. It specifies the requirements against which the organization's QMS can be formally assessed. ISO 9000's issue of particular interest for software industry is ISO 9000-3 Guideline for the Application of ISO 9001 to the Development, Supply and Maintenance of Software.

The ISO 12207 standard covers the entire lifecycle of software, from inception through extinction. It details processes for acquiring and supplying software products and services.

TickIt is the standard related to ISO 9000-3. Its purpose is to fill in the gaps and clarify the relationship between ISO 9000-3 and ordinary software development operations. It does this by adding additional documentation and audit requirements to the ISO 9000-3 Guidelines, and by providing direction needed to implement ISO 9000-3 compliant quality system.

Software Engineering Institute's Capability Maturity Model (CMM), described in the *Organizational aspects* section does not have a formal status; hence it is called a model, or a framework, rather than a standard. It is used as a reference to establish the maturity level of an organization's software processes.

The other aspect of the term ``standards'' in software development is that of deploying organizational standards across the life of a project [38]. In large organizations, having many software development teams, some standardization of methods across teams is important. For example, an organization may prescribe standard (consistent) processes, roles, schedules, and reusability policies across teams and projects. It can lead to many benefits, including better planning, more predictable outcomes, increased staffing flexibility (decreased sensitivity to employee turnover), and reuse of experience.

In that sense, the most important standards are those used for the roles and processes in a development team. There are many such informal standards. Some of them are developed internally by a company for its sole use; other standards are developed and marketed by for-profit companies as products. A good example is the Microsoft Solutions Framework (MSF), which is used in a number of software development organizations. It revolves around a team with a process. The roles it prescribes are product management (focusing on customer satisfaction), program management (on-time delivery), development, testing,

user education, and logistics planning (smooth rollout and migration). Each role may be played by a number of different people. The process model describes the project in terms of phases, milestones, activities, and deliverables. It also shows how they are related to the team model, and what practices and principles drive the product development (versioned releases, scheduling for an uncertain future, managing trade-offs, managing risk, maintaining a fixed ship-date mindset, breaking large projects into manageable parts, performing daily builds, and using bottom-up estimating). The process model also defines what documents each role is expected to deliver at precise points in the iterative process. MSF provides detailed document templates.

Best Practices

Not every practice of software development can be standardized, yet many of them have proved to be useful in a number of projects and organizations. Most of such practices come from experience, and it is extremely beneficial for every project that the project manager and the members of the development team are knowledgeable of as many such practices as possible. The following examples illustrate the idea of using such practices in software project management:

- Communication problems arising when a distributed team is developing software must be handled with special care. A kick-off meeting must be held face-to-face, and all the developers, partners, and contractors must attend. All product deliverables must be clearly defined in the very beginning. After this the communication between team members is mostly electronic. Time zones differences are not necessarily a problem – on the contrary, they may turn out to be an advantage, since by the time one group of developers comes to work in the morning, the testers may have already sent them the test results from the other side of the Globe! Some overlap in working hours is, however, desirable. Establishing *availability standards* (when and how the team members will be available for communication, how quickly they respond to emails, and so on) facilitates electronic communication in software development done by distributed teams [22].
- Top-down approach to software analysis and design should be enforced in software development projects when the application domain is numerically intensive, such as signal processing, pattern recognition, and real-time control. The reason is that in such applications the data are not predictable enough, and developers are usually not familiar enough with the data. For example, if object-oriented design is used in developing such an application, then more general classes (located close to the top of the inheritance tree) should be designed first, and design of their children classes should come next. Contrary to that practice, developing a business-oriented application (an information system) suggests bottom-up

approach. Developers of such applications usually know their data well, and it is quite natural to start the projects from modeling data to be stored in databases [20]. As for the top-down approach to analysis and design, note also that it may inhibit reuse. It may be more economical to look at what reuse and COTS (components-of-the-shelf) software components are available, and select a design based on them. Such a system may not fully meet a client's needs, but the client may be happy to accept it on the basis of its greatly lower cost.

- Productivity of a newly assembled development team should be calibrated in a pilot project. A small pilot project gives project manager the possibility of gaining a rough model of performance for every team member and for the team as a whole before the real work on a large application begins. Put simply, the work on a pilot project can give a multiplier factor *m*, so that if a certain team member estimates to take *n* weeks to do something, in reality he/she will need *m* times *n* weeks. This is important for schedule planning for the larger project. How trustworthy and realistic is the multiplier factor acquired that way depends to a large extent on whether the application domains of the pilot project and the larger application are the same or not and whether the team member(s) is (are) familiar with the domain(s) or not [12].

Discussion and Conclusions

Software project management issues described in this chapter represent the core of project manager's toolbox for leading the project to its successful completion. It is important to note that there are many more interesting topics and practices of software project management, other than those covered here. Due to the enormous expansion of IT, such practices continue to grow.

Again, if a certain issue specific to software development is not addressed in the chapter it doesn't mean that it is not important - it is just because it didn't satisfy some or all of the selection criteria stated in the end of the Introduction. For example, an important issue for every software project manager is contingency planning. It relates to the effect of strikes by personnel, fire, flooding, earthquakes, each of which can have a rather specific effect on the software process - copies of work products are to be kept off site, and be easily accessible. However, contingency planning doesn't get that much space in the relevant software engineering journals and magazines, or in the publications of the relevant committees associations.

Another such important issue is that of how the business reengineering efforts (reorganization of an entire enterprise) have, or should have, affected the management of software projects [17]. Recent results in that sense include reengineering of software enterprises in order to introduce CMM [7]. These efforts aim at reengineering of a software enterprise based on introducing some

important principles into the organization, in terms of very general statements of what engineers do (e.g., software development follows a plan in accordance with requirements, requirements are ranked according to cost-effectiveness and are implemented incrementally, standards are used in development, design includes fault- and failure-tolerance, and so on). The principles are consistent with the capabilities that CMM recognizes as supportive of and important in the development of high quality software.

However, it is important to stress that even if all of the important practices and issues could be briefly covered here, software project management in reality requires a more detailed insight into the practices themselves and many case studies, as well as a lot of experience, judgment, and intuition. Experienced managers always take the issues and practices reported in the open literature just as rough guidelines and adapt them to the context of their current project. This suggests an important general rule of thumb: best practices of software project management are always those that can be applied to the system being built, the technology the developers use, and the organization that develops the system.

Software Project Management Topics on the Web

There are many software project management resources on the Web. The short list of URLs shown below has been composed according to the following criteria:

- the number of useful links following from that URL
- how comprehensive the site is
- how interesting the URL is for software project managers
- how interesting the URL is for practitioners and researchers

- Software Engineering Institute (SEI):
  http://www.sei.cmu.edu
  This is one of the best starting points for all software engineering topics, including software project management. It includes a number of links to relevant on-line literature, conferences, and other resources.
- The International Conference on Software Engineering (ICSE):
  http://www.ul.ie/~icse2000
  ICSE is the largest software engineering conference. The topics cover all aspects of software engineering, including software project management. The program attracts both practitioners and researchers.
- The Unified Process
  http://www.rational.com/products/rup/index.jtmpl
  The Rational Software Corp. site contains various resources related to UML and Unified Process.
- Extreme Programming
  http://www.extremeprogramming.org/

eXtreme programming is a methodology designed to address the specific needs of small-team software development, facing vague and changing requirements.

- Requirements Engineering:
  http://www.cs.ucl.ac.uk/research/renoir/
  This is the site of *Renoir*, the European Union Requirements Engineering Network of Excellence.
- Software Metrics:
  http://www.udmercy.edu/academic/business/metrics.htm
  A number of links and resources related to software metrics.
- Productivity:
  http://www.sttf.fi/html/exppro.html
  Experience Pro, a tool incorporating the Experience database of software projects comprising data collected from companies operating in different business sectors. The data can be used for preliminary project planning, including cost estimation, reuse analysis, software-process-capability analysis, risk analysis, and productivity benchmarking.
- Software Testing:
  http://www.io.com/~wazmo/qa
  An annotated list of links to some of the Web's best testing information.
- Organization's internal standards:
  http://www.microsoft.com/msf/
  Microsoft Solutions Framework (MSF) that prescribes software development process model, principles, practices and roles to deploy consistently across development teams in a company.

References

1. Ahituv, N., Zviran, M., Glezer, C., Top Management Toolbox for Managing Corporate IT. *Communications of The ACM* 42, April 1999, pp. 93-99.
2. Albrecht, A.J., Gaffney, J.E., Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering* 9, 1983, pp. 639–648.
3. Athey, T., Leadership Challenges For the Future. *IEEE Software* 15, May/June 1998, pp. 72-77.
4. Beck, K., *eXtreme Programming Explained: Embrace Change*. Reading: Addison-Wesley, 2000.
5. De Bellis, M., Haapala, C., User-Centric Software Engineering. *IEEE Expert* 10, February 1995, pp. 34-41.
6. Bersoff, E.H., Elements of Software Configuration Management. In: Dorfman, M., Thayer, R.H. (eds), *Software Engineering*. Los Alamitos: IEEE Computer Society Press, 1997, pp. 345-356.
7. Berztiss, A., *Software Methods for Business Reengineering*. New York: Springer, 1996.

8. Boegh, J., Depanfilis, S., Kitchenham, B., Pasquini, A., A Method for Software Quality Planning, Control, and Evaluation. *IEEE Software* 16, March/April 1999, pp. 69-77.
9. Boehm, B., A Spiral Model of Software Development and Enhencement. *IEEE Computer* 21, May 1988, pp. 61-72.
10. Boehm, B.W. Software Risk Management: Principles and Practices. *IEEE Software* 8, January 1991, pp. 32–41.
11. Boehm, B.W., et al., Cost Models for Future Life Cycle Processes: COCOMO 2.0. *Annals of Software Engineering* 1, November 1995, pp. 1-24.
12. Booch, G., *Object Solutions - Managing the Object-Oriented Software Project*. Reading: Addison-Wesley, 1996.
13. Booch, G., Rumbaugh, J., Jacobson, I., *Unified Modelling Language User's Guide*. Reading: Addison-Wesley, 1999.
14. Buckley, F.J., *Implementing Configuration Management: Hardware, Software, and Firmware* (Second Edition). Los Alamitos: IEEE Computer Society Press, 1996.
15. Chidamber, S.R., Kemerer, C.F., A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering* 20, 1994, pp. 476-493.
16. Constantine, L.L., Work Organization: Paradigms for Project Management and Organization. *Communications of The ACM* 36, October 1993, pp. 35-43.
17. Davenport, T.H., *Process Innovation: Reengineering Work through Information Technology*. Boston: Harvard Business School Press, 1993.
18. Davis, A.M., Hsia, P., Giving Voice to Requirements Engineering. *IEEE Software* 11, March 1994, pp. 12-17.
19. Deveaux, P., Counting function points. In: Keyes, J. (ed.), *Software Engineering Productivity Handbook*. New York: McGraw-Hill, 1993, pp. 191-227.
20. Fayad, M.E., Cline, M., Managing Object-Oriented Software Development. *IEEE Computer* 29, September 1996, pp. 26-32.
21. Ferdinandi, P.L., Facilitating Communication. *IEEE Software* 15, September/October 1998, pp. 92-96.
22. Haywood, M., Working in Virtual Teams: A Tale of Two Projects and Many Cities. *IEEE IT Professional* 2, March/April 2000, pp. 58-60.
23. Holtzblatt, K., Beyer, H., Making Customer-Centered Design Work for Teams. *Communications of The ACM* 36, October 1993, pp. 92-104.
24. Jacobson, I., Booch, G., Rumbaugh, J., *The Unified Software Development Process*. Reading: Addison-Wesley, 1999.
25. Jarke, M., Requirements tracing. *Communications of The ACM* 41, December 1998, pp. 32-35.
26. Joint IEEE/CS-ACM Committee, *Guide to the Software Engineering Body of Knowledge - SWEBOK*. http://www.swebok.org/, December 2000.
27. Joint IEEE/CS-ACM Committee, *Software Engineering Code of Ethics*. www.acm.org/serving/se/code.htm, December 2000.
28. Jones, C., *Applied Software Measurement*. New York: McGraw-Hill, 1991.
29. Jones, C., Our Worst Current Development Practices. *IEEE Software* 12, March 1996, pp. 102-104.
30. Keil, M., Cule, P.E., Lyytinen, K., Schmidt, R.C., A Framework for Identifying Software Project Risks. *Communications of The ACM* 41, November 1998, pp. 76-83.
31. Kulik, P., A Practical Approach to Software Metrics. *IEEE IT Professional* 2, January/February 2000, pp. 38-42.
32. Lavazza, L., Providing Automated Support for the GQM Measurement Process". *IEEE Software* 17, May/June 2000, pp. 56-62.

33. Lindvall, M., Rus, I., Process Diversity in Software Development. *IEEE Software* 17, July/August 2000, pp. 14-18.
34. Maxwell, K.D., Forselius, P., Benchmarking Software Development Productivity. *IEEE Software* 17, January/February 2000, pp. 80-88.
35. Mills, H.D., Dyer, M., Linger, R.C., Cleanroom Software Engineering. *IEEE Software* 4, September 1987, pp. 19-24.
36. Paulk, M.C., et al., *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading: Addison-Wesley, 1995.
37. Pfleeger, S.L., Jefferey, R., Curtis, B., Kitchenham, B., Status Report on Software Measurement. *IEEE Software* 14, March/April 1997, pp. 33-43.
38. Rada, R., Craparo, J., Standardizing Software Projects. *Communications of The ACM* 43, December 2000, pp. 21-25.
39. Reel, J.S., Critical Success Factors In Software Projects. *IEEE Software* 16, May/June 1999, pp. 18-23.
40. Reifer, D.J., Requirements Management: The Search for Nirvana. *IEEE Software* 17, May/June 2000, pp. 45-47.
41. Rettig, M., Simons, G., A Project Planning and Development Process for Small Teams. *Communications of The ACM* 36, October 1993, pp. 45-55.
42. Shaw, M., Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs: Prentice Hall, 1996.
43. Shoemaker, D., Jovanovic, V., *Engineering a Better Software Organization*. Ann Arbor: Quest Publishing House, 1999.
44. Siddiqi, J., Challenging Universal Truths of Requirements Engineering. *IEEE Software* 11, March 1994, pp. 18-20.
45. Software Engineering Institute's CMMI Product Development Team, *CMMI-SM for Systems Engineering/Software Engineering, Version 1.02, Continuous Representation (CMU/SEI-2000-TR-019)*. http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr019.pdf, 2000.
46. Wayne Sherer, S., Kouchakdjian, A., Arnold, P.G., Experience Using Cleanroom Software Engineering. *IEEE Software* 13, May 1996, pp. 69-76.
47. Whitmire, S.A., Applying function points to object-oriented software models. In: Keyes, J. (ed.), *Software Engineering Productivity Handbook*. New York: McGraw-Hill, 1993, pp. 229-244.
48. Whittaker, J.A., What Is Software Testing? And Why Is It So Hard? *IEEE Software* 17, January/February 2000, pp. 70-79.
49. Woodward, S., Evolutionary Project Management. *IEEE Computer* 32, October 1999, pp. 49-57.
50. Zahniser, R.A., Design By Walking Around. *Communications of The ACM* 36, October 1993, pp. 115-123.