CHAPTER

Use Case Based Project Planning e4

4.1 Introduction to effort estimation and risk analysis in software projects

The motivation for effort estimation in software projects comes from the fact that historically most software development projects:

- Take more time to complete than expected.
- Cost much more than expected.
- Generate a product that does not have the quality expected.
- Generate a product that does not have the scope expected.

Thus, the point is that software development teams and clients must learn to expect the right time, cost, quality, and scope regarding their project and their team. Effort estimation techniques are used to accomplish these goals.

4.1.1 Ad hoc techniques

A number of techniques for effort estimation have been proposed and used in the last several decades. One that is very popular is the *Smith* technique, which may be summarized as "Smith, tell us how much time you need to develop that system!" A variation of the Smith technique is the *constrained Smith* technique, which may be summarized as "Smith, you have three months to develop that system!" Although popular (and known by other names, such as Susan, Peter, Mary, etc.), this approach is not very accurate.

Another ad-hoc technique that is used is the *six months* technique. It consists of estimating "six months" for any software project at the beginning of its development and then adjusting it up or down as the scope and requirements are discovered. Variations of that technique are virtually infinite, including the *One Year* technique and the *Eighteen Months* technique, among others.

These kinds of techniques are known as *nonparametric estimation* because they are not based on measures on the project to be developed. Somerville (2006) identifies some other nonparametric techniques:

- *Expert judgment*. The expert judgment technique proposes that one or more experts on the project domain and software development should meet and use their experience to produce an estimate. The Smith technique is a potentially bad scenario for the expert judgment technique: the technique may be very inaccurate, depending on the expertise of the estimators. However, if the experts really have experience in similar projects it is a feasible, quick, and relatively cheap technique.

- *Estimation by analogy*. That technique is in fact the pragmatic basis for the expert judgment technique. It is assumed that the effort for developing a new system will be similar to the effort for developing other similar systems. The technique is not feasible if similar projects are not available for comparison. It can make good use of one or more experts in order to determine what qualifies as a *similar* system and how the time spent to develop it may be used as a basis to estimate the effort for the current project. Thus, this technique usually may evolve to expert judgment.
- *Parkinson's Law*. That technique is not usually adopted openly, but it is recurrent in software projects and in other areas: *the project costs whatever resources are available*. The advantage is that the project will not cost more than expected, but frequently the scope of the system will not be completely covered.
- *Pricing to win*. The cost of the project is equal to the price that it is believed will win the contract. The disadvantage of this method is that the system the client gets may not be what she expected, because as costs must usually be reduced, this may impact quality and scope. However, when there is no detailed information about the system and the team lacks experience on more advanced estimation techniques this technique may be chosen: the project cost is agreed based on common understandings and the development is constrained by the cost.

The agile community usually adopts an estimation strategy that is based on story points. A *user story* is a scenario of use of a system, and it is somewhat similar to a use case. However, use cases are more structured and formal; they are produced by a team of analysts. User stories, on the other hand, must be written by the users themselves. They must represent a scenario where the users see themselves using the system that is going to be produced. The idea behind this is that the users present the needs that are most important for them first; the details would be remembered later.

One *story point* is considered to be an *ideal working day* (6 to 8 hours focused and dedicated to a project). The estimation effort is then calculated for each user story in terms of story points. The question to be answered by the team is, "How many ideal days *x* people would take to develop a given user story?" If the answer is *x* people would take *y* days, then the number of story points is x^*y. Story points assignment is subjective and may be considered a nonparametric technique. The team usually evaluates the effort based on their experience on similar past projects. The assignment is made based on effort, complexity, and risk. For example, the following questions could be asked:

- *Complexity*: Does this user story have many possible scenarios?
- *Effort*: Is the change going to be made in many different components?
- *Risk*: Does anyone in the team know the technology necessary to develop this story?

Another way to estimate user stories is to place all user stories on cards on the table. Then, the simplest ones to develop are separated and one story point or less is assigned to them. After that, the simplest of the remaining stories are separated and two story points assigned to them. This procedure is repeated until no more user stories remain on the table. It is suggested that each iteration increase the number of story points by following a series similar to the Fibonacci series such as 0, ½, 1, 2, 3, 5, 8, 13, 20, 40, 100, etc.[1] The reasoning behind this is that for an average human it is

[1]In the actual Fibonacci series, each number is the sum of the previous two. Thus, the original series is 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, etc. However, for estimation purposes an approximate series is used because as numbers grow the estimation is less accurate.

hard to estimate how much a horse weighs; however, most people will agree that a horse weighs more than a dog and less than an elephant.

If a user story is determined to take less than one ideal working day, it must be attached to another story, and stories that are above a given limit (for example, 10 story points) must be split into simpler stories.

4.1.2 **Parametric techniques**

Another class of estimation techniques includes the parametric ones which aim to present an effort estimation based on the estimated size of the system, the technical difficulty to develop it, and the capacity of the team, among other features. The size of the system may be measured in terms of:

- *Lines of code*. Techniques such as COCOMO II – COnstructive COst MOdel II (Boehm, 2000) start estimating how many *source lines of code* (SLOC) the system will have. In fact, as most systems have thousands of lines of code, the usual measure is *kilo source lines of code* (KSLOC).
- *Function points*. Techniques based on function point analysis (FPA) (Albrecht, 1979) and its variations are not concerned about lines of code. They estimate that the effort to develop a system depends on the *apparent functionality* that is going to be implemented. Thus, usually these techniques count the number of functional requirements and evaluate their complexity by asking how many classes or data records they need to use in order to be implemented and how many arguments a user sends or receives from the system in order to perform the function.
- *Use case points*. That is a variation of the function point analysis technique that is based on use cases instead of functional requirements for estimation. This technique is explained in detail in Section 4.2.

Although COCOMO II is the most detailed technique it has a disadvantage compared to function points and use case points: the number of KSLOC must be estimated by specialists before applying the method, and this is a significant source of uncertainty.[2] Function points and use case points are based on requirements, which may actually be known at the beginning of a project (of course requirements may change during development, but when they change estimation may be immediately updated in a systematic way).

In addition to the size of the system measured in lines of code or apparent functionality, the parametric techniques also make use of *technical adjustment* factors. It is assumed that implementing a function of the same size may be easier or harder depending on the kind of system that is being developed. For example, an information system like Livir is more straightforward to implement than a robot surgery system or a core system embedded into an airplane. The same amount of source lines of code may take more or less time to develop depending on the technical complexity of the system.

The most complete technique regarding technical adjustment factors is COCOMO II, which proposed the use of up to 16 effort multipliers divided into four groups:

- *Product attributes*. These include the reliability required for the software, the relative size of the test database, the inner complexity of the product, development aiming at reusability, and the quantity of documentation expected and needed.

[2]To minimize this uncertainty, *backfire tables* that convert function points into KSLOC may be used (Boehm, 2000).

- *Hardware attributes*. These include aspects related to time and space efficiency and the volatility of the development platform.
- *Personnel attributes*. These include the capacity of analysts and programmers, personnel continuity, experience on similar applications, and experience on the platform, programming language, and other software tools.
- *Project attributes*. These include the use of computer-aided software engineering (CASE) tools, distribution of the development team, and the need to accelerate the development schedule.

Use case points and function points have similar effort multipliers but with a different organization and sometimes a different interpretation. In both methods, the effort multipliers are a numeric index that is multiplied by the estimated size of the system to produce effort estimation. However, COCOMO II has a unique feature: a set of *scale factors*. There are five such factors:

- *Precedentedness*. Is the product similar to others that were already developed by the same team?
- *Development flexibility*. Does the product have to be developed straight to the requirements or are the requirements only general goals?
- *Architecture and risk resolution*. Is there good support to resolve risks and architectural issues?
- *Team cohesion*. Have the team worked together before? Are they nicely integrated?
- *Process maturity*. What is the maturity of the organization? CMMI[3] (SEI-CMU, 2010) and SPICE[4] (Emam, Melo & Drouin, 1997) may be used as a measure of maturity. COCOMO II also proposes a simple questionnaire that helps to estimate the maturity of the organization if other assessments are not available.

The difference here is that those scale factors are not multiplied by the system size: they are used in an *exponential*. Average values (near 1) for these factors means a nominal scale where each line of code produced costs the same regardless of the size of the software. If the scale factors are bad, then their numerical value is greater than 1 and this means that a line of code in a bigger system costs more than a line of code in a smaller system. If the scale factors are good, then the team gains in scale, that is, a line of code in a bigger system costs less than a line of code in a smaller system.

4.1.3 Risk analysis

A *risk* is something that may happen and cause trouble.[5] Risks are very frequent and numerous in software projects. There are different categories of risk and they have different impacts: there are some that are just a nuisance and others that may cause the termination of the project. Usually the process of dealing with risks involves:

- *Identification*. The team must discover and be aware of the conditions that may trigger problems in a project. Checklists such as the one by Carr et al. (1993) may help the identification process.

[3]Capability Maturity Model Integration.
[4]Software Process Improvement and Capability dEtermination.
[5]Some definitions also consider *positive* risks as situations that may happen and help the project.

- *Analysis*. Risks are analyzed so that some of their attributes are discovered. Usually at least two attributes must be identified: the *probability* of the occurrence of the condition that triggers the risk, and the *impact* that the risk has on the project. The *exposure* or *importance* of a risk is a combination of probability and impact. Usually risks with high probability of occurrence and high impact on the project are the most important.
- *Planning*. Once the exposure of the risks is assessed, plans must be made for medium- and high-exposure risks and the project must include activities to minimize the probability and/or impact of the most important risks. These plans are the *mitigation plans*. Moreover, *recovery* or *contingency plans* should also be prepared for the most important risks and they will be executed if, despite the mitigation activities, the risk still becomes a problem.
- *Control*. The risks of a project are even more unstable than its requirements: they change frequently and unexpectedly. New risks are discovered during development and their importance may change. Thus, the project manager must be aware of that and pay close attention to the status of the risks during the whole project.

Different sources of risks may be identified. Carr et al. (1993) presents the following structure:

- *Product engineering*:
  - *Requirements*: Are they stable, complete, clear, valid, and feasible? Is the project something that has never been made before? Is the project bigger than any other the organization has undertaken before?
  - *Design*: Is it difficult to obtain? Are the interfaces with other systems clear and easy? Are there performance, testability, and hardware constraints? Will software developed outside the company be used?
  - *Code and test*: Is the implementation feasible? Is the time reserved for testing enough?
  - *Integration and test*: Is the integration and test environment adequate?
  - *Engineering aspects*: Will the software be easy to maintain? Are the reliability, safety, and security requirements easy to obtain and verify? Are there special human factors to be taken into account? Is the documentation adequate for the project?
- *Development environment*:
  - *Development process*: Is there a clear and proven development process in use? Are the plans formally controlled? Is the process adequate for this kind of system? Do the members of the team know and use the process? Are there clear mechanisms for managing changes to the product?
  - *Development system*: Is there adequate hardware and other tools for the development of the project? Are the tools easy to use? Does the team know how to use the tools? Are the tools reliable?
  - *Management process*: Are the plans adequate? Do they include contingency plans? Are roles and authority inside the project well defined? Do the managers have experience? How does the team communicate with other stakeholders?
  - *Management method*: Are there metrics to help management? Are the project personnel trained? Is quality management adequate? Is the configuration of code and other artifacts controlled?
  - *Working environment*: Is the team concerned about quality? Is the team cooperative? Is communication among team members good? Is the morale of the team high?

- *External constraints*:
  - *Resources*: Is the schedule reliable and stable? Is the team experienced and capable? Is the budget adequate? Are the facilities adequate?
  - *Contract*: What kind of contract exists? Are there contractual constraints? Does the project depend on external partners?
  - *Communication interfaces*: Are there problems with the client such as weak communication or incomplete knowledge about the domain?
  - *Subcontracted personnel*: Are there problems with partner companies such as weak interfaces, poor communication, or lack of collaboration? Does the project depend on them?
  - *Main contractor*: If you are a subcontracted company, are there any difficult issues regarding the main contractor?
  - *Management*: Is the higher management absent or micromanaging?
  - *Sellers*: Do the product sellers you are using solve issues quickly?
  - *Policies*: Are policies creating trouble for the project?

This list is just a reference; many other unpredictable risks may appear in most projects.

After being identified, risks must be analyzed. Many risk attributes may be identified. Among them, maybe the most important are:

- *Probability*: The chance that a risk really becomes a problem. If there is a historical series of data, risk probability could be measured numerically. However, most risks are assessed using the *t-shirt scale*, where "high" means that it is almost sure that the risk will become a problem, "medium" means that it is expected that the risk will become a problem, and "low" means that the chance for the risk to become a problem is minimal.
- *Impact*: The price to pay (monetary or not) when the risk becomes a problem. Again, numerical measures may be used if sufficient information is available. For most projects, risk impact could be assessed as "high" (may terminate the project), "medium" (may raise the project's costs significantly), and "low" (may cause some trouble, but losses are easily recoverable).
- *Importance* or *exposure*: The combination of probability and impact, as expressed in Table e4.1.

High- and medium-exposure risks should have mitigation plans for reducing the probability and/or impact. In the case of high-exposure risks, it is advisable to execute the plans as soon as possible, while medium-exposure risks may be addressed later. Thus, risk mitigation activities will be included in the project effort at this time. Low-exposure risks may just be monitored in order to detect if their exposure changes during the project.

Table e4.1 How to Calculate the Exposure of a Risk

		Probability		
		High	**Medium**	**Low**
Impact	**High**	High exposure	High exposure	Medium exposure
	Medium	High exposure	Medium exposure	Low exposure
	Low	Medium exposure	Low exposure	Low exposure

Among other possible definitions, it may be said that a risk is composed of three parts:

- *Cause*: An uncertain condition that may create trouble.
- *Problem*: A situation that could occur given the uncertainty of the cause.
- *Effect*: An impact over one or more of the project's goals.

Thus, there are three kinds of plans that may be associated to risks:

- *Mitigation plan for reducing risk probability*: It must act on the *cause* of the risk, reducing the probability of its occurrence.
- *Mitigation plan for reducing risk impact*: It must act on the *effects* of the risk, reducing the cost of recoverability if the risk really becomes a problem.
- *Contingency plan* or *disaster recovery plan*: It states what to do after the risk actually becomes a problem, that is, it acts on the *problem* created by the risk.

Mitigation plans are preventive: they should be executed before the risk becomes a problem. Contingency plans, on the other hand, are performed only if the risk really becomes a problem.

Table e4.2 presents an example with some risks related to the Livir project and a simple statement of mitigation and contingency actions. More detailed plans could be elaborated, though, if necessary.

Risks must be monitored while the project is being developed. As requirements usually do, risk may change over time. Even their names and properties may change. That is why risks are identified by a code: to keep track of them even when they change.

4.2 Use case point analysis[6]

Once system use cases have been identified, the effort necessary to develop them may be estimated. Furthermore, the use cases must be prioritized so that the most important use cases are developed first. In iterative processes such as the Unified Process, a relatively small set of use cases will be developed at each iteration. Thus, planning a project using this kind of process consists of determining a general plan or *phase plan* and a set of *iteration plans* that determines which use cases or risks will be addressed at each iteration. This is why it is important to determine a project's priorities and estimate the effort necessary to develop them. During Inception, only the phase plan and the plan for the first iteration of Elaboration are built. Each further iteration is planned while the previous one is being performed.

Phase and iteration planning are typical project management activities. Effort estimation is a planning tool that indicates, among other things, how much time and personnel is needed by the project.

The technique known as *use case points* (Karner, 1993) is based on *function points* (Albrecht & Gaffney Jr., 1983), specifically on the *MK II* or *Mark II* counting technique (Symons, 1988), which

[6]Readers not interested in knowing the details on how to estimate effort and develop a project plan may want to skip from here directly to Chapter 5 where object-oriented analysis and design techniques continue to be explained.

Table e4.2 Example of Risks and Plans for the Livir Project

Code	Risk			Risk Attributes			Probability Reduction Plan	Impact Reduction Plan	Contingency Plan
	Cause	Problem	Effect	Pr.	Imp.	Exp.			
K1	Unstable requirements.	Key requirements may change during development.	Time lost developing parts that will not be used.	H	H	H	Take special care during requirements elicitation. Study similar products. Develop prototypes for requirements validation.	Emphasize modular development. Implement an efficient version control system.	Use a change management system to adequately control requirements changes.
K2	The team is inexperienced.	Bad design decisions may be made.	Code hard to maintain and full of defects.	H	M	H	Hire an experienced software architect to coordinate and validate all design decisions.	Use automatic testing extensively to find defects.	Refactor code to improve design.
K3	The team is not familiar with the communication protocol with credit card operators.	Wrong or inefficient communication interface with credit card operators.	Errors in credit card operation.	H	L	M	Develop a throwaway prototype for studying the communication protocol of the credit card operator system.	Implement double-check procedures for credit card operations based on invariants and postconditions.	Double-check security on credit card operations. Interrupt operations with credit cards until the problems are fixed.

is a relatively simpler approach than that of the IFPUG (*International Functional Point Users Group*).[7]

Use case point analysis is simpler than function point analysis, but it still suffers from not having strong reports on its application to a wide number of projects, such as function point analysis and COCOMO II do. Another problem with use case point analysis is that the team comprehension on what constitutes a use case may produce disparate estimations. COCOMO II suffers from a similar problem because it is based on a crude estimation on the number of lines of code to be produced, but it allows estimators to use a *backfire table* (Center for Software Engineering USC, 2000) to convert function points into lines of code (with different conversion rates depending on the language used). Moreover, function point analysis also depends on the capacity of the estimators and the capacity of the analysts to identify the correct and complete set of requirements. The conclusion is that each technique has a weak point; however, a team that is well tuned, with historical records on estimates for past projects, and that applies the same patterns and conventions when developing software would produce better estimations than an inexperienced team, regardless of the estimation method they use. All methods depend on a good estimation effort history and the quality of the parameters obtained for the current project (lines of code, functional requirements, or use cases).

The technique of use case point analysis was incorporated to the Unified Process because that process is strongly based on use cases. As said before, one of the biggest difficulties to applying it is not the method itself, which is very simple, but the lack of standardization on what is effectively a use case (Anda, 2001). Due to the absence of an international standard, the team may create and use its own understanding of the adequate granularity for use cases.

Use case point analysis is based on the quantity and complexity of system actors and use cases, which corresponds to the *unadjusted use case points* (*UUCP*) value. The application of technical and environmental factors leads to the *UCP*, or *adjusted use case points*.

UUCP is calculated from the estimated complexity of the actors (*UAW — unadjusted actor weight*) and use cases (*UUCW — unadjusted use case weight*), as discussed in the next subsections.

4.2.1 UAW — unadjusted actor weight

The method for assigning complexity to actors is quite simple. Each actor is counted once, even if it is associated to many use cases. In the case of actors that are specializations of more generic actors, only the most elementary specializations are counted, that is, actors that do not have subtypes.

Once the actors have been identified, their complexity is defined as follows:

- Human actors that interact with the system through a graphic user interface are considered complex and receive 3 use case points.
- Other systems that interact with the system through a protocol like TCP/IP and human actors that interact with the system only through a command line are considered of medium complexity and receive 2 use case points.
- Other systems that are accessed by application programming interfaces (API) are considered of low complexity and receive 1 use case point.

[7]http://www.ifpug.org/

The value of *UAW* (*unadjusted actor weight*) is simply the sum of the points assigned to all system actors.

In the example of Figure 3.11 the following actors were considered:

- Deposit manager: 3 points.
- Acquisition manager: 3 points.
- Sales manager: 3 points.
- Customer: 3 points.
- Credit card operator ≪system≫: 2 points.

Thus, for that example, $UAW = 3 + 3 + 3 + 3 + 2 = 14$.

The use of *UAW* for effort estimation, however, is not unanimously accepted. Sometimes, it is suggested that it can be ignored in the equations. Usually, even *UCP* counting tools consider *UAW* an optional measure and using it or not is a decision left to the team.

4.2.2 UUCW – unadjusted use case weight

Use cases are also counted once. Only complete processes should be counted. That means that extensions, variants, and other fragments, in general, should not be counted.

In the original proposal of Karner (1993), the complexity of a use case is defined based on the estimated number of *transactions* (information being sent to or received from the system), including alternate sequences (see Chapter 5 for more details). Then, use cases could be characterized as follows:

- *Simple use cases*: No more than three transactions.
- *Medium use cases*: Four to seven transactions.
- *Complex use cases*: More than seven transactions.

However, if we are working with high-level use cases, it may be hard to know for sure how many transactions each use case really has, and that makes that estimation a little risky.

An alternate way to estimate the complexity of a use case is by the number of classes necessary to implement the use case functions. Thus:

- Simple use cases must be implemented with five classes or less.
- Medium use cases must be implemented with six to ten classes.
- Complex use cases must be implemented with more than ten classes.

The number of classes necessary for implementing a use case will depend also on detailed requirements that usually are not known during Inception, given that the use cases have not been yet expanded. That makes this approach also somewhat risky.

Another way to estimate the complexity of a use case is by the analysis of its risk. Thus:

- *Reports* usually have only one or two transactions and low risk, because they do not change data. They can be considered simple use cases.
- *Pattern use cases* such as CRUD have a number of transactions that are known and limited. Their risk is medium, because although their internal logic is known, obscure business rules may affect them, and they can be considered medium complexity.

- *Nonpattern use cases* have an unknown number of transactions, and high risk, because besides the fact that the business rules may be unknown or misunderstood, even the internal structure of the use case is unknown (its main flow and alternate sequences). Thus, this kind of use case is considered complex.

The *UUCW* (*unadjusted use case weight*) value is given by the sum of the value assigned to each use case using the following criterion:

- Simple use case: 5 use case points.
- Medium use case: 10 use case points.
- Complex use case: 15 use case points.

Applying that technique to the example of Figure 3.11, there are eight use cases stereotyped as reports, which receive $8*5 = 40$ use case points. There are also three CRUD use cases (pattern use cases), which receive $3*10 = 30$ points. Finally, there are 10 use cases without any stereotype with higher risk, which receive $10*15 = 150$ use case points. Thus, the *UUCW* of this example is $40 + 30 + 150 = 220$.

There are also other counting techniques. But most of them are based on the detailed version of a use case. Some of them are presented in Section 4.2.9.

4.2.3 UUCP – unadjusted use case points

The value for *UUCP*, or *unadjusted use case points*, is calculated as the sum of *UAW* and *UUCW*:

$$\text{UUCP} = \text{UAW} + \text{UUCW}$$

In the running example, $UUCP = 14 + 220 = 234$.

As mentioned before, actors' weights may be excluded from the computation of *UUCP*, and that would make *UUCP* immediately equal to *UUCW*, that is, $UUCP = UUCW = 220$ in the example.

4.2.4 TCF – technical complexity factor

The technique of use case point analysis proposes that the *UUCP* should be adjusted by two criteria: *technical factors* (which are associated to the project), and *environmental factors* (which are associated to the team).

Each factor receives a rating from 0 to 5, where 0 means no influence on the project, 3 is nominal influence, and 5 is maximum influence on the project.

Surprisingly, this is the only information usually given in literature about the technical factors of use case point analysis, and that makes the assessment of these factors highly subjective, and even sometimes contradictory.[8]

[8]For example, regarding the T12 factor, there are authors that say that more third-party access means less effort, because there is code reuse, while other authors say that more third-party access means more effort, because it is necessary to understand code written by others. In this book, a synthesis between those two views is built: it is considered good to have access to well-written code, not so good to have access to code that needs revision, and worse to not have access to any third-party code. Reusing code of bad quality is considered completely undesirable (it is usually easier to develop it from scratch).

Table e4.3 Technical Factors for Adjusting use Case Points

Code	Factor	Weight
T1	Distributed system	2
T2	Response time/performance objectives	1
T3	End-user efficiency	1
T4	Internal processing complexity	1
T5	Design aiming for code reusability	1
T6	Easy to install	0.5
T7	Easy to operate	0.5
T8	Portability	2
T9	Design aiming for easy maintenance	1
T10	Concurrent/parallel processing	1
T11	Security	1
T12	Access for/to third-party code	1
T13	User training needs	1

There are 13 technical factors, each one with a specific weight, as shown in Table e4.3.

Therefore, each of the technical factors has a rate between 0 and 5 that is multiplied by its weight. The sum of those products corresponds to the *TFactor* value, that is, the impact of the technical factors, which ranges between 0 and 70.

The *TCF* (*technical complexity factor*) may be calculated by adjusting *TFactor* to the range 0.6 to 1.3 with the following formula:

$$TCF = 0.6 + (0.01 * TFactor)$$

For a better reference on the meaning of the ratings 0 to 5, it is suggested that one use the directions below, adapted from the function point analysis technique (Albrecht & Gaffney Jr., 1983).

T1 – Distributed system: Is the system architecture centralized or distributed?

0. The application ignores any aspect related to distributed processing.
1. The application generates data that will be processed by other computers with human intervention (for example, spreadsheets or preformatted files sent by media or email).
2. Application data are prepared and transferred automatically for processing in other computers.
3. Application processing is distributed, and data are transferred in just one direction.
4. Application processing is distributed and data are transferred in both directions.
5. Application processes must be executed in the most appropriate processing core or computer, which is dynamically determined.

T2 – Response time/performance objectives: What is the importance of the application response time to its users?

0. No special performance requirement was defined by the client.
1. Performance requirements were established and revised, but no special action must be taken.

2. Response time and transfer rates are critical during peak hours. No special design for processor core use is necessary. The deadline for most processes is the next day.
3. Response time and transfer rates are critical during commercial hours. No special design for processor core use is necessary. Requirements regarding deadlines for communication with interfaced systems are restrictive.
4. In addition to 3, performance requirements are sufficiently restrictive for requiring performance analysis tasks during design.
5. In addition to 4, performance analysis tools must be used during design, development, and/or implementation, in order to meet the client's performance requirements.

T3 – End-user efficiency: Is the application designed to allow final users just to do their job or is it designed to improve their efficiency?

0. The application does not need any of the items below.
1. The application needs one to three of the items below.
2. The application needs four to five of the items below.
3. The application needs six or more of the items below, but there is no requirement related to user efficiency.
4. The application needs six or more of the items below, and the user efficiency requirements are so strong that the design must include features to minimize typing, maximize defaults, use templates, etc.
5. The application needs six or more of the items below, and the user efficiency requirements are so strong that the design activities must include tools and special processes to demonstrate that the performance goals are obtained.

The following items must be considered for the assessment of the end-user efficiency item:[9]

- Navigational help (for example, dynamically generated menus, adaptive hypermedia, etc.).
- Online help and documentation.
- Automated cursor movement.
- Predefined function keys.
- Batch tasks submitted from online transactions.
- High use of colors and visual highlights in screens.
- Minimizing the number of screens to reach the business goals.
- Bilingual support (counts as four items).
- Multilingual support (counts as six items).

T4 – Internal processing complexity: Does the application need complex algorithms?

0. None of the options below.
1. One of the options below.
2. Two of the options below.

[9]Some items such as "menus" and "scrolling" that were originally considered in the function point analysis were removed from the list because they are considered too trivial for current applications.

3. Three of the options below.
4. Four of the options below.
5. All five options below.

The following options need to be considered for assessing internal processing complexity:

- Careful control (for example, special audit processing) and/or secure processing specific to the application.
- Extensive logical processing.
- Extensive mathematical processing.
- Lots of exception processing resulting from incomplete transactions that needs to be reprocessed, such as incomplete automated teller machine transactions caused by interruption of communication, missing data values, or failed data change.
- Complex processing to manage multiple inputs and output possibilities, such as multimedia or device independency.

T5 – Design aiming for code reusability: Is the application designed so that its code and artifacts will be highly reusable?

0. There is no concern about producing reusable code.
1. Reusable code is generated for use inside the same project.
2. Less than 10% of the application must consider more than the user needs.
3. 10% or more of the application must consider more than the user needs.
4. The application must be specifically packaged and/or documented for facilitating reuse, and the application must be customizable by the user at the level of source code.
5. The application must be specifically packaged and/or documented for facilitating reuse, and the application must be customizable by the user with the use of parameters.

T6 – Easy to install: Will the application be designed so that its installation is automatic (for example, in the case of users with low or unknown technical capacity), or is there no special concern about it?

0. The client established no special consideration, and no special setup is necessary for installation.
1. The client established no special consideration, but a special setup is required for installation.
2. The client established requirements for data conversion and installation, and conversion and installation guides must be provided and tested. The impact of conversion in the project is not considered important.
3. The client established requirements for data conversion and installation, and conversion and installation guides must be provided and tested. The impact of conversion on the project is considerable.
4. In addition to 2, tools for automatic conversion and installation must be provided and tested.
5. In addition to 3, tools for automatic conversion and installation must be provided and tested.

T7 – Easy to operate:[10] Are there special requirements regarding the operation of the system?

0. No special considerations about the operation of the system besides normal backup procedures were established by the user.
1. One of the items below applies to the system.
2. Two of the items below apply to the system.
3. Three of the items below apply to the system.
4. Four of the items below apply to the system.
5. The application is designed to operate in nonsupervised manner. "Nonsupervised" means that human intervention is not necessary for keeping the system operational, even if crashes occur, except maybe for the first startup and final turn off. One of the application features is automatic error recovery.

For the evaluation of the easy to operate factor, the following items must be considered:

- Effective processes for initialization, backup, and recovery must be provided, but operator intervention is still necessary.
- Effective processes for initialization, backup, and recovery must be provided, and no operator intervention is necessary (counts as two items).
- The application must minimize the need for data store in offline media (for example, tapes).
- The application must minimize the need for dealing with paper.

T8 – Portability: Is the application or parts of it designed to work on more than one platform?

0. There is no user requirement to consider the need for installing the application on more than one platform.
1. The design must consider the need for the system to operate in different platforms, but the application must be designed to operate only in *identical* hardware and software environments.
2. The design must consider the need for the system to operate in different platforms, but the application must be designed to operate only in *similar* hardware and software environments.
3. The design must consider the need for the system to operate in different platforms, but the application must be designed to operate in *heterogeneous* hardware and software environments.
4. In addition to 1 or 2, a documentation and maintenance plan must be elaborated and tested to support operation in multiple platforms.
5. In addition to 3, a documentation and maintenance plan must be elaborated and tested to support operation in multiple platforms.

T9 – Design aiming for easy maintenance: Does the client require that the application must be easy to change in future?

0. None of the items below.
1. One of the items below.
2. Two of the items below.
3. Three of the items below.

[10]Occasionally this item is referred to as "usability", but the original concept of usability was already considered in factor T3, end-user efficiency. We prefer to interpret this factor as "easy to operate" for its compatibility with the technical factors of function point analysis, and also to avoid confusion and redundancy with the end-user efficiency factor.

4. Four of the items below.
5. Five or more of the items below.

To evaluate this factor, the following items are considered:

- A flexible report structure must be provided to deal with simple queries such as logical binary operators applied to just one logical archive (count as one item).
- A flexible report structure must be provided to deal with medium complexity queries such as logical binary operators applied to more than one logical archive (count as two items).
- A flexible report structure must be provided to deal with high complexity queries such as combinations of logical binary operators applied to one or more logical archives (count as three items).
- Business control data are kept in tables managed by the user with interactive online access, but changes must only be effective on the next day (count as one item).
- Business control data are kept in tables managed by the user with interactive online access, and changes are effective immediately (count as two items).

T10 – Concurrent/parallel processing: Must the application be designed in order to deal with problems related to concurrency such as, for example, data and resource sharing?

0. No concurrent access to data is expected.
1. Concurrent access to data is expected sometimes.
2. Concurrent access to data is expected frequently.
3. Concurrent access to data is expected all the time.
4. In addition to 3, the user indicates that a lot of multiple accesses are going to happen, forcing performance analysis tasks and deadlock resolution during design.
5. In addition to 4, the design requires the use of special tools to control access.

T11 – Security: Are the security needs just nominal or are a special design and additional specifications required?

0. There are no special requirements regarding security.
1. The need for security must be taken into account in design.
2. In addition to 1, the application must be designed so that it can be accessed only by authorized users.
3. In addition to 2, access to the system will be controlled and audited.
4. In addition to 3, a security plan must be elaborated and tested to support access control to the application.
5. In addition to 4, a security plan must be elaborated and tested to support auditory.

T12 – Access for/to third-party code: Is the application going to use code already developed, such as commercial off-the-shelf (COTS) components, frameworks or libraries? High reuse of good quality software reduces the value of this item as it implies less development effort.

0. Highly reliable preexistent code will be used extensively for developing the application.
1. Highly reliable preexistent code will be used in small parts of the application.
2. Preexistent code that eventually needs to be adjusted will be used extensively for developing the application.
3. Preexistent code that eventually needs to be adjusted will be used in small parts of the application.
4. Preexistent code that needs to be fixed or is hard to understand will be used in the application.

5. No preexistent code will be used in the application or questionable quality code will be used in the application.

T13 – User training needs: Will the application be easy to use, or must extensive training be given to future users?

0. There are no specific requirements for user training.
1. Specific user training requirements have been mentioned.
2. There are formal specific user training requirements, and the application must be designed to facilitate training.
3. There are formal specific user training requirements, and the application must be designed to support users with different levels of training.
4. A detailed training plan must be elaborated for the transition phase and executed.
5. In addition to 4, the users are geographically distributed.

Applying the technical factors to the current example, a result similar to the one presented in Table e4.4 could be obtained.

The technical factors value, *TCF*, for the example is obtained by:

$$TCF = 0.6 + (0.01 * 22.5) = 0.825$$

As that value is less than 1, the technical factors for this project indicate a development time that is smaller than nominal (82.5% of nominal). However, the environmental factors still have to be determined.

4.2.5 EF – environmental factors

One aspect that distinguishes use case point analysis from function point analysis is that use case point analysis uses an adjustment factor for the characteristics of the development team. Thus, the same project, with the same technical factors, may have a different number of adjusted use case points for different teams.

There are eight environmental factors that assess the working environment. Each one has its own weight and two of them have negative weights. Again, ratings range from 0 to 5, but their meaning is a little different from those of the ratings attributed to the technical factors. While the ratings of the technical factors evaluate the *influence* of those factors on the project, the ratings of the environmental factors evaluate the *quality* of those factors in the working environment. For example, a 0 rating for the "motivation" factor means that the team is not motivated, a 3 rating means that motivation is average, and a 5 rating means that the team is highly motivated.

The environmental factors are defined in Table e4.5.

Thus, the ratings multiplied by the weight of the environmental factors may range from −10 to 32.5. This value is called the *EFactor*.

EFactor is adjusted to the interval 0.425 to 1.7, generating the *EF* (environmental factor) by the following formula:

$$EF = 1.4 - (0.03 * EFactor)$$

Ribu (2001) presents a reference for assigning ratings to the environmental factors, which is described below with some adaptation.

Table e4.4 Assessment for the Technical Factors of Livir

Code	Factor	Weight	Rate	Weight × Rate	Justification
T1	Distributed system	2	0	0	The application ignores any aspects related to distributed processing.
T2	Response time/ performance objectives	1	2	2	Response time and transfer rates are critical during peak hours. No special design for processor core use is necessary. The deadline for most processes is the next day.
T3	End-user efficiency	1	3	3	The application needs the following items, but there are no special requirements related to user efficiency: • Navigational help (for example, dynamically generated menus, adaptive hypermedia, etc.) • Online help and documentation • High use of colors and visual highlights in screens • Minimizing the number of screens to reach the business goals • Multilingual support (counts as six items)
T4	Internal processing complexity	1	0	0	None of the options apply.
T5	Design aiming for code reusability	1	0	0	There is no concern about producing reusable code.
T6	Easy to install	0.5	0	0	No special considerations were established by the client and no special setup is necessary for installation.
T7	Easy to operate	0.5	5	2.5	The application is designed to operate in nonsupervised manner. "Nonsupervised" means that human intervention is not necessary for keeping the system operational, even if crashes occur, except maybe for the first startup and final turn off. One of the application features is automatic error recovery.
T8	Portability	2	3	6	The design must consider the need for the system to operate in different platforms, but the client application must be designed to operate in heterogeneous hardware and software environments.
T9	Design aiming for easy maintenance	1	0	0	None of the items apply.

(*Continued*)

Table e4.4 (Continued)

Code	Factor	Weight	Rate	Weight × Rate	Justification
T10	Concurrent/ parallel processing	1	1	1	Concurrent access to data is expected sometimes.
T11	Security	1	3	3	The need for security must be taken into account in design. The application must be designed so that it can be accessed only by authorized users. The access to the system will be controlled and audited.
T12	Access for/to third-party code	1	5	5	No preexistent code will be used in the application.
T13	User training needs	1	0	0	There are no specific requirements for user training.
TFactor				22.5	

Table e4.5 Environmental Factors for Adjusting Use Case Points

Code	Factor	Weight
E1	Familiarity with the development process	1.5
E2	Experience in the application	0.5
E3	Object-oriented experience	1
E4	Lead analyst experience	0.5
E5	Motivation	1
E6	Stable requirements	2
E7	Part-time workers	−1
E8	Difficulty with programming language	−1

E1 – Familiarity with the development process: This factor assesses the team's experience with the development process they are using. Originally the factor mentioned the Unified Process, and some authors even mention UML here. But this has been adjusted to reflect the fact that other processes and modeling languages could be used instead.

0. The team does not have experience with the development process or does not use any process.
1. The team has theoretical knowledge about the development process, but no experience.
2. A few members of the team have already used the process in one project.
3. A few members of the team have used the process in more than one project.
4. Up to half the team has used the process in many projects.
5. More than half the team has used the process in many projects.

E2 – Experience in the application: This factor assesses the familiarity of the team with the application area or domain. For example, if the application is about e-commerce, have the team already worked with systems in the same area?

0. No team member has any experience in projects in the same area.
1. Some team members have 6 to 12 months of experience in projects in the same area.
2. Some team members have 12 to 18 months of experience in projects in the same area.
3. Most team members have 18 to 24 months of experience in projects in the same area.
4. Most team members have greater than two years of experience in the same area.
5. All team members have greater than two years of experience in the same area.

E3 – Object-oriented experience: This factor must not be confused with either *familiarity with the development process* (E1) or *experience in the application* (E2): this is about the experience of the team in doing object-oriented analysis, modeling, design, and programming, independently of the application area and the development process used.

0. The team has no experience in object-oriented techniques.
1. All team members have some experience in object-oriented techniques (up to one year).
2. Most team members have 12 to 18 months of experience in object-oriented techniques.
3. All team members have 12 to 18 months of experience, or most team members have 18 to 24 months of experience in object-oriented techniques.
4. Most team members have more than two years of experience in object-oriented techniques.
5. All team members have more than two years of experience in object-oriented techniques.

E4 – Lead analyst experience: This factor measures the experience of the lead analyst with requirement analysis and object-oriented modeling.

0. The lead analyst has no experience.
1. The lead analyst has previous experience in a single similar project.
2. The lead analyst has about one year of experience in more than one similar project.
3. The lead analyst has about two years of experience in similar projects.
4. The lead analyst has more than two years of experience in similar projects.
5. The lead analyst has more than three years of experience in a variety of projects.

E5 – Motivation: This factor describes the team's motivation.[11]

0. The team has no motivation at all. Without constant supervision the team becomes unproductive. The team only does what is strictly asked.
1. The team has very little motivation. Constant supervision is necessary to keep productivity at acceptable levels.
2. The team has little motivation. Management interventions are necessary from time to time to maintain productivity.

[11]Surely that is one of the hardest environmental factors to assess, because the real motivation of the team may be masked. Ribu's (2001) suggestion seemed to help very little in rating this factor, and we took the liberty of reinterpreting the reference.

3. The team has some motivation. Usually the team has initiative, but management intervention is still necessary sporadically to keep productivity.
4. The team is well motivated. The team is self-managed usually, but the existence of supervision is still necessary because productivity can be lost without it.
5. The team is highly motivated. Even without supervision everyone knows what has to be done, and pace is maintained indefinitely.

E6 – Stable requirements: This factor evaluates if the team has been able to keep requirements stable in past projects, minimizing their change during the project. This factor may vary from project to project, because there are domains where requirements change often independently of the capacity of the analysts. This factor must assess only software changes that were caused by the team's inability to discover the right requirements.

0. There is no historic data about requirement stability or, in the past, poor analysis caused big changes in requirements after the project started.
1. Requirements were predominantly unstable in the past. Clients asked for many changes caused mainly by incomplete or incorrect requirements.
2. Requirements were unstable in the past. Clients asked for some changes caused by incomplete or incorrect requirements.
3. Requirements were relatively stable in the past. Clients asked for changes in secondary functionalities with some regularity. Changes to main functionalities were asked for seldom.
4. Requirements were mostly stable in the past. Users asked for little changes, especially cosmetic ones. Changes in main or secondary functionalities were unusual.
5. Requirements were totally stable in the past. Little changes, if any, had no impact on projects.

E7 – Part-time workers: This is a negative factor, that is, contrary to the first six environmental factors, higher values here represent more development time, not less. A team with many members involved in other projects or activities is less productive than a dedicated team.

0. No team member is part time on the project.
1. Up to 10% of the team is part time.
2. Up to 20% of the team is part time.
3. Up to 40% of the team is part time.
4. Up to 60% of the team is part time.
5. More than 60% of the team is part time.

E8 – Difficulty with programming language: This is another negative factor, meaning that high values here are bad for the development time.

0. All team members are very experienced programmers.
1. Most team members have at least two years of experience in programming.
2. All team members have at least 18 months of experience in programming.
3. Most team members have at least 18 months of experience in programming.
4. A few team members have some experience in programming (not more than one year).
5. All team members are inexperienced programmers.

In order to continue with the Livir example, let us assume that the company is formed by a group of five professionals who recently obtained Computer Science degrees and studied object-oriented

Table e4.6 Environmental Factors Evaluation for a Fictional Team Developing the Livir Project

Code	Factor	Weight	Rate	Weight × Rate	Justification
E1	Familiarity with development process	1.5	1	1.5	The team has theoretical knowledge about the development process but no commercial practice.
E2	Experience in the application	0.5	0	0	No team member has previous experience with similar projects.
E3	Object-oriented experience	1	3	3	All team members have about 12 to 18 months of (academic) experience with object-oriented techniques.
E4	Lead analyst experience	0.5	0	0	The lead analyst has no experience.
E5	Motivation	1	5	5	The team is highly motivated. Even without supervision, each member knows his/her job and maintains the pace indefinitely.
E6	Stable requirements	2	0	0	There is no trustable history for the team.
E7	Part-time workers	−1	3	−3	40% of the team is part time.
E8	Difficulty with programming language	−1	5	−5	Although they have academic experience, all programmers are novices.
EFactor				1.5	

techniques, but do not have significant professional experience. Three of them are fully dedicated to the project, while the other two have other activities. Livir will be the first project they are going to perform professionally together. With this fictional company, the evaluation of the environmental factors could be something like Table e4.6.

Applying the adjustment formula to the company's environmental factors the following is obtained:

$$EF = 1.4 - (0.03 * 1.5) = 1.355$$

This indicates that the environmental factors are bad for this team of beginners, and that their effort will be around 35% greater than nominal.

4.2.6 UCP – adjusted use case points

Once actor and use case weights as well as technical and environmental factors are calculated, the adjusted use case points may be obtained by simple multiplication:

$$UCP = UUCP * TCF * EF$$

For the running example we obtain:

$$UCP = 234 * 0.825 * 1.355 = 261.583$$

4.2.7 Effort

The key goal of all estimation techniques is to predict the *effort* necessary to develop a project. In the first place, it is necessary to define the extension of the effort that is being estimated. The COCOMO II method, for example, when applied with the Unified Process, estimates the effort to be spent during Elaboration and Construction only.

Besides this, does the effort include administrative activities, sales, etc.? Or does it include only the effort spent on software engineering activities? Literature on use case points usually is not as detailed as COCOMO II and function point analysis regarding the kind of effort that is being estimated. As use case point analysis is based on function point analysis Mk II (UKSMA Metrics Practices Committee, 1998), we can assume that by default the same definitions apply, such as "The project effort should include staff who is clearly allocated to the project and who work according to the project requirements. The time of a user being interviewed, for instance, would not be included, whereas the time of a user allocated for a significant amount of time to the project team would be included." Similarly, the following definition can also be applied: "The work effort should be measured in units of a 'productive' or 'net' work-hours, which is defined as: 'One hour of work by one person, including normal personal breaks, but excluding major breaks, such as for lunch'." Mk II also states that a project starts when "it has been agreed and approved that work should be undertaken to deliver an IS[12] solution [. . .] one or more staff have been allocated to the project, and start working on it" (this happens during the Inception phase), and that a project ends when "the application is made 'live' by moving it into the production environment for the first time, and the users make use of it" (this happens during the Transition phase).

The effort that is calculated usually must include all risks and unexpected situations. Estimation is not a precise science though. In some situations things may get out of control and adjustments must be made on the fly to keep the project on the rail. We can say that the final schedule would define the *maximum time tolerable* for delivery of a product with the *minimum scope acceptable*. That is, if we are lucky and risks cause little trouble, we can deliver the product earlier or deliver a better product than expected.

Once the number of use case points is calculated, they have to be transformed into an effort measure by using a *team productivity index* (*TPI*). If the TPI is measured as the time one team member spent on a single use case point[13] then the effort may be given by:

$$E = UCP * TPI$$

As *UCP* is a scale-independent value, it must be decided in which scale the effort will be measured. The use case point literature usually refers to *staff hours* per use case point. However, other methods such as COCOMO II (Boehm, 2000) prefer *staff months*.

Because most formulas that are usually applied to productivity in the software industry are defined in terms of *staff months*, and because the unit chosen may affect results, this book adopts staff months as the productivity unit, but refers to staff hours when necessary.

[12]Information System.

[13]Optionally, the *TPI* could be expressed as the number of use case points a team member can produce in a given unit of time. In that case, the formula should be $E = UCP / TPI$.

Considering that a month corresponds to 158 working hours on average (holidays and weekends disregarded), then x staff hours per use case point correspond to $x/158$ staff months per use case point.

Originally, Karner (1993) estimated than 20 staff hours (approximately 0.127 staff months) per use case point[14] would be a good guess when a trustable history is lacking. Later, Ribu (2001) determined that that value could range from 15 to 30 staff hours (0.095 to 0.190 staff months) per use case point.

The *TPI* value can also be estimated by taking past projects, looking for the total staff time spent and dividing it by the number of use case points. Thus, for example, if the total effort spent on a given project was 9 staff months and the total project UCP was 60, then the team productivity index in that case is 9/60, that is, 0.15 staff months per use case point.

Other work (Schneider & Winters, 1998) proposes that the number of environmental factors E1 to E6 rated below 3 must be added to the number of environmental factors E7 and E8 rated above 3, and:

- If the result is 2 or less, assume 20 staff hours (0.127 staff months) per use case point.
- If the result is 3 or 4, assume 28 staff hours (0.177 staff months) per use case point.
- If the result is greater than 4, environmental factors present too high a risk for the project, and must be improved before any development commitment is assumed, or else assume 36 staff hours (0.228 staff months) per use case point.

In the current example there are five negative environmental factors: E1, E2, E4, E6, and E8. The recommendation, in this case, would be trying to improve the quality of the environment before developing any project. However, if the team decides to take the risk, the project should assume the worst estimate of 0.228 staff months per use case point. Thus, the total effort for the Livir project is $E = 261.583*0.228 = 59.64$ staff months.

4.2.8 Calendar time and average team size

The effort value obtained for the example in the last section (59.64 staff months) refers to the total effort for developing the project from its start until the delivery of the product. However, more than one staff member will be working on the project and thus calendar time (linear time with weekends and holydays disregarded) will probably be much smaller.

There is a simple rule of thumb formula that allows one to approximate the ideal calendar time for a project from its total effort (McConnel, 1996). The formula is:

$$T = 2.5 * \sqrt[3]{E}$$

In this formula, T is the ideal calendar time for developing the project.

This formula only works if E is expressed in *staff months*. The use of other units such as staff weeks, or staff hours, would cause distortions in the result due to the use of the cubic radix.

Other formulae based on function points (Jones, 2007) or lines of code (Boehm, 2000) are available. But, in order to be applied with use case points, some conversions are needed. If a more

[14]Remember that a complex use case has 15 use case points and thus requires about 300 staff hours to be fully developed according to Karner (1993).

accurate approximation is desired maybe this is worth the effort because Boehm's and Jones's formulae use information about the project complexity and team capacity as parameters to produce the calendar time estimate.

Applying the rule of thumb formula above to the current example produces:

$$T = 2.5 * \sqrt[3]{59.64} \cong 9.768 \text{ months}$$

The average size of the team (P) is calculated by:

$$P = E/T$$

For the running example, we obtain:

$$P = 59.64/9.768 = 6.106$$

Thus, the average size of the team, in that case, must be approximately equal to six. In summary, the project would be ideally developed by six people in about 10 months. For this example, the development company could choose to hire a new professional (especially a more experienced professional that could help to improve the environmental factor ratings); trying to develop the project with five people could take a time longer than the ideal. For this example, it is possible to calculate the time a team of five would take for developing the project as $59.64/5 = 11.928$. In that case, the project would be completed by five people in about 12 months. We said "it is possible" because the number of people considered is smaller than the recommended size for the team. However, it would be unrealistic to apply this division to calculate the calendar time for a team larger than the ideal. For instance, it would not be realistic to assume that 20 people could complete the project in $59.64/20 = 2.982$ months. Any team size higher than the ideal means that the total effort (E) must be adjusted to a higher value. The COCOMO II effort multiplier *SCED* (*Required Development Schedule*) indicates that running a project on a time that is 85% of the ideal time implies adding 14% to the total effort, and running a project on a time that is 75% of the ideal time implies adding 43% to the total effort (Boehm, 2000).

It may appear that those times are too high for the project presented in the example. However, it must be considered that the environmental factors are very bad, because the team has very little experience. They will make errors and possibly will have to redo a lot of work. Most important, the lack of experience may lead the team to misunderstand requirements and have to fix them later, increasing development time significantly.

If instead of those novices we had a *dream team* with the best rates possible for environmental factors, then EF would be 0.425 (the best value possible). In that case, $UCP = 234*0.825*0.425 = 82.046$. Using the recommendation of Schneider and Winters (1998), it can be assumed that the dream team could work at a rate of 20 staff hours or 0.127 staff months per use case point. Thus, $E = 82.046*0.127 = 10.42$. Applying the formula for calendar time and average team size the result is $T = 5.46$ and $P = 1.908$. That is, a dream team of two people could complete the project in about five and a half months. That seems a very reasonable estimate considering the size of the project and the capacity of the team.

However, it is important to mention that those formulas are somewhat "magical" because there are a great range of projects and development teams. Before using them seriously in a company it is always useful to test and adjust them to the local reality.

4.2.9 Counting methods for detailed use cases

The use case point analysis technique presented in previous subsections considers only a guess on the complexity of the use cases because it deals with high-level use cases, which are represented only by their name and, eventually, one or two explaining sentences.

There are techniques for counting use case points with expanded use cases (explained in Chapter 5), but their utility is more limited because, with the Unified Process, use cases are usually only detailed during the iterations. During Inception, the team usually only have high-level use cases.

Even so, those techniques may be useful if the use cases are already detailed, or if effort estimation must be refined when an iteration starts. This is why those techniques are summarized in this section.

Robiolo and Orosco (2008) suggest that the text of the detailed use case must be analyzed and the entities (classes) and transactions (steps) counted. The size of an application in unadjusted use case points would be the sum of those values.

Braz and Vergilio (2006) propose a technique named FUSP (Fuzzy Use Case Size Points), where counting is based on the complexity of:

- Actors that participate in the use case.
- Use case pre- and postconditions.
- Number of entities necessary for the use case.
- Number of transactions of the use case.
- Complexity of the alternate flows of the use case.

Thus, the measure is quite detailed, but a lot of work is necessary to apply it and the use cases must be expanded for it to work.

Mohagheghi, Anda, and Conradi (2005) propose a technique (*Adapted Use Case Points*) that assumes that every actor has medium complexity, and every use case is complex in the beginning. Later, use cases must be split into smaller use cases and reclassified as simple or medium as the refining process goes on. Extended use cases and alternate flows are counted as well.

Kamal and Ahmed (2011) present an extensive comparison among the aforementioned and other counting methods.

The *Full Use Case Size* method (Ibarra & Vilain, 2010) was proposed with the objective of measuring the size of use cases. The method measures functional and nonfunctional aspects and, to do that, it needs high-level use cases and a list of nonfunctional requirements annotated for each use case. Measurement is based on three components:

- The kind of use case.
- The number of system operations (transactions) and business rules.
- The number of interface requirements.

Details about these techniques may be found in the original works cited above.

4.3 Planning an iterative project

The goal of project planning is to build a *plan* for the project as a whole. Among other things, it is important that the person or group responsible for planning use the best tools available to assess the amount of effort that should be expended in the project. That estimate will lead to the project schedule and budget.

There are several aspects to consider at the start. The scope declaration has already defined the goals of the project and acceptance criteria. Also, the necessary business modeling has already been completed, and the high-level requirements were incorporated to the system use cases.

With the technique of use case point analysis, the total effort for developing the project may be estimated, as well as the ideal calendar time and the average size of the team; these are global measures for the project.

Now it is necessary to define the duration and number of iterations.

4.3.1 Estimating the duration of iterations

An iteration begins with planning and finishes with a new version of the system being released internally or delivered to the client. The duration of an iteration in UP and most agile methods usually ranges from one to eight weeks,[15] and it depends, among other factors, on the complexity of the project and the size of the team.

Small teams with up to five people[16] may do the planning together on a Monday morning, perform the work during the week, and generate a release on Friday.

Groups with more than 20 people might need more time to distribute and synchronize activities. Consider that the amount of work is naturally higher. Moreover, the generation of a release takes more time, because there will be a higher amount of software parts to be integrated, verified, and fixed. Thus, in that case, iterations with three to four weeks would be allowed.

Groups with more than 40 people need to work on a more structured and formal environment, with more intermediary documentation, so that the information flow will naturally be slower. Thus, an iteration with six to eight weeks would be advisable.

Other factors that may affect the duration of the iterations are the following:

- Automatic code generation and integrated life cycle software tools allow for shorter iterations.
- A team that is very competent in UP and analysis, design, and coding techniques allows for shorter iterations.
- If quality is a critical concern and consequently tests and revisions are intense, then iterations must be longer, unless extensive automation and effective quality coding patterns are used.

4.3.2 Number of iterations

The number of iterations of a project depends on the calendar time divided by the duration of each iteration. For example, for the Livir project, with the novice team, the project would take about 10 months with a team of six people. Thus, an iteration of two weeks would be used, with a total of 20 iterations.

The number of iterations that would be considered for Elaboration and Construction depends fundamentally on the need to address architectural problems. This can be roughly estimated by the proportional quantity of complex use cases that should be studied and developed. As the milestone

[15]Agile methods prefer the shortest duration possible. Some methods such as XP are more radical by forbidding iterations with more than three weeks, but other methods are more relaxed on that.
[16]Or larger teams that are very tuned and have excellent tool support.

of the Elaboration phase is a stable architecture, it can be assumed that Elaboration will not terminate while there are complex high priority use cases that need to be analyzed. Kruchten (2003) estimates that the Elaboration would take about 30% of the linear time of a typical project and Construction would take about 50% of the time of a typical project, while Inception and Transition would take about 10% each.

Another question that comes to mind at this point is whether the effort estimation includes the Inception phase or not (because Inception is practically finished when the estimate is made). The COCOMO II method, for example, when applied to the Unified Process, estimates only the effort to be spent in Elaboration and Construction, leaving Inception and Transition to be calculated as a percentage of the total effort. Usually, it is believed that use case point analysis estimates the total effort of a project, from its beginning to the delivery of the product. However, if the team uses another interpretation, the important thing is to keep it coherent from one project to another. Any discrepancy in estimation measures would be absorbed by the team productivity index.

4.3.3 Effort per use case point

In order to find the effort to develop each kind of use case (simple, medium, or complex), the following procedure may be used:

1. Take the total effort for the project (E), and divide it by the total unadjusted use case weight ($UUCW$) − ignore the actors' weight.
2. The resulting value is the effort estimated per unadjusted use case point for the specific project and team ($E_{UCP} = E/UUCW$).
3. Finally, find the effort for each kind of use case by multiplying E_{UCP} by 5 in the case of simple use cases, by 10 for medium, and by 15 for complex ones.

For the running example we have:

$$E_{UCP} = \frac{59,64}{220} = 0.271 \text{ staff months per use case point}$$

Thus:

- Simple use cases demand 5*0.271 = 1.355 staff months.
- Medium use cases demand 10*0.271 = 2.71 staff months.
- Complex use cases demand 15*0.271 = 4.065 staff months.

Notice that the dream team would produce values much lower than those. In the case of the dream team:

$$E_{UCP} = \frac{10,42}{220} = 0.047 \text{ staff months per use case point}$$

Thus:

- Simple use cases demand 5*0.047 = 0.235 staff months.
- Medium use cases demand 10*0.047 = 0.47 staff months.
- Complex use cases demand 15*0.047 = 0.705 staff months.

4.3.4 Team load capacity

Other information that can be calculated is the *team load capacity (TLC)* for each iteration. This is calculated as the number of team members multiplied by the iteration length (expressed in months):

$$TLC = P * \text{iteration length (in months)}$$

In the running example, considering the novice team (six people) and two week iterations (about 0.5 months), $TLC = 6^{*}0.5 = 3$. That means that each iteration for that project has a team load of three staff months.

With a two-week iteration, the team would not be able to finish a complex use case (it demands 4.065 staff months, as seen above). Two options may be considered:

- Extend the duration of the iterations to three weeks (about 0.75 months).
- Slice use cases so that they may be partially implemented in two-week iterations.

Although agile developers would prefer shorter iterations, if we choose to extend the iterations, the calendar time for the project (10 months) divided by the iteration duration (0.75) results in 13.3 iterations, which can be adjusted to 14, unless the project is to be hurried. This way, the team load per iteration would be $6^{*}0.75 = 4.5$ and one iteration would be sufficient to deal with a complete complex use case.

4.3.5 Defining use case priority

One of the main techniques for reducing risk in a project is dealing first with the most complex use cases, especially if they are those that represent the most critical business processes, because with them the team may learn more about the system than with other use cases. Pattern use cases such as CRUD (medium complexity) and reports (smallest complexity) can be addressed later during Construction.

Planning indicates when each use case is going to be analyzed, designed, and implemented. The Unified Process, like most agile techniques, does not expect that the general project plan defines when each use case is to be implemented. However, they are placed in a dynamic priority list, and although it may be updated on the fly, it is possible to know when each use case is supposed to be completed. Planning an iteration begins by choosing which use cases, risks, or change requirements would be addressed on the next iteration. When planning an iteration, the following aspects must be considered when choosing which use cases to address:

- *Use case complexity*: Use cases of higher risk and complexity should be addressed first in most cases, unless special conditions (for example, a relatively simple use case with high technological risk) justify choosing different use cases.
- *Dependencies among use cases*: Use cases that have a strong dependence to the ones that were already accommodated in the iteration should be addressed together if possible, because it avoids fragmenting the work on the classes.
- *Team load capacity*: The development work must be assigned to team members based on their *TLC*.

Table e4.7 Suggestion for Use Case Priorities for the Livir Example

1.	Order books	12.	Manage books ≪crud≫
2.	Pay order	13.	Manage publishers ≪crud≫
3.	Deliver order	14.	Order status ≪report≫
4.	Register delivery confirmation	15.	Past sales ≪report≫
5.	Receive books	16.	Book sales by period ≪report≫
6.	Resend order	17.	Books available for sale ≪report≫
7.	Register order return	18.	Book returns by period ≪report≫
8.	Discard books	19.	Upcoming orders by period ≪report≫
9.	Cancel order	20.	Deliveries by period ≪report≫
10.	Create/remove special offer	21.	Discarded books by period ≪report≫
11.	Manage customers ≪crud≫		

One suggestion for prioritizing use cases for the Livir example presented in Figure 3.11 is given in Table e4.7.

The order of this list is most critical for the complex use cases, because a wrong choice in terms of their priorities could lead to otherwise unnecessary refactoring in subsequent iterations. We considered that the most important use cases for this business are those related to the process of selling books, because that is what generates profit for the company. However, the order of use cases with lower risk such as *cruds* and reports is not so critical.

4.3.6 Planning phase and iterations

With the information obtained in previous sections, it is possible to elaborate the preliminary release plan utilizing the use case priorities, effort, and team load capacity.

The goals of an iteration may be of three types:

- Implementing totally or partially one or more *use cases*.
- Mitigating one or more *risks*, generating or performing a probability reduction, impact reduction, or disaster recovery plan.
- Implementing totally or partially one or more *changes* requested. As the system architecture evolves during iterations, changes may be asked for due to nonconformities to requirements or design errors. Incorporating these changes into the software may be one of the goals of an interaction. This kind of goal usually only arises after the first iteration.

For each element (use case, risk, or change requested) there must be effort estimation. The elements will be selected for inclusion in one iteration or another depending on their priority (priorities may change as the project progresses – therefore, planning may change too). Higher priority must be given to the more complex and risky elements, which are the ones with greater potential for learning about the system. The suggestion is to pick first:

- Use cases that represent the most critical business processes, for example, those which help the company achieve its organizational goals, such as obtaining profit.
- High-importance (or high-exposure) risks, that is, those with high impact and a high probability of becoming problems.
- Urgent change requests, for example those that demand architecture refactoring.

When considering the elements of highest priority, other elements with lower priorities, but closely related to the higher priority ones, may be added to the same iteration for convenience. The important thing is that the total effort assigned to the iteration does not surpass the staff month value that corresponds to the team load capacity.

Risk mitigation and some change requests may not be suitable for effort estimation because their size cannot be measured in terms of use case points (or any other metric). For example, finding and fixing a bug may take seconds or weeks. In these cases, usually a fixed amount of time is assigned, such as, for example, investing one team member during one week to work on the mitigation of a given risk. At the end of the time box[17] previously defined the result is evaluated and it is decided if the activity must continue or not.

Let's consider that the project has 10 months (40 weeks) and the iteration duration is three weeks. Then, Inception and Transition would take about four weeks (10% of the total time), Elaboration would take about 12 weeks (30% of the total time) and four three-week iterations, and Construction would take about 20 weeks (50% of the total time) and close to seven iterations (one of them could be a week shorter, or one week from the Transition phase could be borrowed for the last Construction iteration – the chosen alternative presented below).

Kroll (2004) presents a suggestion for a generic phase plan that is adapted for the Livir example in Table e4.8. At the moment change requests were not included because they appear only when the project is running, not at the beginning. Thus, the plan in this case was made only based on use case release and risk mitigation.

After phase planning concludes (usually at the end of Inception), the plan for the first Elaboration iteration *E*1 can be detailed, with use cases being chosen and activities detailed and assigned to developers. Only when that iteration is running should the planning for the second iteration begin.

The purpose of an iteration plan includes the following (Kroll 2004):

1. Define the scope of the iteration by defining its goals in terms of artifacts to be produced such as:
   - Detailing and implementing specific use cases.
   - Mitigation of known risks.
   - Performing requested changes.
2. Create a detailed plan on how the iteration must unfold.
3. Create, identify, and manage tasks dependencies.
4. Assign tasks to people.

[17]A *time box* is a fixed period of time allocated for a task.

Table e4.8 Generic Phase Planning for the Livir Example

Phase	Iteration	Primary Objective (Risks and Use Cases Addressed)	Schedule
Inception	I1	Define vision Determine project scope Define candidate architecture Create business case Create software development plan	Weeks 1 to 4
Elaboration	E1	Install and test architecture components Validate requirements details Implement priority use cases. Test proposed architecture	Weeks 5 to 7
	E2-E4	Mitigate architectural risks. Complete architecture installation and test Implement additional use cases Load test architectural elements	Weeks 8 to 16
Construction	C1-C6	Describe and implement additional use cases Integrate product and validate	Weeks 17 to 34
	C7	Describe and implement additional use cases Integrate product and validate Plan beta test Plan user manual	Weeks 34 to 37
Transition	T1	Deliver beta to client Analyze feedback from beta users Apply corrections Finalize user manual and other artifacts Deliver to client	Weeks 38 to 40

For example, consider that you are planning the first Elaboration iteration *E*1 for the Livir project. Consider the general goals presented in Table e4.8, and the current state of the system. Some iteration goals could be defined such as:

- Expand and implement use case 01: *Order books*.
- Test and refine the preliminary architecture.
- Perform mitigation plans for risk *K*1: *Unstable requirements*.

Then, planning may continue by defining the iteration deliverables and the tasks necessary to produce them, and assigning them to developers. Agile teams might prefer to produce this plan in a team meeting, while teams following RUP may prefer to instantiate some of the RUP workflows in order to determine which tasks should be performed in order to achieve the iteration goals. Explaining this is out of the scope of this book, but readers interested in learning more may start by consulting West (2002) or Crain (2004).

4.4 The process so far

	Inception	Elaboration
Business Modeling	Build a general view of the system: • Build a business use case diagram and determine the automation scope for the project. • Build preliminary activity diagrams for business use cases. • Build preliminary state machine diagrams for key business objects.	
Requirements	Prepare the system use case diagram (functional requirements): • Identify the system actors from the business use case model. • Identify the system use cases from the business use case model, and activity and state machine diagrams from business modeling. Identify nonfunctional requirements as use case annotations: • Identify the main business rules associated to use cases. • Identify the main quality issues associated to use cases. Identify supplementary requirements.	
Analysis and Design	Prepare the preliminary conceptual model by observing system use cases and the concepts needed by them.	
Implementation		
Test		
Project Management	**Estimate the total effort, ideal calendar time, and average team size for the project**. **Estimate the duration and quantity of iterations for each phase**. **Prepare the phase plan and iteration plan for the first iteration**.	

4.5 Questions

1. Explain the differences among COCOMO II, function point analysis, and use case point analysis. What are their relative advantages and disadvantages?
2. In use case point analysis what is the difference between technical and environmental factors? Why does use case point analysis consider them as separate groups?
3. Let *E* be the total effort for developing a project, and *T* the ideal calendar time recommended for the same project. What happens with *E* when the project must be developed in a time shorter than *T*? Does the same occur when the project may be developed in a time longer than *T*?
4. How do you determine the ideal duration of an iteration for a given project?
5. How do you establish a list of priorities for use cases?