Planning: Measurement and Estimation

This is the first of two notes covering planning of software projects. This note provides an overview of planning activities, more detail on techniques used to measure attributes of software projects and on using such measures as a basis for estimating the resources needed to complete a particular project. Such measures and estimates are based on empirical observation of actual projects. The measures and estimating procedures are reasonably accurate provided they are being used on projects which are similar, in most respects, to the previous projects used to devise the estimators. For example, a group who are in the process of releasing the 20th revision of an operating system can accurately estimate the effort needed to achieve a new release of acceptable quality. Unfortunately many software projects are not of this nature. As software becomes more ubiquitous it is used in novel situations where we have no historical data on which to base estimators. In such circumstances estimation becomes an uncertain business.

Software Project Planning

In planning any software project we aim to produce a product that is on time, fit for its intended purpose and within budget. Many system development projects fail to meet some or all of these. Some widely publicised examples are:

- The London Stock Exchange TAURUS system. This was intended to automate the exchange of share certificates between share owners after deals had been carried out using the current automated trading systems. The project ran for 5 years and consumed £75 M before being abandoned since there was no prospect of it being completed in reasonable time. Currently this work is still done manually. It can take months to finalise a share transaction.

- The London Ambulance Service automated scheduling system. This was intended to provide aid to ambulance dispatchers in the London area. The system was brought into service without adequate testing and quickly demonstrated that it could not cope with peak demand. Its failure left acutely ill people waiting hours for service and it appears to have lost some calls for service. It was withdrawn after two days of service. Scheduling is still carried out manually.

- The American Express Company planned an "Information Warehouse" which would provide comprehensive management information to its executives. The project had substantial sums spent on it but was abandoned when the company discovered that the warehouse would contain terabytes of data and the project had not devised effective ways to summarise such volumes of data.

Examples such as these are the tip of the iceberg. Software projects are failing to meet their aims every day and many end in cancellation. Recent US government statistics indicate that less than 10% of software systems commissioned by the US government ever enter service. Many of these failures are attributed to failures in the planning of software projects. Because software projects are often innovative, they include unknown factors to a degree unprecedented in other engineering disciplines. Thus because we are dealing with an uncertain situation we need to plan flexibly. *Planning is a process which goes on throughout a software project. Our state of knowledge improves as the requirements and design become clearer, and then we can revise plans to allocate resources more accurately to the tasks in the project.*

Planning iterates the following stages as we move through the lifetime of the project from the feasibility study taken on before the project begins through requirements, design, implementation and maintenance. *All* the phases of planning are relevant throughout the lifetime of a project. For example, it may be sensible to narrow the scope of a system as it approaches the end of its life or it may be sensible to widen the scope of a successful project once it has been accepted by a range of users.

1. *Establish objectives and scope:* The objectives of the project are expressed in general terms and in the language of the application area *not* the system developer. The scope establishes the broad constraints of the system. This gives a characterisation of the environment (computer, human, organisational) within which the system must be integrated, the scale of the problem and the resources available to develop the system. Initially, this phase sets the boundaries for a feasibility study to see whether an acceptable system can be constructed with the given level of resources.

2. *Explore alternative approaches:* This explores the different possible approaches to solving the problem in general terms. In the early stage of the project this constitutes a feasibility study to see if such a system can be built. The alternatives should include non-computer based solutions. It may be that by making organisational changes the environment changes sufficiently that a new system is not required. For example, an airline does not need a complex seat reservation system if instead you cannot book and must just pay a crew member as you board. (Some airlines in the USA operate this kind of system. It only works on busy, frequent routes.)

3. *Establish a measurement framework:* Planning and project management need to estimate the resource required for a task. In general that will vary with the "size" of the problem. In general this is a difficult task since some seemingly small systems are hard to design, and so are "big" in resource terms, while some seemingly large systems are easy to design, and so are "small". On the face of it it seems absurd to measure the size of a problem in the likely size of a program which solves the problem. However, designers usually specialise on a fairly narrow range of types of system and on fairly routine, tried-and-tested, solution techniques. Thus for any one design team such measures are stable and predictable while across teams they may show no stability.

4. *Estimate resources required:* This involves two activities: deciding on the kinds of resource (support, design, implementation, documentation, testing); and estimating the quantity of each resource that is required for the task to be undertaken. In the early phases of a project such estimates are necessarily imprecise. As requirements analysis and design proceed then the size of tasks gets smaller and more precisely defined, and so estimates become more accurate.

5. *Risk analysis:* This involves identifying potential risks in the project (i.e. in the development process, *not* in the application area of the system — that is part of requirements capture). A risk is a potential bad event (e.g. failing to deliver a part of a system on time), a measure of the severity of the bad event (e.g. losing the contract for the rest of the project), and a likelihood that the bad event will happen. The aim of risk analysis is to identify risks and to manage the risks so the overall risk in a project is acceptable.

6. *Scheduling:* Having identified tasks and estimated the resources needed to carry out the tasks, scheduling explores the possible ways of allocating resource (which is limited) across the tasks so they all get the required resource and the duration of the phases of the project is acceptable. In general many different schedules are possible for a given project and pattern of resource availability.

7. *Tracking and control:* As a project progresses it will not necessarily keep to plan. It is important to track the progress of the project and compare it to the plan. If significant divergences arise it is necessary to re-plan to take account of the changed circumstances.

Measuring Software

Most modern books on software engineering have a chapter on "Software Metrics" — see for example Pressman [5] or Sommerville [6]. There you will find further references to the extensive literature on such metrics. Because there is little to guide the choice of metric other than empirical experience, there is great diversity in metrics. Some are simple while others attempt to capture the complexity of a piece of software by looking at its structure. Here we discuss simple metrics on which we can base estimation procedures. We consider three classes of metric:

Size metrics: This measure is based on the size of the resulting piece of software, usually in KLOC's (thousands of lines of code). From this we can derive various measure like KLOC/person-month which might be a crude measure of productivity or £/KLOC which might be a measure of cost. The problem with this measure is that it is weak in a predictive context. Unless we are in a familiar setting we have little idea of how big the resulting code will be for a given system component. However in familiar situations we can use it predictively.

Function-oriented metrics: These are used to attempt to measure the complexity of a system component in terms of its interaction with the outside world. A typical function-oriented metric might be some linear function of the numbers of inputs, outputs, files used, external interfaces and functions supported. These sorts of metrics are better for estimation purposes because they are not based on the size of code. Again the particular parameter values in such a metric are very sensitive to the kind of problem being considered, experience of the development team etc.

Quality Metrics: These measure quality attributes like correctness, maintainability, integrity, and usability — recall the attributes discussed in the Requirements part of the course.

Correctness: This is generally based on fault reports from users. Often it is measured in terms of faults/KLOC or as faults relative to some function-oriented metric. Stronger correctness requirements will raise the resources needed to generate a system. In some cases the requirement for a very low level of defects will cause a system to be abandoned. Most guidelines on the development of safety critical systems include a "do not build" category which classifies systems which have correctness requirements that cannot be consistently met.

Maintainability: This is usually measured by the mean time to effect a change in a system averaged over the defects or changes in requirement identified in the life of the system. High maintainability will also increase the resources needed to build a system.

Integrity: This is an estimate of how well a system will resist threats to its normal running. These include security breaches, power and communication disruption etc. This measure is usually based on estimating the capacity of the system to continue operation in the face of a standard repertoire of threats.

Usability: This is assessed by time trials of how long it takes users to learn to carry out standard tasks and by subjective questionnaires.

The topic of metrics is controversial. Many would argue that they are not useful outside of a narrow range of traditional software systems. However, they are widely used and are a basis for estimating the resources required for tasks in projects. Adopting them wholeheartedly requires proper instrumentation of the entire software development process in order to collect metric data. When an organisation begins using metrics they are often wildly inaccurate. As more projects are completed and data collected from the final product is compared with the estimated value of the measures used during project planning, the estimates can be refined in order to provide a better basis for planning on the next project. Thus over time a valuable body of empirical data is collected and the accuracy of planning is improved.

A consequence of this is that companies using such metric approaches are secretive about metric data because it provides a great deal of detail on their costs and how they estimate their costs in bidding for new projects. The value of such data to competitors is obvious.

So to summarise, the use of metrics on a single project is meaningless. To some extent metrics are an attempt to capture and summarise all the projects undertaken by an organisation. Thus a decision to use metrics must be taken at an organisational level and not at the level of an individual project. Engineers working on individual projects must be motivated to use past metric data and collect it on current projects in order to gain any benefit from their use.

Estimating Resource Requirements

Having established some kind of measuring framework to provide an estimate of the size of the design task at an organisational level, project engineers then go on to estimate the resources which will be required to do requirements capture, design, implementation, testing and maintenance. Resources include the *infrastructure* of training, methods, equipment and tools to support staff, the *staff* required, the *time* required and how it is to be scheduled. Estimation of the need for these resources depends on the size of the task and other features such as the type of code, experience of the team and criticality of the application area. Estimating is a risky business and is always inaccurate but some estimate of the resources required to complete a project must be undertaken throughout the life of a project to ensure it can be completed within resource constraints.

Risk in estimation increases with three relatively independent factors:

Size of problem: As size increases it becomes more difficult to accurately estimate the amount of effort required.

Variability of requirements: If the requirements are difficult to nail down and are likely to vary in the lifetime of the project then estimation becomes harder. Often projects which set out with vague goals like “raising office efficiency” run into difficulties because it is not clear what problem is being addressed.

Complexity of task relative to previous experience: Taking on a task that is much harder than anything you have previously tackled makes it much harder to estimate the resources needed.

In planning we should always attempt to reduce project risk. So strategies like narrowing the scope of projects, phasing to reduce the size of individual deliverables, and sub-contracting unfamiliar tasks to consultants, are all used to reduce project risk.

The first task in estimating is to decide what kind of resources will be required by the project. The following categories cover most of the ground:

Human: The balance of experience and expertise is important. It should be appropriate for the application area, size, complexity, difficulty and criticality of the task.

Hardware: This should provide an adequate software development environment along with a test environment which is as close to the final operating environment as possible.

Tools: There is a wide range of tools used in constructing systems addressing: project management, document processing, analysis and design, programming, integration and test, prototyping and simulation, maintenance. If these tools are properly instrumented and used consistently on projects they can be used to help in gathering metric data on the software as it is developed, both to refine the project plan and to feed back data to the estimation process to other projects.

Reuse: When a design team is consistently working in a particular application area the opportunity to amass a large library of useful software can greatly reduce the effort required to produce a system. In general we should always try to reuse rather than construct a program from scratch.

Effort Estimation

Once we have worked out the scope of a project and have chosen our measure of size we can begin to estimate the cost and time required to construct the main components of a system. This means we should identify the main parts of the system, estimate the size and difficulty of constructing them on the basis of historical data, and generate a table reflecting the estimated cost and time of the project. The table below is representative of the kind of estimate which might be generated:

Component	Optimist	Pessimist	Expect	£/KLOC	Lines/Mo	Cost	Mo
User Interface	1.8	2.6	2.3	10K	.3	23K	7.7
Database	2.9	3.6	3.3	13K	.24	43K	13.8
User Functions	4	6.2	5	15K	.2	75K	25
Logging	2	2.4	2.1	18K	.14	37.8K	15
			12.7			178.8K	61.5

Where more accurate data is available we may further disaggregate this table by taking a profile of the effort and cost for each component for each lifecycle phase. This would give us a much more detailed account in which we have a better idea of the way the effort is distributed across the development and we may also have a more accurate estimate of the eventual real resources required by the project.

This section does little more than scratch the surface of the estimation problem. There is a very large literature on estimation which is based on empirical evidence. As the field has grown, various models of cost estimation have been developed. The most well known is probably Boehm's COCOMO (COnstructive COst MOdel) [1] which at its simplest is a simple model which estimates cost and time on the basis of a simple polynomial in the size estimate of the project. The polynomial is based on 4 parameters which are derived from experience. Three different sets of parameters are used depending on the class of system which is intended to reflect the difficulty of developing the system.

Conclusion

Measurement and estimation for software projects are problematic. There is little science in the process. Measures and estimators are developed from project data and are specific to development teams and application areas. However in the construction of commercial software systems they are important processes, since they allow planning in a resource-constrained environment. In established areas where the problems are well understood and the production process is stable, the approaches described here are often effective. In more speculative and innovative areas, estimates can be wildly inaccurate.

Planning: Risk and Scheduling

This note continues our discussion of planning a software project. The topics considered in this note are risk *in projects and* scheduling *of project activities.*

In any project one is generally constrained in a variety of ways. The resources and timescales are set externally to the project (though estimation as part of planning should have made an input to this). In addition one must interface effectively to existing systems and the choice of packaged software and hardware platform for the system may be influenced by this. As a consequence the risks in projects are diverse. Failure to meet deadlines or cost limits are typical problems; others are failure to build a usable system or a maintainable system. In a commercial environment this variety of possible risks is usually reduced to a single risk, that of losing money. Whether one is concerned with this or with the variety of risks outlined above, the planning of a software project must try to reduce all risks to acceptable levels. Risk cannot be eliminated — whenever one takes on the task of making something there is the risk of failure. Planning must acknowledge this and try to control risk effectively. We consider two different approaches to risk control.

Having estimated and considered potential risks, the scheduling of effort to the tasks in hand can take place. Scheduling takes account of the available resources, their availability over time, the estimates of effort required for tasks, dependency between tasks, and their associated risks to plan start dates and timescales for each task. Techniques for scheduling are well established and there are a number of tools available to assist the process.

Controlling Risk

We begin by asking: *What is a risk?* For our purposes a risk is composed of three things:

- A potential undesirable event, e.g. failing to deliver some component on time and to specification.
- An estimate of the severity of the event. This can be dealt with either quantitatively by assessing the cost of such an event (e.g in civil engineering works on roads, the contractors "hire" pieces of road for a given cost and if there are time overruns then there is an additional cost to the contractor) or qualitatively by assessing the severity on some scale (e.g. little consequence, moderate consequences, severe consequences, catastrophic consequences).
- An estimate of the likelihood of the event. Again this can be quantitative in attempting to estimate the probability of the event happening during the lifetime of the project (e.g. on high-profile projects it may be reasonable to assume there is a 50% percent chance that a good project manager will be head-hunted by another company) or qualitatively (e.g. often, fairly often, infrequent, incredible).

The process of planning a software project is not a static business, circumstances change and this has an influence on the degree to which a project is exposed to risk. Throughout the lifetime of a project it is important to keep the risks involved under constant review. An important part of this process is the documentation of the perceived risks of a project. The following checklist suggests how one might go about incorporating risk information into the planning of a project.

Identifying potential undesirable events: Associated with each requirement of a project there is the associated event of failing to meet that particular requirement. In addition there may be many further undesirable events arising within the organisation constructing the system (for example, expending too much resource on constructing the system).

An important part of planning to deal with risk is constructing a catalogue of the most important undesirable events associated with a project. This list is not static. As the project progresses, planning becomes more detailed because there is more information and the list of events may grow. As parts of the system are delivered some events disappear from the list because they cannot now occur. The environment (human, hardware, software, organisational) may also change, eliminating some potential events and introducing new ones. An important part of planning is to review the potential list of undesirable events continually as the project progresses.

Estimating risk: For some collection of identified undesirable events one then goes on to estimate the severity of the event and its likelihood. This converts the events into risks. For some risks the severity can be estimated numerically from data collected by the organisation. For example, the cost to banks of accounting errors in their systems is fairly accurately known and can be predicted. In contrast, the severity of some risks can only be dealt with qualitatively using engineering judgement. For example, the failure of a message switch in a telephone network could have disastrous consequences (e.g. loss of 999 calls) but estimating the cost of such a failure is impossible. Similarly, for frequently occuring events statistical analysis can provide accurate information on the likelihood of occurrence (e.g. the frequency of bugs introduced by a long-established programming team). By comparison, estimating the likelihood of infrequent events (e.g. the major failure of a nuclear plant) suffers from a paucity of data and one can only estimate this by expert judgement.

So estimating risk involves working with various kinds of empirical estimators, for example: the likelihood that working days will be lost through sickness, the likelihood of system failure, the likelihood of catastrophic loss of data, times taken to produce similar systems with a similar project team.

Reducing risk to acceptable levels: Given some empirically-based method for estimating the likelihood of risk, the importance attached to the risks, the amount of available resource, and the possible ways of organising that resource, one can consider various different ways of using the available resources to bring the level of risk to within acceptable levels. For example, if a system is intended to support a one-off event (e.g. a booking system for a large sporting event), the importance of deadlines is very great and it may therefore be important to neglect certain aspects of quality (e.g. the user interface) to ensure it is delivered on time. Another system (e.g. one which is safety critical) may need to be externally certificated and so much more of the project resource will be spent in documenting the development process to make the safety case. Planning to take account of risk involves attempting to reduce all risks to acceptable levels, giving priority to those perceived to be of greatest importance.

Revising estimates: Planning takes place throughout a project. One should revise estimates of risk on a continuous basis. Whenever there is any change in circumstances, e.g. user response to a delivered component, some change in the external environment, a change in the project team or a failure in computer systems, one should re-examine the priorities attached to risks, and the estimates of their likelihood. On the basis of these revisions the project plan should be revised to bring all risks to acceptable levels. For example, if illness in the programming team means that a deadline will not be met, one can take a variety of approaches: bringing in more programmers; renegotiating the deadline; or simplifying the functionality of the system required at the deadline. Each of these is constrained by considerations like the availability of resource, willingness to reschedule etc. In the end it may be necessary to accept higher levels of risk because of changes that are beyond your control.

Some authors see controlling risk as the most important factor in controlling a systems development project. For example, in the Spiral Model [2] we saw earlier in the course, systems develop through a number of prototyping stages. Before a new stage is initiated the risks are estimated and a plan to reduce risks to acceptable levels is put in place, or the project abandoned if the risks are unacceptable.

Conventional Planning

The conventional approach to controlling risk in the development process is to adopt a structured plan to guide the development process. An example outline structure of a conventional software plan (drawn from Fairley [3]) is attached to this note. There is a pointer on the course web page to avionics standard DO-178B, which illustrates how the kind of planning espoused by Fairley is embodied in a software standard. It also provides insight into the level of detail required in large-scale software projects.

The essence of this approach is to set targets for certain attributes of the development process — e.g. predictability of cost, quality of product, satisfaction of staff involved, etc. — independent of a particular project and on that basis devise a series of project phases which will achieve those levels with acceptable likelihood. So, for Fairley, any project has a predictable structure which is given in a number of phases (as reflected in the lifecycle model) and each phase must be planned. Each phase also has a structure and that structure is dictated by the documents which are expected to be produced during the phase.

Looking at Table 2.6 we see how the phases of a typical project are set out in terms of the plans they require. Looking at Table 2.9 we see the structure of the Software Verification phase. The situation is similar for each phase of the development process.

This approach is typical of planning systems development projects and has been successful in controlling the systems development process. However, it does suffer from some deficiencies, the principal one being that it is rather monolithic. It tends to set rather large goals (or at best a small number of large, phased, goals) and direct large quantities of resource to generating a system which is not subjected to the operational environment until a great deal of work has been committed. This is an inherently risky process because it fails to recognise a whole class of risks related to misunderstandings between the developers and the eventual users. Recently a number of approaches have been developed which attempt to reduce the risk in the development process by adopting so-called *evolutionary planning.* This attempts to reduce the risk of failing to meet the needs of the user by adopting a much more flexible approach to planning.

Evolutionary Planning

This is a sketch of the approach to planning advocated by Tom Gilb [4]. A number of other authors also advocate this approach, arguing that the planning of system development should focus on reducing the risk in the development process. The evolutionary planning approach accepts that the planning process is dynamic and plans should be adjusted continuously throughout the design process to reduce risks as much as possible.

For Gilb, planning and development are bound together and proceed hand in hand. The basic development/planning cycle is the following:

- Deliver a component to the user. This should be small, usable and comprehensible by the user.
- Measure the impact of the component on the environment in which it is used.
- Adjust the design objectives to maximise the benefit of the next stage of the project while keeping risks within acceptable level.

Thus a plan for Gilb has a number of components:

- A dynamically changing collection of system components which can be delivered to the user. Design objectives consist of system components and targets for the values of particular attributes of the components.
- A means to measure important attributes of system components and their impact on their operating environment. These means of measurement may also change as the project progresses.
- A strategy for adjusting the design objectives on the basis of the measures of the delivered system components and the operational environment.

Here is Gilb's list of "features" of his approach to design:

- Systems developers should have multiple, heterogeneous, objectives — not just the functionality of the system. Part of the skill of developing systems is to keep all objectives in mind.
- Developers should iterate through the design and deliver cycle early and frequently to gain direct feedback on how well the system is meeting the user's needs. This also reduces the commitment of the developer to any particular design decision, thereby reducing risk.
- At each step the developer should go through a complete Analysis, Design, Build and Evaluate cycle. By going through all these aspects the possibility that risky assumptions get built into the system in an irrevocable way is reduced.
- The development process should be user oriented.
- The development process should be oriented to generating early and measurable results, i.e. effects on the operational environment.

Scheduling

Whatever the approach to managing risk, in any large project there will be a large number of tasks which must be coordinated in order to complete the project. If one is using conventional planning then the structure of the tasks will change slowly. In evolutionary planning one may expect to see a more rapidly changing situation. An important part of any planning process is deciding on resource allocation and the scheduling of tasks.

In any project some tasks will depend on the completion of others in order to allow them to commence. Identifying task dependencies is an important part of scheduling since this is the primary constraint on the order in which tasks are carried out. Task dependencies are usually represented by a directed acyclic graph where an edge denotes dependency and the vertices are the tasks. Figure 1 shows a small example of a task dependency graph (for 8 tasks named T1 ... T8).

Each task is also labelled with an estimate of the number of person-months required to complete it. If for the moment we assume that we allocate one person per task then these are durations.[1] If we now explore the lengths of path through the dependency graph we can identify a longest path (usually known as the *critical path*) which defines the minimum

[1] Of course if we allocated two people to each task this would *not* halve the time taken because of communication time and the internal structure of the task. See F. Brooks, *The Mythical Man-Month*, Addison-Wesley, 1975 for a discussion of this. On the basis of his experience with the management of very large software projects at IBM, Brooks says that adding people to a software project sometimes makes it take *longer* to complete.

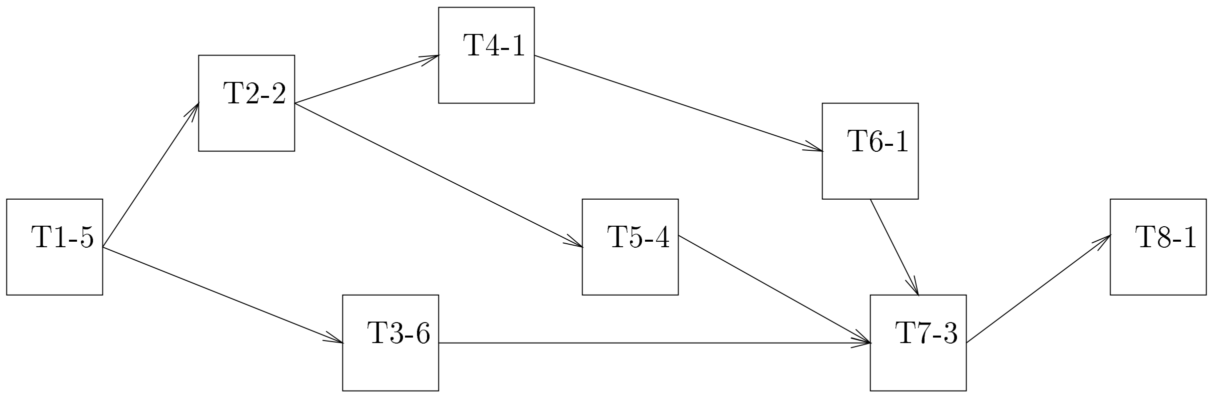

Figure 1: Task Dependencies

time the project can take to complete. Here such a path is T1, T3, T7, T8 and the minimum duration is 15 months. Of course if we allocate different levels of resource to different tasks we change their durations and the critical path may change. This kind of analysis is used to consider how different resource allocation strategies influence time to completion.

Once we have decided on the allocation of resources we can construct a project timeline. Figure 2 is an example of a project timeline for our sample task dependency graph. Notice

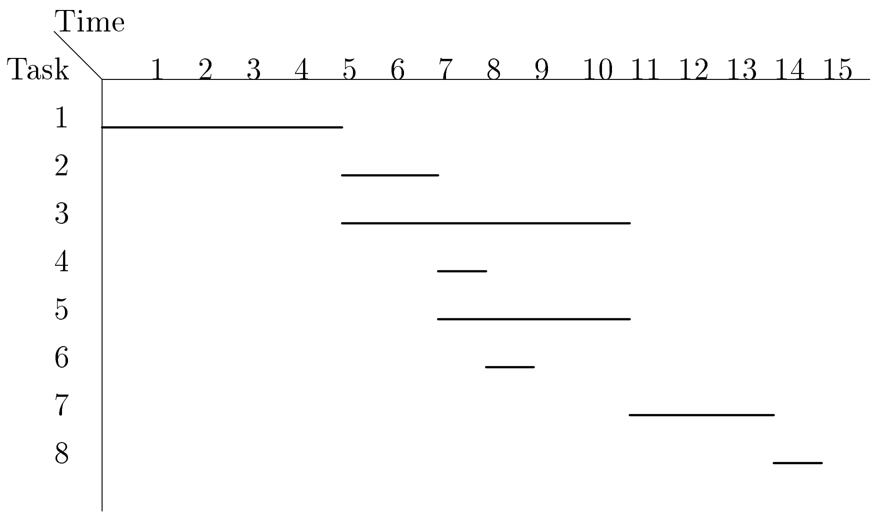

Figure 2: A Project Timeline

that only those tasks on the critical path are fully constrained on start time. Those off the critical path have some flexibility in start time. For example T4 and T6 can be started at a number of different combinations of time.

The aim in constructing a project timeline is to try to match resources needed on the project with their availability. Often the aim is to try to have a roughly similar need for resources throughout the project. But in some circumstances it may be appropriate to allow the size of team to begin small, grow to a maximum size, then diminish to the maintenance team.

Tracking and Monitoring

Planning is only effective if it is based on accurate and timely information. It is important to establish procedures to track and monitor the progress of projects. Frequent small scale milestones allow the early identification of divergence from the plan and small corrective actions can be taken promptly but these have a cost in effort required from the project team. Longer intervals with larger milestones require less effort on the part of the team but may result in a loss of effective control of the project.

References

[1] B.W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981

[2] B.W. Boehm. A Spiral Model of Development and Enhancement. *ACM SIGSOFT Software Engineering Notes* 11(4):14–24, 1986.

[3] R. Fairley. *Software Engineering Concepts.* McGraw-Hill, 1985.

[4] T. Gilb. *Principles of Software Engineering Management.* Addison-Wesley, 1988.

[5] R. Pressman. *Software Engineering: A Practitioner's Approach*, 4th edition. McGraw-Hill, 1996.

[6] I. Somerville. *Software Engineering*, 5th edition. Addison-Wesley, 1995.