# Estimating With Use Case Points

by Mike Cohn, mike@mountaingoatsoftware.com

If you've worked with use cases, you've probably felt there should be an easy way to estimate the overall size of a project from all the work that went into writing 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*. Similar in concept to function points, use case points measure the size of an application. Once we know the approximate size of an application, we can derive an expected duration for the project if we also know (or can estimate) the team's rate of progress.

Use case points were first described by Gustav Karner, but his initial work on the subject is closely guarded by Rational Software. This article, therefore, primarily documents Karner's work as describer by Schneider and Winters (1998) and Ribu (2001).

## 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 nonfunctional and environmental factors.

Fundamental to the use of use case points is the need for all use cases to be written at approximately the same level. Alistair Cockburn (2001) identifies five levels for use cases: very high summary, summary, user goal, subfunction, and too low. Cockburn's very high summary and 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 recommends that user goal-level use cases form the foundation of a well thought through collection of use cases. At a lower level, subfunction use cases are written to provide detail on an as-needed basis.

If a project team wishes to estimate with use case points, they should write their use cases at Cockburn's user goal level. Each use case (at all levels of Cockburn's hierarchy) has a goal. The goal of a user goal-level use case is a fundamental unit of business value. There are two tests for the whether a user goal use case is written at the proper level: First, the more often the user achieves the goal, the more value is delivered to the business;

Second, the use case is normally completed within a single session and after the goal is achieved, the user may go on to some other activity.

A sample user goal use case is shown in Figure 1. This use case is from a job posting and search site. It describes the situation in which a third-party recruiter has already posted a job opening on the site and now needs to submit payment for placing that ad.

Since this is not an article on use cases, I won't fully cover all the details of the use case shown in Figure 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 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.

| | |
|---|---|
| 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 1. A sample use case for pay for a job posting

## *Unadjusted Use Case Weight*

If all of a project's use cases are written at approximately the level of detail shown in Figure 1, it's possible to calculate use case points from them. Unlike an expert opinion-based estimating approach where the team discusses items and estimates them, use case points are assigned by a formula. In Karner's original formula, each use case is assigned a number of points based on the number of transactions within the use case. A transaction (at least when working with user goal-level use cases) is equivalent to a step in the use case. Therefore we can determine the number of transactions by counting the 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 when he first proposed use case points (1993). Extensions clearly 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.

In Figure 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 1 is not counted. The same, of course, is true for 2b, 2c, and 3a. The transaction count for the use case in Figure 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 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.

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

Table 1. Use case weights based on the number of transactions

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 2 shows how to calculate UUCW for a project with 40 simple use cases, 21 average, and 10 complex.

| **Use case complexity** | **Weight** | **Number of use cases** | **Product** |
|---|---|---|---|
| Simple | 5 | 40 | 200 |
| Average | 10 | 21 | 210 |
| Complex | 15 | 10 | 150 |
| **Total** | | | **560** |

Table 2. Calculating Unadjusted Use Case Weight (UUCW) for a simple project

## 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 a use case 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 simple, average, or complex, and is assigned a weight in the same way the use cases were weighted.

In Karner's use case point system, a simple actor is 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 3.

| Actor Type | Example | Weight |
|---|---|---|
| Simple | Another system through an API | 1 |
| Average | Another system through a protocal<br>A person through a text-based user interface | 2 |
| Complex | A person through through a graphical user interface | 3 |

Table 3. Actor complexity

Each actor in the proposed system is assessed as either simple, average, or complex and is weighted accordingly. The sum of all actor weights in known as *Unadjusted Actor Weight* (UAW). This is shown for a sample project in Table 4.

| Actor Type | Weight | Number of Actors | Product |
|---|---|---|---|
| Simple | 1 | 8 | 8 |
| Average | 2 | 7 | 14 |
| Complex | 3 | 6 | 18 |
| **Total** | | | **40** |

Table 4. Calculating Unadjusted Actor Weight (UAW) for a sample project

### *Unadjusted Use Case Points*

At this point we have the two values that represent the size of the system to be built. Combining the Unadjusted Use Case Weight (UUCW) and the Unadjusted Actor Weight (UAW) gives the unadjusted size of the overall system. This is referred to as *Unadjusted Use Case Points* (UUCP) and is determined by this equation:

$$UUCP = UUCW + UAW$$

Using our example, UUCP is calculated as:

$$UUCP = 560 + 40 = 600$$

To this estimate of the size of the application, Karner's use case points approach next applies a pair of adjustments to reflect the technical and environmental complexity associated with the system being developed.

## Adjusting For 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 nondistributed 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 5. Many of these factors represent the impact of a project's nonfunctional requirements on the effort to complete the project. The project is assessed and rated from 0 (irrelevant) to 5 (very important) for each of these factors.

| 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 |

Table 5. The weight of each factor impacting technical complexity

An example assessment of a project's technical factors is shown in Table 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 vendor 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 tens of 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 training needs so that is assessed as a zero.

| **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** |

Table 6. Example assessment of a project's technical factors

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

$$TCF = 0.6 + (0.01 \times TFactor)$$

In our example, $TCF = 0.6 + (0.01 \times 42) = 1.02$.

## *Adjusting For 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 7 shows the eight environmental factors Karner's formulas consider for each project.

| **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 |

Table 7. Environmental factors and their weights

An example assessment of a project's environmental factors is shown in Table 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:

$$EF = 1.4 + (-0.03 \times EFactor)$$

In our example, this leads to:

$$EF = 1.4 + (-0.03 \times 17.5) = 1.4 + (-0.51) = 0.89$$

| **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 8. Example calculation of EFactor

## *Putting It All Together*

To come up with our final *Use Case Point* (UCP) total, Karner's formula takes the Unadjusted Use Case Points (UUCP, the sum of the Unadjusted Use Case Weight and the Unadjusted Actor Weight) and adjusts it by the Technical Complexity Factor (TCF) and the Environmental Factor (EF). This is done with the following formula:

$$UCP = UUCW \times TCF \times EF$$

The values that were determined for these components in the example throughout this article are summarized in Table 9. Substituting values from Table 9 into the UCP formula, we get:

$$UCP = 600 \times 1.02 \times 0.89 = 545$$

| Factor | Description | Weight |
|---|---|---|
| UUCW | Unadjusted Use Case Weight | 560 |
| UAW | Unadjusted Actor Weight | 40 |
| TCF | Technical Complexity Factor | 1.02 |
| EF | Environmental Factor | 0.89 |

Table 9. Component values for determining total Use Case Points

## Deriving Duration

First, notice that this section is titled "Deriving Duration." It is not called "Estimating Duration." An appropriate approach to planning a project is that we estimate size and derive duration. Use case points are an estimate of the size of a project. We cannot, however, go to a project sponsor who has asked how long a project will take and give the answer "545 use case points" and leave it at that. From that estimate of size we need to derive an appropriate duration for the project. Deriving duration is simple—all we need to know is the team's rate of progress through the use cases.

Karner originally proposed a ratio of 20 hours per use case point. This means that our example of 545 use case points translates into 10,900 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. 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 too many environmental factors stacked against the project. The project should be put on hold until some environmental factors can be improved.

Rather than use an estimated number of hours per use case point from one of these sources, a better solution is to calculate your organization's own historical average from past projects. For example, if five recent projects included 2,000 use case points and represented 44,000 hours of work, you would know that your organization's average is 22 hours per use case point $(44{,}000 \div 800 = 22)$. If you are going to estimate with use case points, it is definitely worth starting a project repository for this type of data.

To derive an estimated duration for a project, select a range of hours. For example, you may use Scheider and Winters' range of 20 to 28 hours per use case point. Based on your experience with writing use cases, estimating in use case points, and the domain of the application you might want to widen or narrow this range. Using the range of hours and the number of use case points, you can derive how long the project will probably take. For example, suppose we have the following information:

- The project has 545 use case points
- The team will average between 20 and 28 hours per use case point
- Iterations will be two weeks long
- A total of ten developers (programmers, testers, DBAs, designers, etc.) will work on this project

In this case, the complete project will take between 10,900 hours and 15,260 hours to complete ($545 \times 20 = 10{,}900$ and $545 \times 28 = 15{,}260$). We estimate that each developer will spend about 30 hours per week on project tasks. The rest of their time will be sucked up by corporate overhead—answering email, attending meetings, and so on. With ten developers, this means the team will make $10 \times 30 = 300$ hours per week or 600 hours of progress per iteration. Dividing 10,900 hours by 600 hours and rounding up indicates that the overall project might take 19 two-week iterations. Dividing 15,260 by 600 hours and rounding up indicates that it might take 26 two-week iterations. Our estimate is then that this project will take between 19 and 26 two-week iterations (38 to 52 weeks).

## Some Agile Adaptations

As originally conceived, a use case point approach to estimating is not particularly suited to teams using an agile software development process such as Scrum or Extreme Programming. This is one of the reasons I ultimately chose not to describe the approach in my book *Agile Estimating and Planning* (Cohn 2005). In particular, the need to create a complete use case model at the user goal level is incompatible with agile values because it encourages the early creation of a (supposedly complete) set of requirements. However, because many teams work with use cases and because many of them are moving in agile directions, it is worth suggesting how the approach can be applied in a semi-agile context.

### *Tracking Progress*

One of the most useful techniques to come out of agile software development is the burndown chart (Schwaber and Beedle 2001). A typical release burndown chart shows the estimated amount of time remaining in a project as of the start of each iteration. The sample burndown chart in Figure 2 shows a project that had approximately 250 days of work at the start of the first iteration, about 200 by the start of the second iteration, and about 175 by the start of the third iteration. Things didn't go well during the third iteration, and by the start of the fourth iteration the team was back to an estimate of 200 days of work remaining. The cause of this increase is unknowable from the burndown chart. But this is usually the result of adding new requirements to the project or of discovering that some upcoming work had been incorrectly estimated.

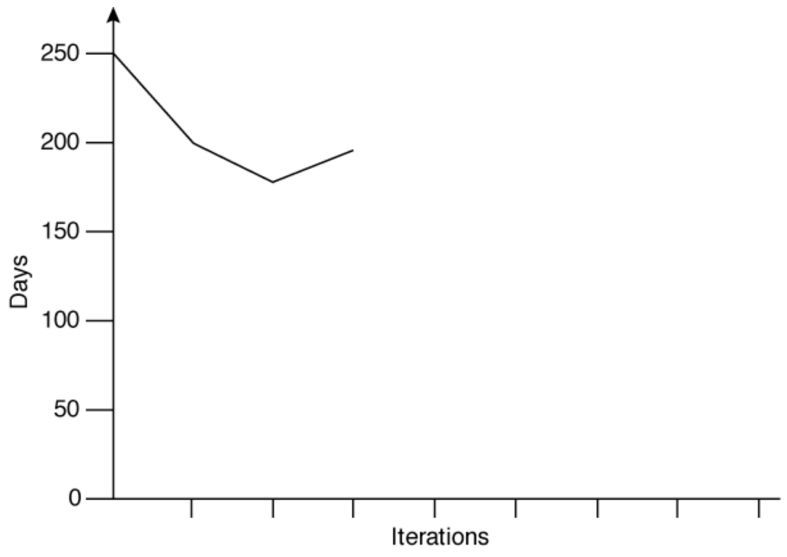

Figure 2. A sample burndown chart

Having become addicted to the use of burndown charts as a technique for monitoring the progress of a team, I am reluctant to let go of such a powerful communication and tracking tool. Fortunately, there is a way to use a burndown chart even for projects that estimate in use case points.

The best way to do this is to use only the Unadjusted Use Case Weight on the vertical axis, and to allow a team to burndown 5, 10, or 15 points for every simple, average, and complex use case they finish. (You'll recall these were the weightings shown in Table 1.) For the sample project discussed throughout this article, the intercept on the vertical axis would then be at 560, the Unadjusted Use Case Weight as calculated in Table 2. The burndown chart shown in Figure 3 starts at this point and shows the team's progress through the first two iterations.

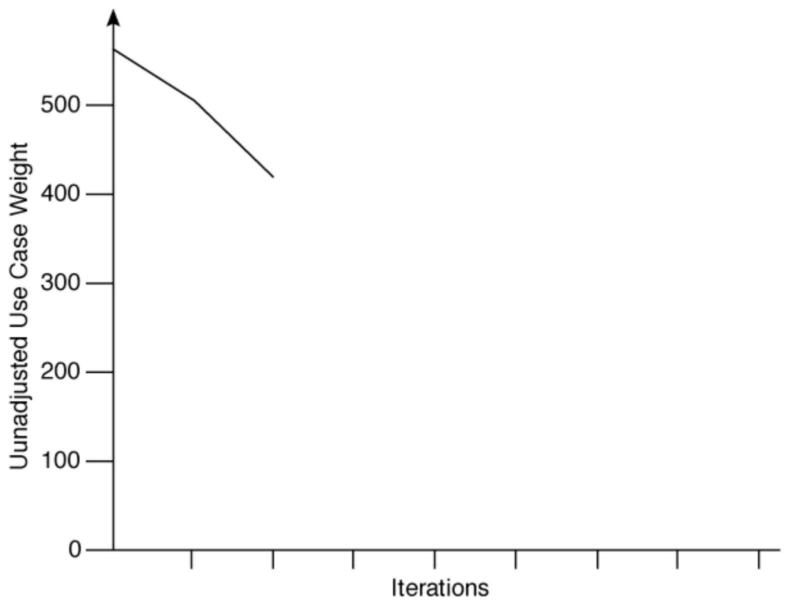

Figure 3. A burndown chart of Unadjusted Use Case Weight

### *Measuring Velocity*

Agile teams like to measure their *velocity*, which is their rate of progress. With a use case point approach and with burndown charts drawn as described in the prior section, velocity is calculated as the sum of the weights of the use cases completed during an iteration.

## Advantages and Disadvantages to Estimating with Use Case Points

As with most things, there are some advantages and disadvantages to the use case point approach. The final two sections of this article briefly outline the key issues.

### *Advantages*

The first advantage to estimating with use case points is that the process can be automated. Some use case management tools will automatically count the number of use case points in a system. This can save the team a great deal of estimating time. Of course, there's the counter argument that an estimate is only as good as the effort put into it.

A second advantage is that it should be possible to establish an organizational average implementation time per use case point. This would be very useful in forecasting future schedules. Unfortunately, this depends heavily on the assumption that all use cases are consistently written with the same level of detail. This may be a very false assumption, especially when there are multiple use case authors.

A third advantage to use case points is that they are a very pure measure of size. Good estimation approaches allow us to separate estimating of size from deriving duration. Use case points qualify in this regard because the size of an application will be independent of the size, skill, and experience of the team that implements it.

### *Disadvantages*

A fundamental problem with estimating with use case points is that the estimate cannot be arrived at until all of the use cases are written. Writing user goal use cases is a significant effort that can represent 10–20% of the overall effort of the project. This investment delays the point at which the team can create a release plan. More important, if all the use cases are all written up front, there is no learning based on working software during this period.

Use cases are large units of work to be used in planning a system. As we've seen in this article's example, 71 use cases can drive 38 to 52 weeks of work for a ten-person team. 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.

A better approach will often be to break the use case into a set of user stories and estimate the user stories in either story points or ideal time (Cohn 2005).

A related issue is that the rules for determining what constitutes a transaction are imprecise. Counting the number of steps in a user goal user story is an approximation. However, since the detail reflected in a use case varies tremendously by the author of the use case, the approach is flawed.

An additional problem with use case points is that some of the Technical Factors (shown in Table 5) 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, technical factor T6 reflects the requirement 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 user stories) rather than as a multiplier against the overall size of the system.

## References

Cockburn, Alistair. 2001. *Writing Effective Use Cases.* Addison-Wesley.

Cohn, Mike. 2004. *User Stories Applied for Agile Software Development.* Addison-Wesley.

Cohn, Mike. 2005. *Agile Estimating and Planning.* Addison-Wesley.

Ribu, Kirsten. 2001. *Estimating Object-Oriented Software Projects with Use Cases.* Master of Science Thesis, University of Oslo, Department of Informatics.

Schwaber, Ken and Mike Beedle. 2001. *Agile Software Development with Scrum.* Prentice Hall.

Schneider, Geri and Jason P. Winters. 1998. *Applying Use Cases: A Practical Guide.* Addison Wesley.

## About The Author

Mike Cohn is the founder of Mountain Goat Software, a process and project management consultancy and training firm. He is the author of *User Stories Applied for Agile Software Development* and *Agile Estimating and Planning*, as well as books on Java and C++ programming. With more than 20 years of experience, Mike has previously been a technology executive in companies of various sizes, from startup to Fortune 40. A frequent magazine contributor and conference speaker, Mike is a Certified ScrumMaster Trainer and a founding member of the Agile Alliance, and serves on its board of directors. He can be reached at mike@mountaingoatsoftware.com.