Module 2

Software Processes, Project Planning

Objectives:

1. Understand what a software process is.
2. Be familiar with several types of software process models, and when they would be useful.
3. Understand the importance of project planning and scheduling.
4. Understand the following terms: deliverable, milestone, critical path.
5. Know how a Gantt chart is used.
6. Understand the terms, measure and metric, and what they are used for.
7. Be able to give examples of software measures and metrics.
8. Be familiar with 3 ways of measuring software size.
9. Understand some of the limitations of software estimation models.

Reading:

1. Chapters 2 & 23.
2. Web sections accompanying chapters 2 & 23 from the textbook's companion website (http://www.cs.st-andrews.ac.uk/~ifs/Books/SE9/).
3. Also, read the pages I've posted regarding the semester project (team guidelines deliverables, and the rubric). They are located in the "Semester Project" section of the Blackboard site.

Assignment:

Answer one question in the discussion forum for Module 2, and respond to the posts of at least two other students.

When I have posted the teams for the group project, get acquainted with your teammates, and complete the Scope Statement assignment that will be posted later this week.

Quiz:

Take the quiz for Module 2.

Software Processes and Process Models

Most products that are marketed to the general public are developed and manufactured according to some kind of plan, in order to ensure quality, consistency, and that the product is completed on schedule. The plan that is used to guide the manufacturing is called a **process**. The process consists not only of the actual manufacturing steps that must be followed, but also management activities, feasibility analyses, quality assurance, etc.; that is, any activity required for the manufacture of a product. Different products will have differences in their processes, but they will also share some common features. An abstract form describing the common activities of a process, that can be re-used for multiple projects, is called a **process model**.

There are processes and process models in software engineering as well, and as with other types of products, there is no single process model that is optimal for all software projects. However, there are four activities that are fundamental to the success of all software projects:

1. **Software specification**

   The purpose of software specification is to determine what it is that needs to be built. This activity consists of eliciting the software requirements from the customer, and developing a written specification detailing what the software should do, what it should not do (where applicable), and any constraints that affect the functionality that should be present.

2. **Software design & implementation**

   This activity is actually two separate activities. Software design is concerned with translating the software requirements into an architectural blueprint that states how the software should be built in order to meet its requirements. Software implementation is the actual manufacturing process where the actual product is built according to the design. In the context of software engineering, implementation is usually spoken as though it is synonymous with coding (and you may catch me treating the two that way during the course), but as the field has advanced it has now become possible to develop some software with a minimal of code writing.

3. **Software validation**

   Software validation is concerned with determining whether or not the implemented software does what the customer wants it to do. This activity involves testing the software to make sure each of the requirements in the specification has been met.

4. **Software evolution**

   Over time, as the needs of the customer change, business rules change, and technology changes, software must accommodate these changes in orer to remain useful. The software must, therefore, evolve with its environment. Evolution can occur by modifying or extending the existing software, or by developing a new version of the software.

Many software process models have been created over the years, all of which incorporate the above activities. Some models are more suitable to certain types of projects than other models, and some models have more or less fallen into disuse. Next, we will look at some of the most commonly used software process models.

The Waterfall Model

The waterfall model has been around for ages. It’s a strictly linear approach to building software, where the four common process activities are completed in order. A given phase of development cannot proceed until the preceding phase has been completed. No coding can be started until the design has been completely finished; testing cannot begin until all the code has been written. Although the waterfall model seems very straightforward and easy to follow, there are two fundamental problems with using it:

1. Most people don't tend to solve complex problems linearly. Complexity is usually associated with incomplete understanding of the problem being solved. Progression through a particular phase of development brings about greater understanding of the problem, which in turn can shed light on flaws or errors that were made in earlier phases. Many people do not tend to truly think linearly anyway, and so tend to skip around from one part of the problem to another.

2. Mistakes can be found anywhere in the process life cycle, but many mistakes are not detected until the testing phase, which doesn't occur until near the end of the process life cycle. Correcting a mistake always involves going backward through the process to the stage where the mistake originated. This can be very costly, depending where in the process mistakes are discovered. Figure 2.1 shows how the cost of correcting a mistake increases with time. I've left off the units since the cost will vary depending on the project, the team, and other factors. The important point is that the cost always increases the later one gets into development. The broken bar for the cost after the software has been released indicates an enormous jump in cost. This is due to the fact that once the software is released, correcting a mistake also involves providing patches or upgrades to customers already using the software. Although I am focusing on errors here, you should realize that the same applies to any kind of change that must be made in the software. Correcting an error or fault is simply one type of change.

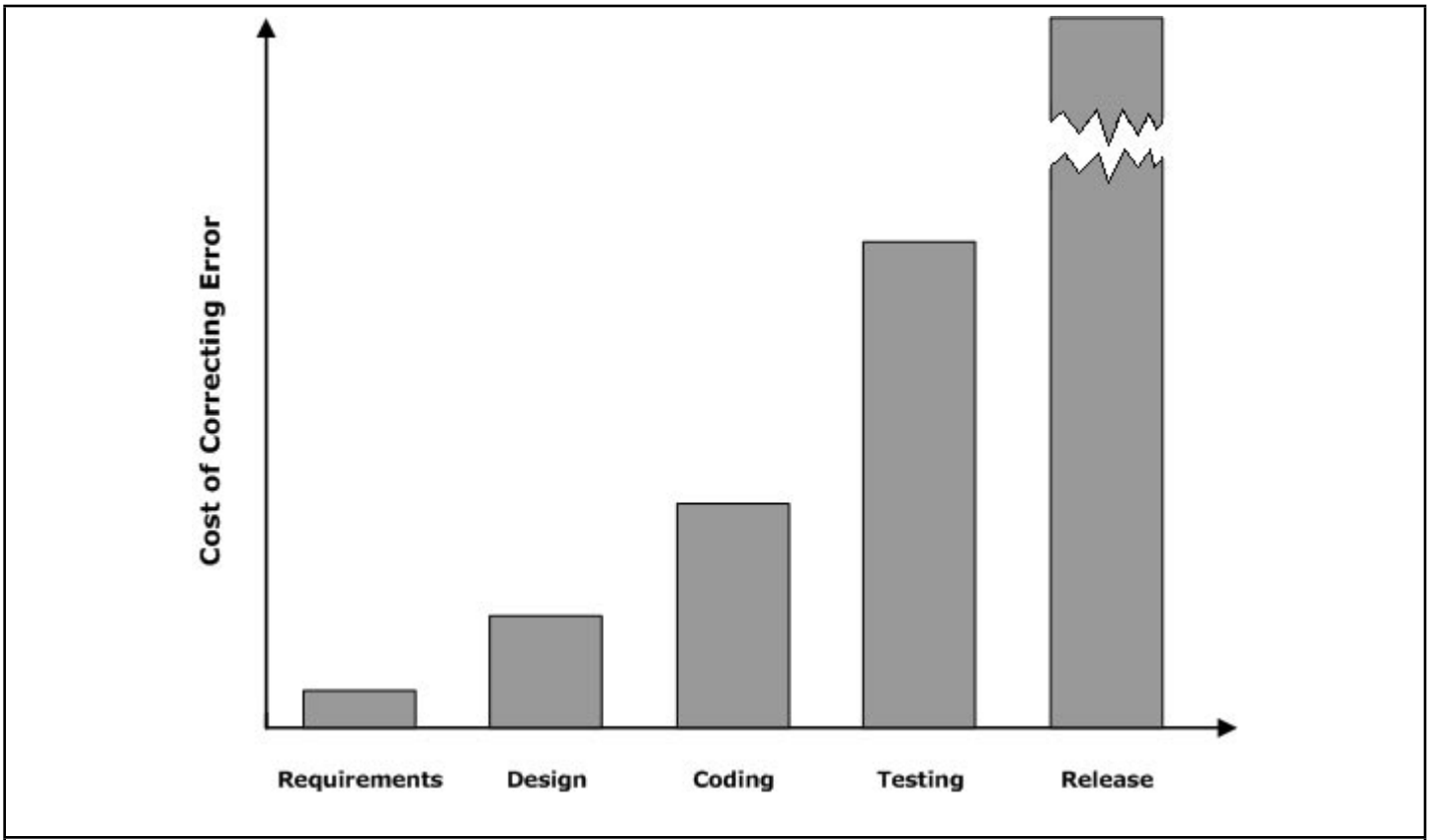

Figure 2.1. Cost of correcting an error using the waterfall model.

The V Model

The V Model, shown in figure 2.2, was proposed to illustrate the importance of testing throughout the phases of development. Each phase of development has a corresponding testing phase. The coding phase is at the bottom of the 'V'. Unit and integration testing are used to verify individual components. System testing verifies the entire software system. Requirements are validated by acceptance testing. Finally, the released software represents the solution to the original problem.

The terms **verify** and **validate** have distinct meanings in software engineering. Both ensure the software conforms to its specifications, but validation is more customer-oriented, while verification is more developer-oriented. Their definitions are probably most succinctly described by Barry Boehm [1]:

- "Are we building the right product?" (validation)
- "Are we building the product right?" (verification)

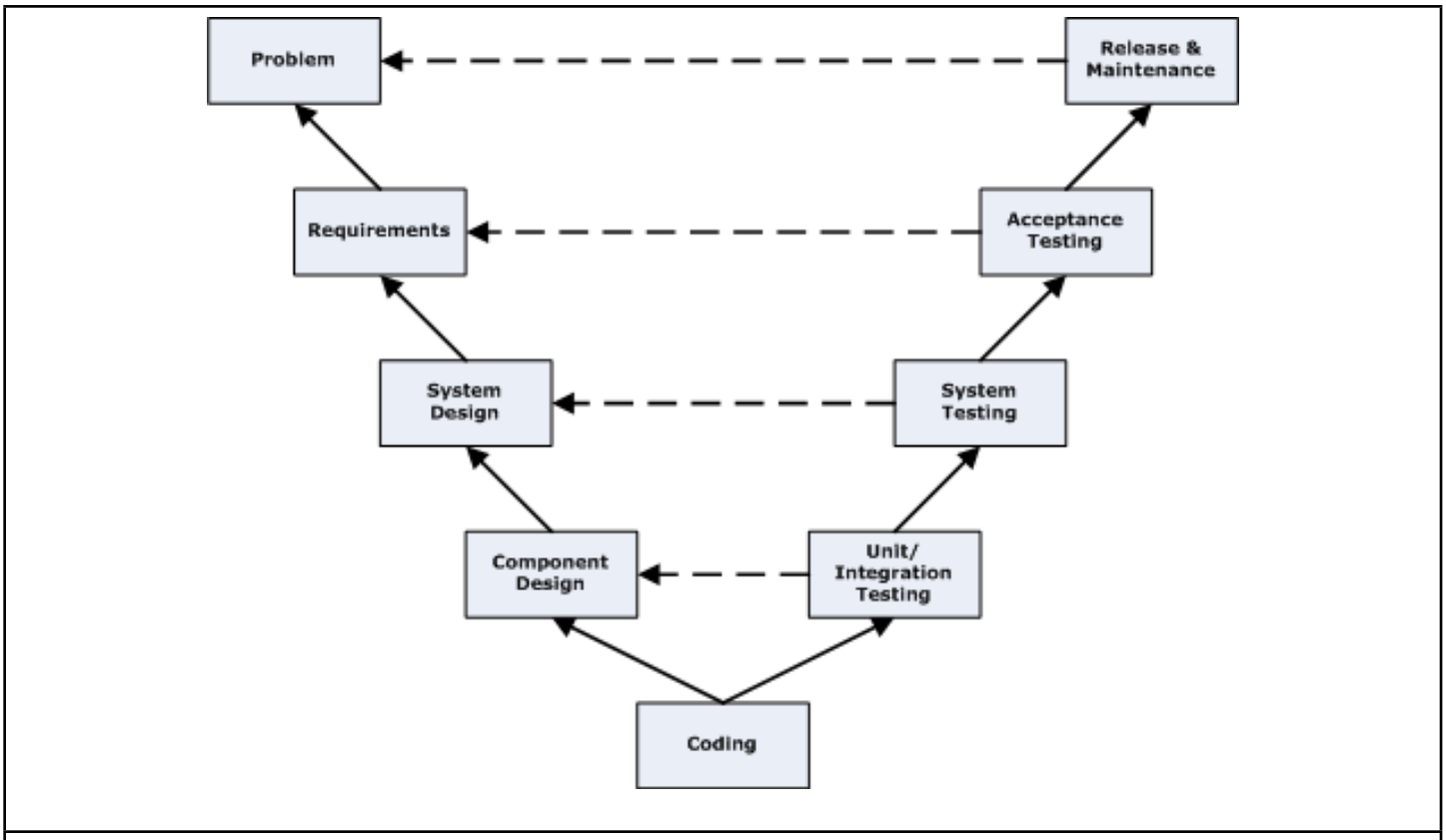

Figure 2.2. The V Model for developing software.

Iterative Models

The most significant problem inherent with waterfall-type process models is the potentially long time lapse from one activity to the next. Errors or changes that affect early phases of development are costly and time consuming. Greater efficiency can be achieved if the phases of development are interleaved with one another, or if they can be performed in cycles, or iterations, where the software is built up a little at a time. Iterative models provide two advantages over waterfall-type models:

1. They allow testing to occur earlier during the development process. Earlier testing promotes earlier detection of flaws, allowing them to be corrected while it is still cost-effective to do so.

2. Changes requested by the customer can be more easily accommodated.

The most popular process models in use today tend to be of the iterative variety. Of the many types of iterative models that have been proposed, two models that are most easily applicable to a wide range of project types are 1) the stepwise refinement model, and 2) the incremental model. Often, both models are used simultaneously for the same project.

Stepwise refinement is concerned with taking a functional component, or possibly an entire program, and refining it to make its performance more efficient, polish the look of its user interface, standardize its code, or make any other modifications that do not change the essential functionality of the component. For example, suppose one requirement of a component states that it should retrieve data from a

database, but the database is being created by another team, and it isn't complete. So that work can continue, the component could be implemented to retrieve data from a text file, which will suffice for getting the rest of the component completed. When work on the database is complete, the component can be refined to retrieve data from the database instead of the temporary text file. The essential functionality hasn't changed, but it has been refined. As another example, suppose a component is implemented with an algorithm that is simple to write, but isn't scalable to larger problem sizes. Later, once the component has been tested to make sure its other functionality works properly, the simple algorithm can be replaced by an algorithm that is more difficult to design, but scalable to any size of problem. As a final example, a command-line interface may be used until a GUI can be developed. Each cycle of the process refines the software, until eventually the software is finished. Figure 2.3 shows a diagram of a generic stepwise refinement process model.

An incremental approach, on the other hand, does not strive to include all the required functionality in the first version. Instead, the first version includes what is determined to be the most important functionality. Additional features are added in later versions. For each version, however, the functionality that is provided is completely refined. A generic incremental model is shown in figure 2.4. An incremental approach is usually prescribed for projects where the requirements are not completely known, or are volatile, meaning they are prone to frequent change. An incremental model is also very useful when building software with the capacity to be extended later. At the customer's discretion, an "incomplete" software application may be deployed early, with additional features rolled out later according to the needs of the end users.

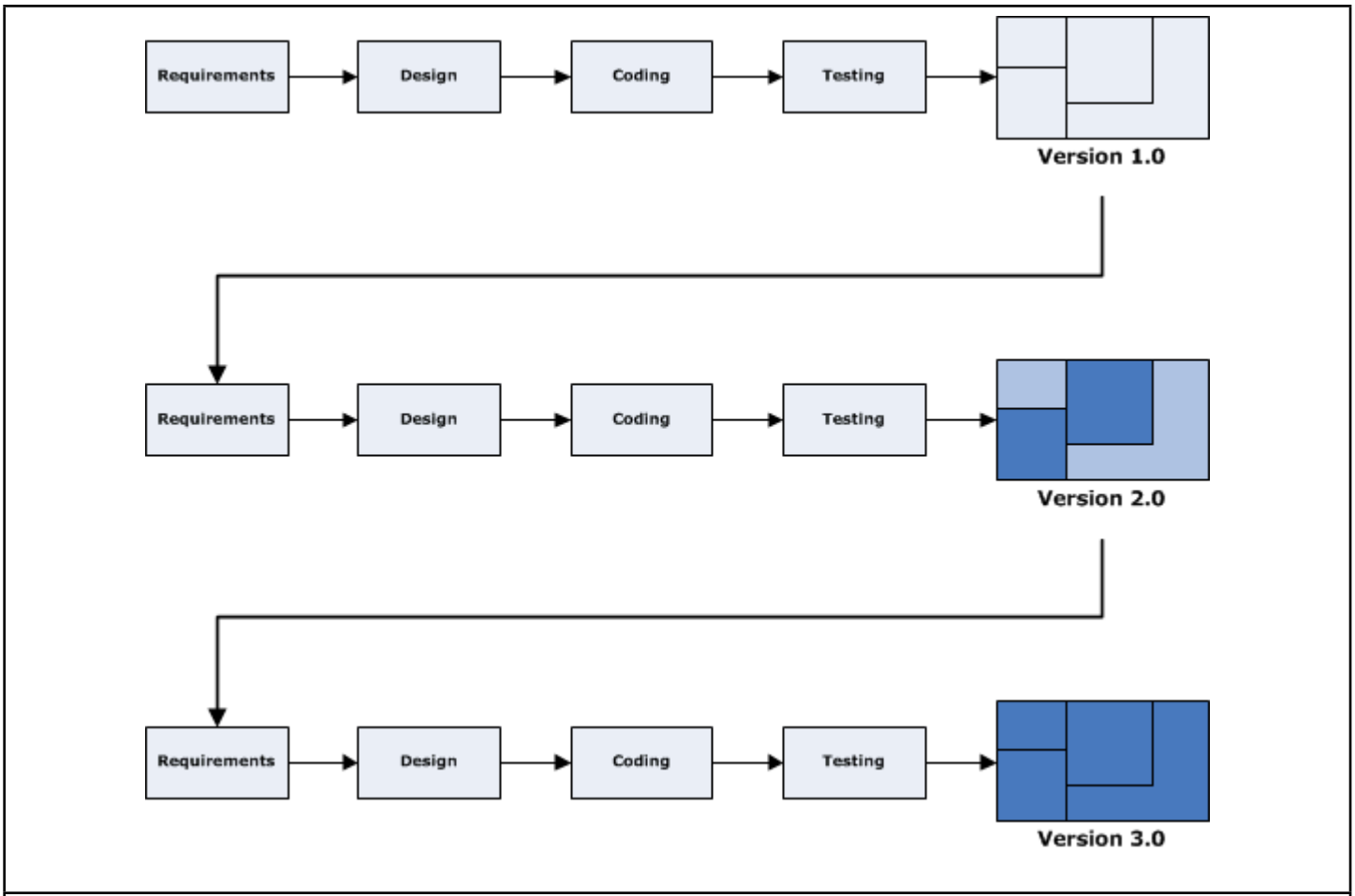

Figure 2.3. A generic stepwise refinement process model. Lighter shading indicates less refined components, while darker shading indicates more refined components.

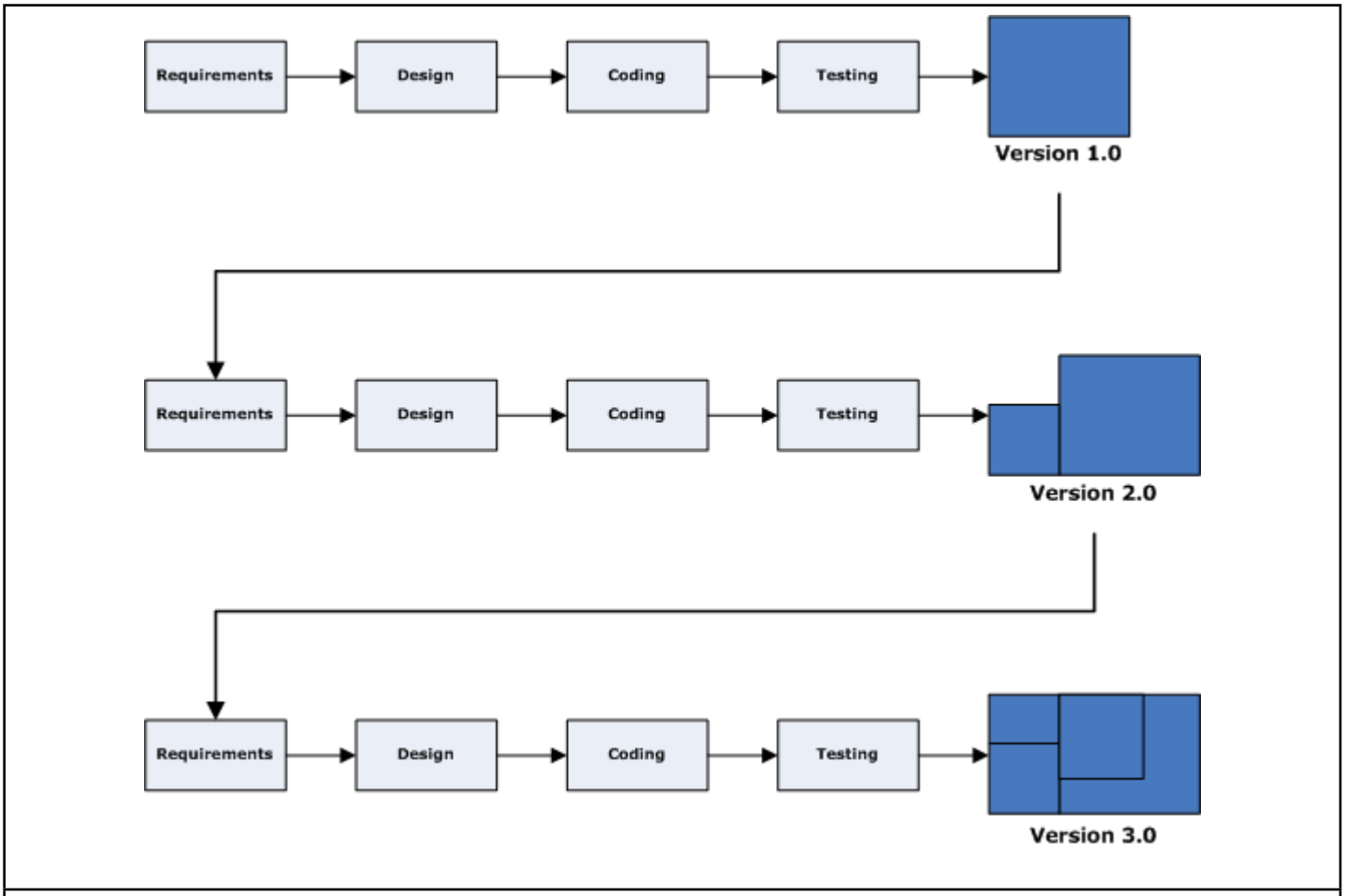

Figure 2.4. A generic incremental process model.

Prototyping

Prototyping is an iterative approach that is often used when designing user interfaces, determining the feasibility of requirements, or when the requirements are vague or may be subject to change. You don't want to build an entire system only to have the customer say, "Gaaaa! That's not what I wanted! I'm not paying for that!" A prototype of a user interface, for example, can have text fields, buttons, and other controls that don't actually do anything, but rather simulate what the user will see. The user interface can be refined independently of the work on the "back-end". Figure 2.5 shows the typical cycle for using a prototype model that incorporates customer feedback. One thing that should be noted about prototyping is that, strictly speaking, a prototype itself should not be incorporated into the final product, but it should be saved to direct the proper development of the component that it prototypes. The reason for this is that prototypes are meant to be developed quickly, so the rules for good programming practice and good design do not apply when building a prototype.

Other Process Models

There are many other process models, that have been developed. Object-oriented programming spawned the emergence of many process models aimed primarily at object-oriented concepts. Most of these have now been supplanted by the Rational Unified Process, based on the Unified Modeling Language (UML). The spiral model developed by Boehm, and discussed in the textbook, is a cyclical model that focuses on risk mitigation. Agile development, which we will talk about in a later module, uses a modified version of the incremental model to produce increments with a very rapid turnaround time.

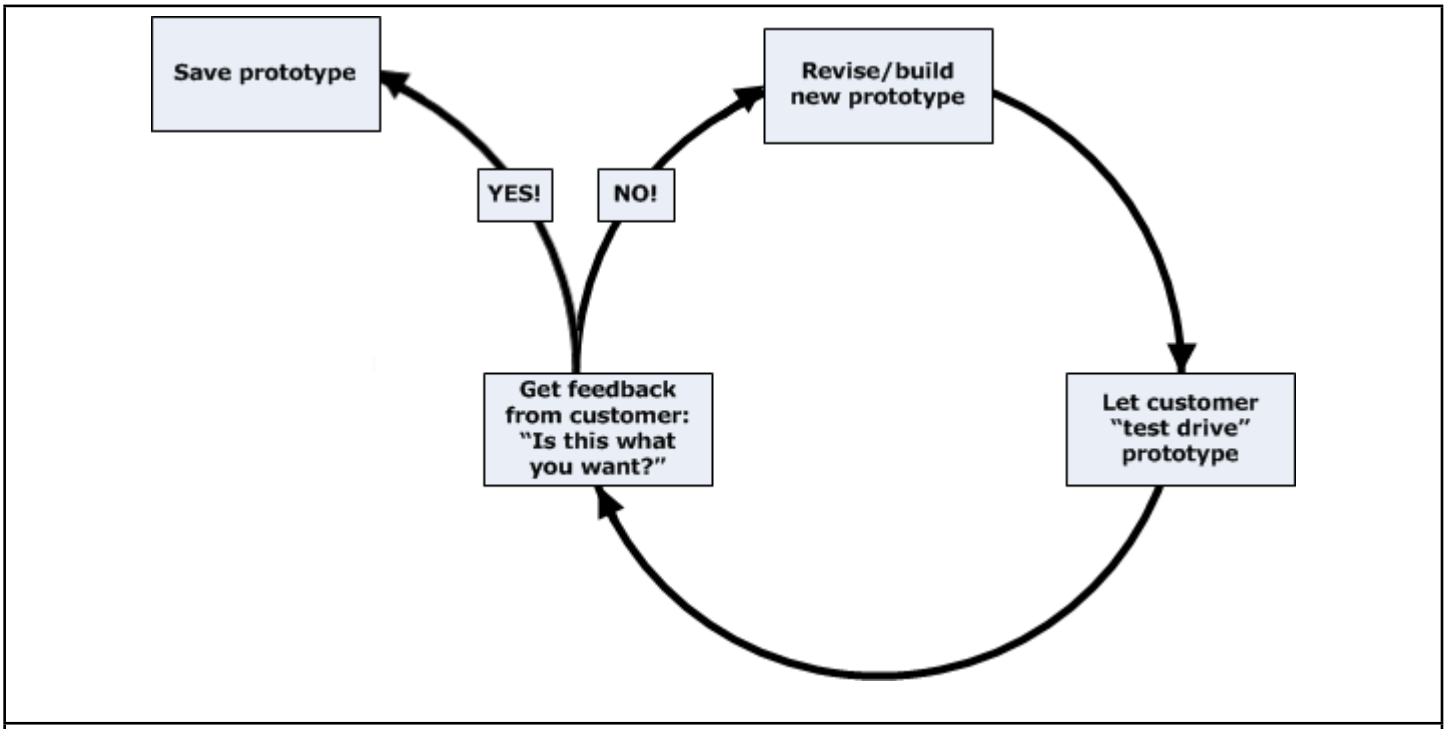

Figure 2.5. A simple prototyping model.

Project Scheduling

Planning a software project is a difficult task. It wouldn't be much of a problem if you had unlimited resources and infinite time, but that's never the case. For every project, the prospective customer is always interested in two things:

1. How much is it going to cost?
2. When can I have it?

To make matters worse, they expect you to give them an answer before you've even begun. How can you accurately estimate the cost and time required to complete the project, at the beginning of the project? Practically, it's almost impossible unless you're very experienced. We'll talk about project estimation in more detail near the end of the semester. For now, I'll focus on the scheduling aspect of project management, since for the semester project we already know the answers to the above two questions. The deadline is December 14, so it will take approximately 48 person-weeks (assuming 4 team members and 12 weeks). Cost isn't going to be much of an issue since you won't be hiring anyone or buying equipment, etc.

For this course, the project manager's primary task is to figure out how to get the project done with the available time and resources. The canonical tool for doing this is the Gantt chart, a simple example of which Sommerville shows on page 630. A full Gantt chart allows you to specify resources (people) and tasks, allocate people to specific tasks, and show how much time will be spent on each task. Software tools are available for constructing Gantt charts. Microsoft Project is one such tool, and you can get it for free though the Microsoft Academic Alliance as a computer science student at UIS (see syllabus). Project will allow you to do the abovementioned things, and more. You can specify which tasks are dependent upon which other tasks, and you can track your progress by indicating at any given time how much of each task has been completed. Based on the dependencies you have specified, you can have Project automatically create an activity diagram that shows the order in which tasks must be completed, and which tasks are dependent on which other tasks. From this activity diagram, you (or Project) can calculate the **critical path** for the project, which will tell you the minimum amount of time required to complete the entire project. The critical path always refers to the path that requires the longest time, which may or may not correspond to the path that contains the most tasks. Figure 2.6 shows an example of an activity diagram showing three major task lines and associated, arbitrary values for time required to complete each

task. In the diagram, completion of tasks H, L, and G together yield the complete project. A task's dependencies are determined by following the arrows leading into a task in reverse order. As you can see from the figure, the path from A to L contains the most tasks, yet it is not the critical path since the total time required to go from task A to task L is 21. The critical path is the path A --> B --> E --> H, with a time cost of 31. Although task H is dependent on completion of both tasks E and F, once F is completed there will be a delay while work continues on task E.

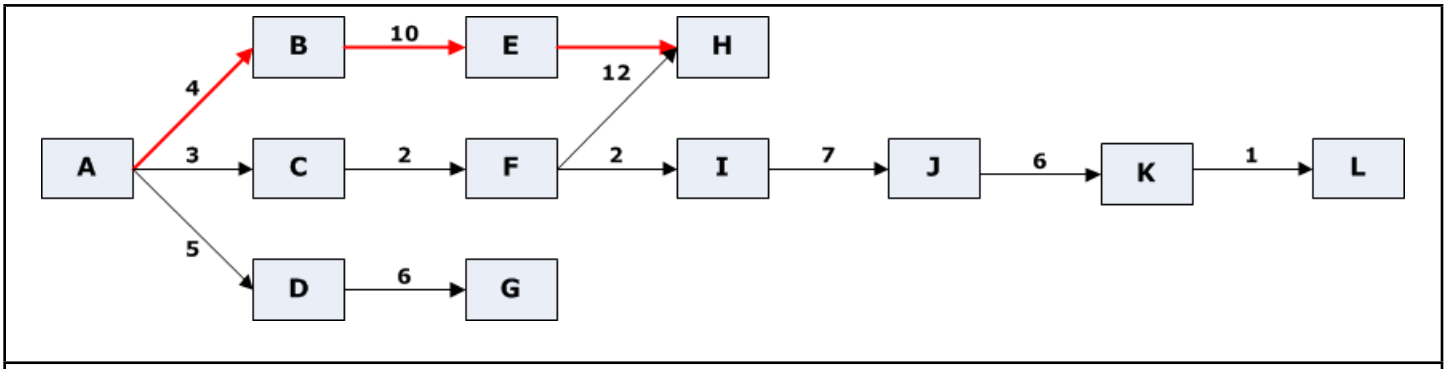

Figure 2.6. An activity diagram showing the critical path marked in red.

Project Estimation

As with any other type of prediction, when you are trying to estimate the cost, effort required, or completion time for a project, you are trying to provide a glimpse into the future. The closer you get to that future date, the more accurate your prediction will be. When you have completed a software project, and tallied the final bill, you can predict with 100% accuracy when the software will be finished (now), and exactly how much it will cost (whatever amount you spent on it). The downside is that such a late prediction is of no use to the customer.

This doesn't mean it's impossible to come up with good estimates, it just means an organization needs to have a defined process for making estimates. Usually the process involves keeping track of important details from past projects the organization has built, or has attempted to build. Failures can often be just as valuable as successes. Some examples of these details are:

1. the size of the project (e.g., lines of code)
2. technologies used (programming languages, tools, etc.)
3. difficulty level of the project
4. how many people it took to build it
5. how long it took to build it

Any detail that can be quantified in some way is called a **measure**. Measures allow for comparisons. Given any number of projects, you can compare those projects by comparing one or more measures they all have in common. For example, a project that required 50,000 lines of code (LOC) may be said to be larger than a project that required 10,000 LOC. Time spent on a project is usually measured in person-months. If you have 10 people working on a project for 6 months, you've invested 60 person-months in that project. Measures have to be considered carefully, though. Although lines of code are still widely used as a means of measuring project size, it can be misleading depending on what is considered a line of code.

A ratio of two measures is known as a **metric**. Metrics are more useful when it comes to estimating since they can be used to determine the rate at which something can be done. For example, if it takes a group of 10 programmers 5 months to write 50,000 lines of code, you have the metric, LOC/person-month, which in this example works out to 50,000 LOC/50 person-months, or 1000 LOC/person-month. All other things being equal, you now have some basis for estimation. If you can somehow determine approximately how many lines of code a project will require, you can use this metric to estimate how long the

project will take. A 100,000 LOC project should take about 10 months, whereas a 25,000 LOC project should only take about 2.5 months. Some examples of other useful metrics include:

1. cost/LOC
2. errors/LOC
3. cost/error

All measures and metrics must be used cautiously, however. They naively assume that all lines of code are equal, and do not take into account such things as the technical difficulty of a project or reused code. A project that is very technically difficult may not require many lines of code, relatively speaking, but it may take a long time to get the right lines of code. A 100,000 LOC project where 75% of the lines of code are reused from libraries and other projects will probably not take as long to complete as a 100,000 LOC project where all the code must be freshly written.

Despite the pitfalls associated with measures and metrics, they still remain our best data for estimation. Imagine if you didn't have any data from past projects, and you tried to estimate how long it would take to build a new project. All you would have to go on is your best guess, which will likely be a shot in the dark, especially if you don't have much experience with estimation. Even experienced estimators will usually come up with three estimates:

1. worst case scenario (longest time and/or highest cost)
2. best case scenario (shortest time and/or lowest cost)
3. most probable scenario (usually somewhere in between)

Providing a customer with these three scenarios is a good way to let the customer know that you can't be 100% certain how long it will take or how much it will cost. It also gives you some flexibility, since you're committing to a range time and cost rather than a specific deadline or budget.

The Great Lines of Code (LOC) Debate

Lines of code have been a standard measure for determining project size for many years. The accuracy of the correlation between LOC and effort, however, is quite controversial. For example, should comments be considered lines of code? One might argue no, since comments can't actually be executed, and thus do not end up in the executable program. However, one could equally argue yes, since comments do contribute to the understanding of the code itself, and are useful to those who must debug or maintain the code. And, comments don't write themselves, so someone has to be paid to spend the time to write them.

The programming language chosen for a project can also have an effect on how many lines of code must be written. Those of you who are familiar with Perl know that Perl's syntax allows you to cram a tremendous amount of programming power into a single, albeit gibberish-like, line of code. Languages with large libraries of reusable components also reduce the number of new lines of code that must be written.

There are two other types of measures that are often used, either in place of, or in addition to lines of code. One of these is **function points**. Function points break software down into 5 classes of characteristics:

1. # of user inputs (e.g., data entered by a user)
2. # of user outputs (e.g., results or data displayed to a user)
3. # of user inquiries (e.g., a database query)
4. # of files (e.g., a data file)
5. # of external interfaces (e.g., connections with other software)

The other type of measure is called application points. Application points break down software into three more general characteristics:

1. # of screens
2. # of user reports
3. # of 3rd generation language components

The key with both of these alternatives is that they try to characterize software based on functional units rather than lines of code. The big problem with these alternatives, though, is that some function points or application points may require significantly more code than others. In some cases, too much detail may be hidden from the estimation process. Some software may be better characterized in terms of function points, while other software may be better characterized by simple lines of code. Most available software estimation tools will work with either lines of code or function points, or both.

Estimation Models

Many models have been proposed for software estimation. All of them, as far as I know, are based on empirical data, at least on some level. This means data from past projects have been used to produce the model, as opposed to the model being produced out of proven theory. One of the best known and most extensive models is the COCOMO model, discussed by Sommerville in the text. COCOMO stands for COnstructive COst MOdel. Version 2 (COCOMO II) of the model is the one in use today. The COCOMO model makes use of all three of the size measures discussed above. It also incorporates a wide variety of other factors, including experience of the staff, familiarity of the project, and risk.

All estimation models attempt to condense a certain set of factors into an equation. Estimates can be made for future projects by simply plugging values into the equation. Usually there are one or more "fudge factors", which appear as constants in the equations. These fudge factors are designed to make up for gaps in the understanding of the software engineering process itself. Most of the popular models such as COCOMO are now supported by software tools that perform all the calculations for you - all you have to do is fill in the blanks. Most such tools are only available for purchase, but for some of them you can download trial versions. You can download a trial for the COSTAR estimation tool for free at http://www.softstarsystems.com/ (or, you can purchase a single license for the low, low price of $1900). The COSTAR tool is based on the COCOMO II model. If you have time I encourage you to play around with it to see approximately how long the project you've chosen for this course is supposed to take, according to COCOMO.

What the models and tools don't tell you, though, is how to come up with the values to plug into the equation - that simply requires experience. Not surprisingly, these models don't always work well with all projects. In fact, they usually tend to work best with projects that are of the same types used to construct the model in the first place. They may not work at all with unrelated projects.

Estimation and Project Tracking

We've mentioned that in order to make reasonable estimates, you need to have data from previous projects. But where do the data come from? The data are collected as you track a project. All the information from the Gantt charts can be used to predict how long it will take to complete specific tasks. Even if tasks are not completed on time, this is recorded, and the information can be used to identify problem areas in future projects. Many static analysis software tools exist that will count lines of code, compute cyclomatic complexity, and other measures. When this static data is combined with project tracking data, the resulting metrics provide information about the efficiency of your software engineering process.

References

1. Boehm, B.W. (1979). Software engineering: R & D trends and defense needs. In Research Directions in Software Technology (P. Wegner, ed.). Cambridge, MA: MIT Press, 1-9. (Ch. 22).