Ethan Crawford

April 1, 2007

CSS 360

What is software engineering?

Software engineering is the discipline concerned with turning software into dollars. This has surprisingly little to do with writing code, a fact which may come as a disappointment to a majority of computer science students. The relatively small amount of computer science in software engineering, however, is more a function of the immense scope of the discipline, rather than any lack of technical opportunity within it. Software engineering encompasses not only design and coding, but multiple management disciplines, hardware, mathematics, systems engineering, law, social sciences, and arts and humanities. If software development is the star actor on a stage, software engineering is the agents, the catering crew, the lawyers, the janitors, the accountants, the producer and the director.

More formally, software engineering is the set of practices and processes that enable development teams to build useful software products in a timely fashion at a predictable cost. In this definition, “useful” is either how well the software product meets the requirements of the customer, or how well it does in the marketplace against competing products. A “timely fashion” is soon enough to make money on the product; preferably, soon enough to maximize the product’s revenue potential. “Predictable cost” is self-defined, as no project can outspend its budget without significant consequences.

Delivering a complicated software product with limited time and constrained resources is a complex task, and pulling it off successfully is a team task that requires different players to perform specific roles at different points of the software development lifecycle. It is very common for a single individual to devote his or her career to perfecting only one or two of these roles. Most large product teams will have at least one of the following members: systems analyst, program (or project) manager, architect, developer, tester, marketer, salesperson, usability engineer, designer, support engineer, maintenance programmer, and one or more manager.

To understand why the discipline of software engineering is required, it can be helpful to put oneself into the shoes of a member of a product team. Most team members are responsible for only a small aspect or component of the entire project, and their regular work is relatively short-term (measured in weeks or months). This organizational structure is effective at creating experts in specific areas, but it tends to cause individuals to lose sight of larger product-related issues. This is an issue because each of the pieces must work with the other pieces, and communication between both software modules and individuals is key. This becomes a larger issue as the size of the team increases.

To address this need for greater communication and coordination, some members of an effective product team must specialize in those subjects. Some members deal with the overall project progress, focusing on clarifying requirements, developing schedules, tracking progress, and addressing performance problems and schedule slips. Some deal with the larger health of the organization as a whole: while it is good if the next product produced by the team is a best-seller, it is bad if the effort expended in producing the best-seller leads the company to bankruptcy, or if the time consumed in producing the best-seller causes multiple other business opportunities to be missed. In mature organizations, an even smaller set of people focus on strategic business requirements that might be years in the future. Between these three general groups, short-term, medium-term, and long-term business goals are addressed, hopefully to the benefit of the organization.

Perhaps more than anything, software engineering facilitates long-term business goals. It may very well be possible to deliver a single product without following many of the software engineering processes. However, it is nearly impossible to ship multiple versions without them. Repeatedly producing quality products implies the presence of advanced and involved processes for defect prevention, defect repair, and incorporating customer feedback. Customers interested in purchasing the most robust software product may require a set of metrics that reflect the level of process at use within the organization, and may also require proof that testers have checked for certain classes of code defects. An organization interested in attracting these customers must develop additional process to report on internal processes. Aligning this “meta-process” with an established industry standard is a useful way to allow customers to compare one organization to another with some semblance of objectivity.

In order to comprehend the arc of the software development lifecycle, it is useful to examine a set of general questions likely to be faced by most teams attempting to produce a software product:

* What does our customer need?
* What processes will guide our development effort?
* How will we keep the project on track, and which portions are risky?
* How will we build our product?
* How will we test our product?
* How do we know when the product is finished, and how do we determine how “good” it is?

These questions describe all the basic sub-divisions of software engineering, and each one covers several topics. The remainder of the paper provides a quick summary of the main topics within each subdivision.

“What does our customer need?” Assessing Requirements.

Just as every play requires a conflict and a resolution, every software product requires at least one problem and a requirement that addresses the problem. Turning a problem into a requirement involves many considerations. One of the first is identifying the parties who are interested in the outcome of the project, otherwise known as “stakeholders”. Typical stakeholders are the end-user, the customer, and the development team. Next, feature requirements (“the customer needs a website…”) must be separated from environmental, process, and system requirements (“…that runs on Linux and is developed with Java”). Supplementary quality requirements (“the site should respond within 10 ms to client requests and should be available 99.99% of the time”) must be also identified.

Then, the hard work begins. Requirements must be documented in language precise enough for the development team, but clear enough for the customer. Each requirement receives a priority and a completion state, to help with allocating resources and tracking project status. Product planners, legal professionals, usability experts, programmers, and testers all need to review the requirements and provide feedback. Multiple levels of requirement specifications may need to be created to satisfy all the stakeholders. Specifications must then be reviewed, updated with feedback, iterated, and kept up-to-date as the project progresses. From the specs, object models and prototypes can be built to resolve design ambiguities and prove key concepts.

The requirements process is the backbone of any development effort. Project management relies on requirement specifications (“specs”) to create schedules, track progress, and communicate with customers. Development relies on specs to direct the design effort. Test relies on specs to verify product conformance to requirements. Quality Assurance teams rely on specs as a datapoint used to calibrate future development efforts. When fixing bugs and adding features, maintenance programmers rely on specs for hints as to the original intentions (and future goals) of the dev team. Without a strong requirements process, chances are good that the rest of the project will fall apart.

“How will we keep the project on track, and which portions are risky?” Process and Project Management.

Project management is the discipline most involved with keeping projects on track. Unlike many of the other disciplines, it is required at all phases of the development lifecycle. Effective project management requires involvement with a large set of different processes, ranging from technical (“what bug-tracking system will we use?”) to process-related (“How will we ensure that everyone knows how to use the new bug-tracking system?”) to people-related (“We do we need to hire to teach our bug-tracking system training course?”)

Project managers must be effective communicators. They are responsible for explaining project progress to managers higher in the organization, and facilitating communication between engineering teams and outside partners and customers. They also create schedules, vision documents, human resource job postings, and almost any other form of written communication required by a project.

Project managers are also the keepers of the various processes used by the product team to do their job. They adopt, drop, and customize processes to facilitate the workflow of the entire team. Additionally, they identify risky features, help determine areas of emphasis for evaluating software quality, hold meetings, and do any number of other required tasks. Perhaps of all the disciplines, this one is the most difficult to summarize.

“How will we build our product?” Development (design and coding).

Programming is the star role in the development process. Individual developers can become well-known and well-respected for their technical prowess, and many young computer science graduates aspire to this position. For some, it is the only “real” position in the entire process. Taken as a percentage of the software engineering discipline, however, coding occupies only a tiny fraction over the overall project time. Most of the energy expended in developing the code comes in the form of design. Design falls into two main areas: high level and low level.

Experienced developers, often called “architects”, will focus on the design of several key high-level areas. First, the enclosing system is identified. After this, an overall architecture (3-tiered, client/server, etc) is selected. On top of this basic premise, the public interfaces and functional modules (sets of inter-related classes) are roughly defined. During this process, architects will check for the “coupling level”, the number of module interdependencies. “Loose coupling”, a low rate of inter-module dependence, is a high-priority goal of any new development effort, as it facilitates maintenance. Architects will also inspect existing modules and determine which are reusable with little or no modification in the new product.

As portions of the high-level design are completed, low-level design in the area can begin. This involves selecting data structures, design patterns (common idioms used by developers to solve similar problems), calling conventions, error-handling mechanisms, programming conventions (object-oriented, structural, etc) and other common computer science tools. Subroutine prototypes can be created and tested for flow and readability before the bodies of the subroutines are implemented. As part of the process, flow charts, modeling diagrams, and developer design specifications may be created to help document the ongoing work.

Finally, it is time to write code. The primary focus of the coding phase is on the requirements established at the beginning of the process. Conformance to requirements is the single largest measure of product quality, and the coding phase is when this either does or does not happen. If it does not, fixing the code after the fact becomes significantly more expensive than fixing it now, due to the cost of training a maintenance programmer and of deploying the updated software to what could potentially be millions of customers.

Writing solid code is a complete discipline, and involves many elements. Among them are consistency in error-handling, reducing code duplication, creating modular, reusable designs, and following established project coding conventions. Each of these contributes to making possible other goals of software engineering, namely quality, reliability and ease of maintenance.

“How do we test the product, how do we know when it is finished, and how do we determine how good it is?” Testing and Quality Assurance.

Along with code, developers create bugs. Testing and quality assurance are the processes of finding bugs today and preventing bugs in the future. While frequently a separate discipline from development, testing is practiced by both programmers and non-programmers. In fact, in some development methodologies, the role of tester has been merged completely into the role of developer. Traditionally, however, they are separate disciplines with unique areas of emphasis.

Testers are the first line of defense between the product team and customer support, and it is their job to reduce support costs by finding as many code defects as possible prior to releasing the software to customers. The most basic form of testing is exercising daily builds of a product, using it the way a user would. More complex methodologies are typically employed, using tools and various automation techniques to both expand the scope of the test coverage and to check for code defects in areas not visible to a user.

Traditionally, quality assurance team members focus more on failure analysis (determining the introduction vector of bugs in the previous product) and defect prevention (proactively modifying test and development process to prevent introducing a similar class of bugs in future products). Increasingly, it is more common to see some failure analysis techniques practiced against the current product while it is still in development, partially as a bug-finding mechanism.

Quality assurance teams are also responsible for providing metrics on quality, which can be used as part of an organization’s overall rating in a standardized metrics reporting scheme for vendor comparison purposes.

Summary

Building successful, large-scale software products is an enormously complicated undertaking, and its sheer complexity lends weight to the argument that “Software Engineering”, while perhaps not as mature an engineering discipline as civil or electrical, is at least not misnamed. Given that most Computer Science graduates are likely to work somewhere in the software industry, it is well worth their time to make a brief study of the overall discipline. Coding, while challenging and enjoyable, is but a tiny piece in the larger puzzle, and enormous challenges and great benefits await anyone willing to step away from their compiler and into the real world.

**References:**

Roger S. Pressman, *Software Engineering: A Practitioner’s Approach, 6th Ed.* (New York: McGraw-Hill, 2005)

Wikipedia, “Software Engineering”, <http://en.wikipedia.org/wiki/Software_engineering>

Software Engineering Body of Knowledge, <http://www.swebok.org>

IEEE Software Engineering online, <http://www.computer.org/>