Chapter 6

Use Case Points

If you've worked with use cases before you've probably felt there should be an easy way to estimate the overall size of a project based on the use cases. There's clearly a relationship between use cases and code in that complicated use cases generally take longer to code than simple use cases. Fortunately, there is an approach for estimating and planning with *use case points*, which represent a measure of the effort needed to develop a set of use cases. In this chapter, we will look at both use case points and at a second approach to estimating and planning with use cases.

Use Case Points

The number of use case points in a project is a function of the following:

- the number and complexity of the use cases in the system
- the number and complexity of the actors on the system
- various non-functional requirements (such as portability, performance, maintainability) that are not written as use cases
- the environment in which the project will be developed (such as the language, the team's motivation, and so on)

The basic formula for converting all of this into a single measure, use case points, is that we will "weigh" the complexity of the use cases and actors and then adjust their combined weight to reflect the influence of the non-functional and environmental factors.

Fundamental to the use of use case points is the need for all use cases to be written at approximiately the same level. Alistair Cockburn identifies five levels for use cases: very high summary, summary, user goal, subfunction, and too low.[1] Cockburn's two types of summary use cases are useful for setting the context within which lower-level use cases operate. However, they are written at too high of a level to be useful for estimating. Cockburn's subfunction use cases are written to provide detail, on an as-needed basis, for the user goal-level use cases that should form the foundation of a well thought out collection of use cases. A sample goal-level use case is shown in Figure 6.1.

Since this is not a book on use cases we won't fully cover all the details of the use case shown in Figure 6.1; however, it is worth reviewing the meaning of the Main Success Scenario and Extensions sections. The Main Success Scenario is a description of the primary successful path through the use case. In this case, success is achieved after completing the five steps shown. The Extensions section defines alternative paths through the use case. Often, extensions are used for error handling; but, extensions are also used to describe successful but secondary paths, such as in extension 3a of Figure 6.1. Each path through a use case is referred to as a scenario. So, just as the Main Success Scenario represents the sequence of steps one through five, an alternate scenario is represented by the sequence 1, 2, 2a, 2a1, 2, 3, 4, 5.

Unadjusted Use Case Weight

If all of a project's use cases are written at Cockburn's user goal level then it is possible to calculate use case points from them. Unlike estimating with story points where the team discusses each story and subjectively assigns a point value, use case points are assigned by a formula. In Karner's original description of use case points, each use case is assigned a number of points based on the number of transactions within the use case.[2] At the user-goal level, the number of transactions can be determined by counting the number of steps in the use case. Karner originally proposed ignoring transactions in the extensions part of a use case. However, this was probably largely because extensions were not as commonly used in the use cases he worked with during the era (1993) he first proposed use case points. Clearly, extensions represent a significant amount of work and need to be included in any reasonable estimating effort.

Counting the number of transactions in a use case with extensions requires a small amount of caution. That is, you cannot simply count the number of lines in the extension part of the template and add those to the lines in the main success scenario.

1. See Cockburn, *Writing Effective Use Cases*, pp. 61–69.

2. See Karner, "Use Case Points—Resource Estimation for Objectory Projects."

Use Case Title: Pay for a job posting
Primary Actor: Recruiter
Level: Actor goal
Precondition: The job information has been entered but is not viewable.
Minimal Guarantees: None
Success Guarantees: Job is posted; recruiter's credit card is charged.
Main Success Scenario:

1. Recruiter submits credit card number, date, and authentication information.
2. System validates credit card.
3. System charges credit card full amount.
4. Job posting is made viewable to Job Seekers.
5. Recruiter is given a unique confirmation number.

Extensions:

2a: The card is not of a type accepted by the system:
 2a1: The system notifies the user to use a different card.
2b: The card is expired:
 2b1: The system notifies the user to use a different card.
2c: The card number is invalid:
 2c1: The system notifies the user to re-enter the number.
3a: The card has insufficient available credit to post the ad.
 3a1: The system charges as much as it can to the current credit card.
 3a2: The user is told about the problem and asked to enter a second credit card for the remaining charge. The use case continues at Step 2.

Figure 6.1 A sample use case for pay for a job posting.

In Figure 6.1, each extension starts with a result of a transaction, rather than a new transaction itself. For example, extension 2a ("The card is not of a type accepted by the system") is the result of the transaction described by step 2 of the main success scenario ("System validates credit card"). So, item 2a in the extensions section of Figure 6.1 is not counted. The same, of course, is true for 2b, 2c, and 3a. The transaction count for the use case in Figure 6.1 is then ten. You may want to count 2b1 and 2b2 only once but that is more effort than is worthwhile and they may be separate transactions sharing common text in the use case.

Table 6.1 shows the points assigned to each simple, average, and complex use case based on the number of transactions. Since the use case we're considering contains more than seven transactions it is considered complex.

Table 6.1 Use case weights based on the number of transactions.

Use case complexity	**Number of transactions**	**Weight**
Simple	3 or fewer	5
Average	4 to 7	10
Complex	More than 7	15

Repeat this process for each use case in the project. The sum of the weights for each use case is known as the *Unadjusted Use Case Weight*, or UUCW. Table 6.2 shows how to calculate UUCP for a project with 12 simple use cases, eight average, and eight complex.

Table 6.2 Calculating Unadjusted Use Case Weight (UUCW) for a simple project.

Use case complexity	**Weight**	**Number of use cases**	**Product**
Simple	5	12	60
Average	10	8	80
Complex	15	8	120
Total			**260**

Unadjusted Actor Weight

The transactions (or steps) of a use case are one aspect of the complexity of a use case, the actors involved in the use cases are another. An actor in a use case might be a person, another program, a piece of hardware, and so on. Some actors, such as a user working with a straightforward command-line interface, have very simple needs and increase the complexity of a use case only slightly. Other actors, such as a user working with a highly interactive graphical user interface, have a much more significant impact on the effort to develop a use case. To capture these differences, each actor in the system is classified as either simple, average, or complex and is assigned a weight in the same way the use cases were weighted.

As simple actor is an another system that is interacted with through an API (Application Programming Interface). An average actor may be either a person interacting through a text-based user interface or another system interacting through a protocol such as TCP/IP, HTTP, or SOAP. A complex actor is a human interacting with the system though a graphical user interface. This is summarized, and the weight of each actor type is given, in Table 6.3.

Table 6.3 Actor complexity.

Actor type	Example	Weight
Simple	Another system through an API	5
Average	Another system through a protocal A person through a text-based user interface	10
Complex	A person through a graphical user interface	15

Each actor in the proposed system is assessed as either simple, average, or complex and is weighted accordingly. This is shown for a sample project in Table 6.4.

Table 6.4 Calculating Unadjusted Actor Weight (UAW) for a sample project.

Actor type	Weight	Number of use cases	Product
Simple	5	4	20
Average	10	3	30
Complex	15	6	90
Total			**140**

Technical Complexity

The total effort to develop a system is influenced by factors beyond the collection of use cases that describe the functionality of the intended system. A distributed system will take more effort to develop than a non-distributed system. Similarly, a system with difficult to meet performance objectives will take more effort than one with easily-met performance objectives. The impact on use case points of the technical complexity of a project is captured by assessing the project on each of thirteen factors, as shown in Table 6.5. Many of these factors represent the impact of a project's non-functional requirements on the project's size. The project is assessed and rated from 0 (irrelevant) to 5 (very important) for each of these factors.

Table 6.5 The weight of each factor impacting technical complexity.

Factor	Description	Weight
T1	Distributed system	2
T2	Performance objectives	2
T3	End-user efficiency	1
T4	Complex processing	1
T5	Reusable code	1
T6	Easy to install	0.5
T7	Easy to use	0.5
T8	Portable	2
T9	Easy to change	1
T10	Concurrent use	1
T11	Security	1
T12	Access for third parties	1
T13	Training needs	1

An example assessment of a project's technical factors is shown in Table 6.6. This project is a web-based system for making investment decisions and trading mutual funds. It is somewhat distributed and is given a three for that factor. Users expect good performance but nothing above or beyond a normal web application so it is given a three for performance objectives. End users will expect to be efficient but there are no exceptional demands in this area. Processing is relatively straightforward but some areas deal with money and we'll need to be more carefully developing, leading to a two for complex processing. There is no need to pursue reusable code and the system will not be installed outside the developing company's walls so these areas are given zeroes. It is extremely important that the system be easy to use, so it is given a four in that area. There are no portability concerns beyond a mild desire to keep database server options open. The system is expected to grow and change dramatically if the company succeeds and so a five is given for the system being easy to change. The system needs to support concurrent use by thousands of users and is given a five in that area as well. Because the system is handling money and mutual funds, security is a significant concern and is given a five. Some slightly restricted access will be given to third-party partners and that area is given a three. Finally, there are no unique tranining needs so that is assessed as a zero.

Table 6.6 Example assessment of a project's technical factors.

Factor	**Weight**	**Assessment**	**Impact**
Distributed system	2	3	6
Performance objectives	2	3	6
End-user efficiency	1	3	3
Complex processing	1	2	2
Reusable code	1	0	0
Easy to install	0.5	0	0
Easy to use	0.5	4	2
Portable	2	2	4
Easy to change	1	5	5
Concurrent use	1	5	5
Security	1	5	5
Access for third parties	1	3	3
Training needs	1	0	1
Total (TFactor)			**42**

The weighted assessments for these twelve individual factors are summed into what is called the *TFactor*. The TFactor is then used to calculate the *Technical Complexity Factor,* TCF, as follows:

$$\text{TCF} = 0.6 + (0.01 \bullet \text{TFactor})$$

In our example, TCF = 0.6 + (0.01 * 42) = 1.02.

Environmental Complexity

Environmental factors also affect the size of a project. The motivation level of the team, their experience with the application, and other factors affect the calculation of use case points. Table 6.7 shows the eight environmental factors that are considered for each project.

Table 6.7 Environmental factors and their weights.

Factor	Description	Weight
E1	Familiar with the development process	1.5
E2	Application experience	0.5
E3	Object-oriented experience	1
E4	Lead analyst capability	0.5
E5	Motivation	1
E6	Stable requirements	2
E7	Part-time staff	-1
E8	Difficult programming language	-1

An example assessment of a project's environmental factors is shown in Table 6.8. The weighted assessments for these eight individual factors are summed into what is called the *EFactor*. The EFactor is then used to calculate the *Environment Factor,* EF, as follows:

$$\mathrm{EF} = 1.4 + (-0.03 \bullet \mathrm{EFactor})$$

In our example, this leads to:

$$\mathrm{EF} = 1.4 + (-0.03 \bullet 17.5) = 1.4 + (-0.51) = 0.89$$

Putting It All Together

We can now use the Unadjusted Use Case Weight (UUCW), the Unadjusted Actor Weight (UAW), the Technical Complexity Factor (TCF), and the Environmental Factor (EF) to calculate Use Case Points (UCP) with this simple formula:

$$\mathrm{UCP} = (\mathrm{UUCW} + \mathrm{UAW}) \bullet \mathrm{TCF} \bullet \mathrm{EF}$$

The values that were determined for these components in the example throughout this chapter are shown in Table 6.9. Substituting values from Table 6.9 into the equation, we get:

$$\mathrm{UCP} = (260 + 140) \bullet 1.02 \bullet 0.89 = 363$$

Table 6.8 Example calculation of EFactor.

Factor	**Weight**	**Assessment**	**Impact**
Familiar with the development process	1.5	3	4.5
Application experience	0.5	4	2
Object-oriented experience	1	4	4
Lead analyst capability	0.5	4	2
Motivation	1	5	5
Stable requirements	2	1	2
Part-time staff	-1	0	0
Difficult programming language	-1	2	–2
Total (EFactor)			**17.5**

Table 6.9 Component values for determining total Use Case Points.

Factor	**Description**	**Weight**
UUCW	Unadjusted Use Case Weight	260
UAW	Unadjusted Actor Weight	140
TCF	Technical Complexity Factor	1.02
EF	Environmental Factor	0.89

From Use Case Points to Time

Karner (1993) originally proposed a ratio of 20 hours per use case point. This means that our example of 363 use case points translates into 7,260 hours of development work. Building on Karner's work, Kirsten Ribu (2001) reports that this effort can range from 15 to 30 hours per use case point. In much the same way an XP team will often need to take a wild guess at its initial velocity, a team working with use cases may need to take a similar guess at first.

A different approach is proposed by Schneider and Winters (1998). They suggest counting the number of environmental factors in E1 through E6 that are above 3 and those in E7 and E8 that are below three. If the total is two or less, assume 20 hours per use case point. If the total is 3 or 4, assume 28 hours per use case. Any total larger than 4 indicates that there are two many environmentals factors stacked against the project. The project should be put on hold until some environmental factors can be improved.

Using Use Case Points

The most obvious use for use case points is predicting an overall completion date for a project. For example, suppose we make the following assumptions:

- The project has 363 use case points
- The team will average 20 hours per use case point
- Iterations will be two weeks long
- There are five developers on the team

In this case, the complete project will take 7,260 hours to complete (363 use case points * 20 hours per point). Developers will work a combined total of 400 hours per iteration (5 developers * 40 hours per week * 2 weeks). Dividing 7,260 hours by 400 hours indicates that the overall project is likely to take 19 iterations (rounding up from 7,260 / 400 = 18.15). This means this project is likely to be done in about 36 to 38 weeks.

Use Case Points for Release Planning

Although not as straightforward as story points, it is possible to make use of use case points in release and iteration planning. To do this, we return to the Unadjusted Use Case Weight (UUCW), which was simply the number of use cases weighted by the complexity of each. As you'll recall, each use case was assessed as simple, average, or complex based on the number of transactions it performed. We'd like to be able to schedule iterations based on the collection of use cases and their complexities.

In the extended example we've been working throughout this chapter, we determined that the project would take 19 iterations. We also know that the UUCW was 260. If we divide the UUCW by the number of iterations we get the weight of use cases that should be targeted in each iteration. In this case, that is 260 / 19 = 13.7 or 14. This is just short of a complex use case per iteration. Naturally, the team can make up that work however they and the customer agree—two or three simple use cases, one average and one simple use case, or one complex use case.

The resulting of dividing UUCW by the number of iterations can be thought of as the velocity of a use case-driven team. As the team completes iterations, they should monitor the amount of UUCW completed and use the moving average of perhaps the last two or three iterations as their velocity.

Some Problems with Use Case Points

Use cases are large units of work to be used in planning a system. As we've seen in this chapter's example, only 28 use cases drive 38 weeks of work. While use case points may work well for creating a rough, initial estimate of overall project size they are much less useful in driving the iteration to iteration work of a team. In this case the team needs to plan on completing 14 UUCW each iteration. However, UUCW only comes in increments of five. The team must therefor either undercommit and plan for 10 or overcommit and plan for 15.

The team can, of course, elect to split use cases into multiple, smaller use cases. However, this presents a problem because it will be difficult to assign value to the smaller use cases. In total, the new use cases will need to equal the same UUCW as the use case being split. But, for example, a team that is under pressure to make progress could subtly overweight the portion they'll work on first. This will understate the amount of work remaining, which will cause problems during the latter portions of the project.

An additional problem with use case points is that some of the Technical Factors (shown in Table AAAA) do not really have an impact across the overall project. Yet, because of the way they are multiplied with the weight of the use cases and actors the impact is such that they do. For example, techinical factor T6 reflects the requires for being able to easily install the system. Yes, in some ways, the larger a system is, the more time-consuming it will be to write its installation procedure. However, I typically feel much more comfortable thinking of installation requirements on their own (for example, as separate stories) rather than as a multiplier against the overall size of the system.

Use Case Scenarios as Stories

Let's look at one possible alternative to working with use case points for teams that are driving development from use cases. Recall that each use case has a single main success scenario but that each use case may contain a variety of additional scenarios. It has been argued elsewhere (Cohn 2004) that each scenario of a use case can be thought of as equivalent to a user story. If that's the case, the scenarios of a use case can be individually estimated and scheduled just like user stories.

Considering each scenario of a use case as a story and estimating it individually allows a team to work with much smaller units of work. It is no longer necessary to schedule an entire use case into an iteration. Instead, scenarios from one or more use cases can be scheduled into an interation based on the aggregate size of the scenarios. Further, working with scenarios instead of complete use cases allows for more effective prioritization of work. If most of the value of a particular use case can be achieved with a subset of its stories then just that subset needs to scheduled.

A Combined Approach

It's possible to combine use case points with estimating the scenarios of a use case. To do this, calculate the use case points of the project. Then, convert use case points into hours and hours into a number of iterations as described in "Use Case Points for Release Planning." Next estimate each scenario as if it were a user story and sum those estimates. Finally, divide that sum by the number of iterations the use case point method implied and see if the resulting implied velocity makes sense. Any time we can estimate something

As an example, suppose we estimate each scenario in our example program and the sum of those estimates is 323 story points. We divide the 323 story points by the 19 iterations we estimated from the use case points and come up with a velocity of 17. If a velocity of 17 makes sense to the team (based on how they estimated) then they have applied two approaches to the problem and should feel better about the overall estimate of 19 iterations than if they had used either method in isolation.

Once the project is underway, abandon the use case points and allocate and monitor work by using the story point estimates of each scenario.

Summary

- SUMMARY POINTS WILL BE ADDED LATER

Questions

- QUESTIONS WILL BE ADDED LATER