METHODS & TOOLS

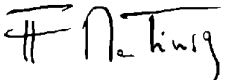
Global knowledge source for software development professionals **ISSN 1661-402X**

Fall 2005 (Volume 13 - number 3)

Man-Machine Interface

The reduction of the transformation activities between man thoughts and executable software has been the quest of the software development world for a very long time. Behind this objective is the fact that the translation process to executable instructions is where distortions and errors are created. Among the solutions adopted to try to solve this problem, we can mention the approach that wants to express computer instructions in syntax as close as possible to "natural language" and the efforts to transform specifications automatically in executable code. With the natural language approach, you meet quickly the barrier of giving a clear meaning to natural language. If you have tried to build a general interpreter, you have seen how difficult it is to manage the input of natural language. In a similar approach, attempts have been made to create programming languages with "verbose" syntax so that they could be understood easily by non-programmers. The "automatic transformation" road is close to the previous approach but recognise that a special language is needed to express requirements. Its difficulties are the creation of a specification language that can be understood by the end-users and the detail of specifications needed before code generation. Maintaining different levels of abstraction is not easy and you could simply end by writing your code in a high level language.

Both solutions generate even more complexity if you try to implement them so that they could be applicable to all problem domains. The domain specific modelling approach presented in this issue tries to overcome some of the problems mentioned above. It provides a specific environment to model user activities and creates particular transformation processes for the target machine. This is however not an easy way, because it requires enough "brainware" to model the domain and to create the transformation processes. It requires also enough time to perform these activities in a world where "yesterday" is a common deadline for new software development projects. Industries with large software investments could find worthwhile to follow the domain specific modelling path as they could achieve the availability of an automated software development tailored to their needs. But they would have first to recognise the importance of managing their software development process and not outsourcing it, as the current trend seems to be.



Inside

Estimating With Use Case Points ..... page 3

Domain-Specific Modeling for Full Code Generation..... page 14

Agile Requirements..... page 24

Estimating With Use Case Points

Mike Cohn, mike @ mountaingoatsoftware.com
Mountain Goat Software, www.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.

Advertisement – Good Requirements = Better Software - Click on ad to reach advertiser web site

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 protocol A person through a text-based user interface	2
Complex	A person 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.

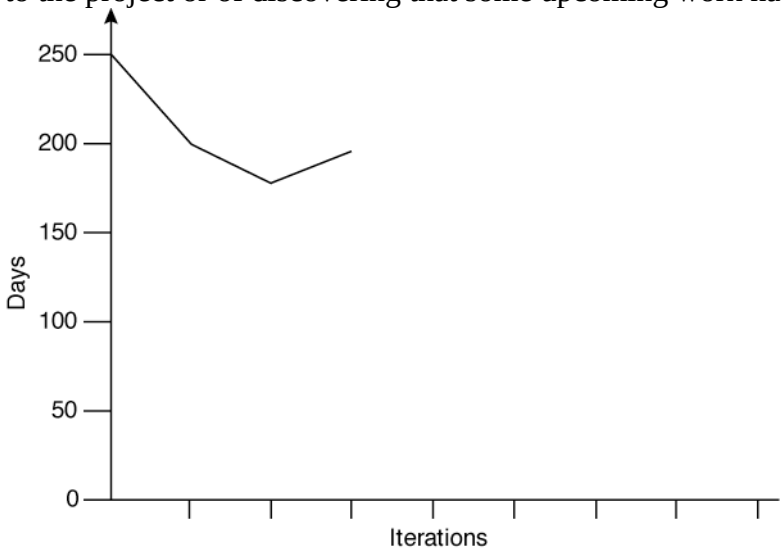


Figure 2. A sample burndown chart

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.

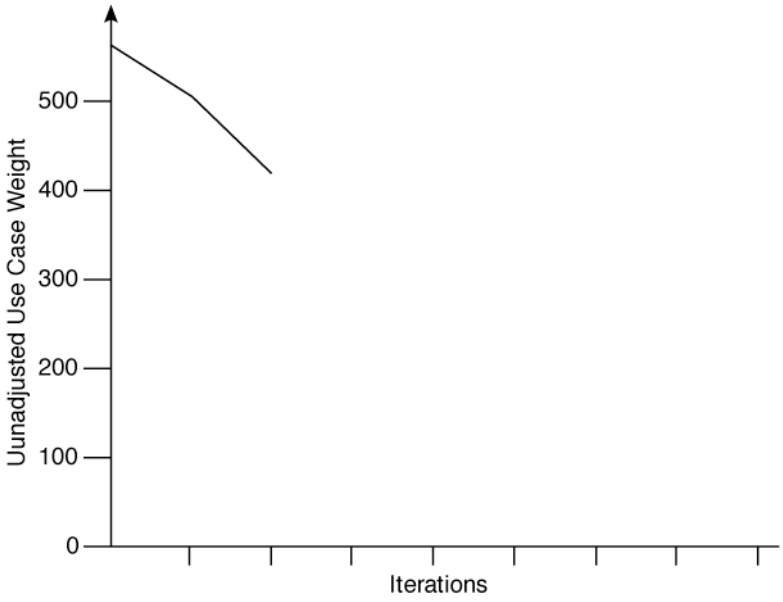


Figure 3. A burndown chart of Unadjusted Use Case Weight

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.

Domain-Specific Modeling for Full Code Generation

Juha-Pekka Tolvanen
MetaCase, www.metacase.com,

Domain-Specific Modeling raises the level of abstraction beyond programming by specifying the solution directly using domain concepts. The final products are generated from these high-level specifications. This automation is possible because both the language and generators need fit the requirements of only one company and domain. This article describes Domain-Specific Modeling (DSM) with examples and compares it to UML and MDA.

UML for code generation?

Generating complete code from models has been an industry goal for many years. Models serve as mechanisms for better understanding and documentation, but they can also be used for generating complete and working code. This automates development leading to improved productivity, quality and complexity hiding.

Unfortunately, many current modeling languages are based on the code world and offer only modest possibilities to raise design abstraction and to achieve full code generation. For example, UML uses directly programming concepts (classes, return values, etc.) as modeling constructs. Having a rectangle symbol to illustrate a class in a diagram and then equivalent textual presentation in a programming language does not provide real generation possibilities – the level of abstraction in models and code is the same! As a consequence of this, developers easily find themselves making models that describe functionality and behavior that they find easier to write directly as code. The limited code generation possibilities force developers to start manual programming after design. It has also lead to round-trip problems: Having the same information in two places, code and models, is a recipe for trouble.

Domain-Specific modeling solution

Generation challenges can be solved in a similar manner as in the past with programming languages: by continuing to raise abstraction. Models should not be conceived to visualize code, but describe higher-level abstractions above programming languages. Similarly, in the past it was better to move to C to raise the abstraction than start visualizing Assembler code!

In Domain-Specific Modeling (DSM), the model elements represent things in the domain world, not the code world. The modeling language follows the domain abstractions and semantics, allowing modelers to perceive themselves as working directly with domain concepts. The rules of the domain can be included into the language as constraints, ideally making it impossible to specify illegal or unwanted design models. The close alignment of language and problem domain offers several benefits. Many of these are common to other ways of moving towards higher levels of abstraction: improved productivity, better hiding of complexity, and better system quality.

The higher level of abstraction varies between applications and products, though. Every domain contains its own specific concepts and correctness constraints. Therefore, modeling languages need to be specific for each domain. Let’s take an example. If we are developing a portal for insurance product comparison and purchase, why not use the insurance terminology directly in the design language? Language concepts like ‘Risk’, ‘Bonus’ and ‘Damage’ capture facts about insurances better than Java classes do. An insurance-specific language can also guarantee that the products modeled are valid: insurance without a premium is not a good product, so such a

product should be impossible to design. These domain concepts are typically already known and in use, are more natural and reflect already the underlying computational models needed to design the products. Final code (assembler, 3GL, object-oriented etc.) can be still generated from these high-level specifications. Cornerstone for the automated code generation from models is that both the language and generators need fit only company's requirements.

DSM examples

Let's demonstrate DSM next using practical examples. We show here three cases from different application domains: cellular phone application, business process workflow and voice menus in an 8-bit microcontroller. These samples focus on application logic and behavior, not just the static structures that are usually easier to generate.

Symbian Series 60 phone applications

Suppose you develop enterprise applications for cellular phones. Your developers make then applications such as inventory status checking, ordering, event registration etc. Before any new applications can be built developers must design them in the phone domain. This involves applying the terms and rules of the phone, such as lists, softkey buttons, views, text messages and user's actions. DSM would apply these very same concepts directly in the modeling language.

An example of such a modeling language is illustrated in Figure 1a. If you are familiar with some phone applications, like phone book or calendar, you most likely already understood what the above application makes. A user can register for a conference using text messages, choose a payment method, view program and speaker data, browse the conference program via the web, or cancel his registration.

The design model is directly based on domain concepts, such as Note, Pop-up, Text Message, Form, and Query. DSM uses these widgets and services of the phone as modeling concepts and ensures that the phone's programming model is followed. It also prevents many illegal or unwanted designs – e.g. those leading to poor performance. The specific generator produces then full code from the design model calling the phone's platform services and executes the result in an emulator or in the target device (Figure1b).

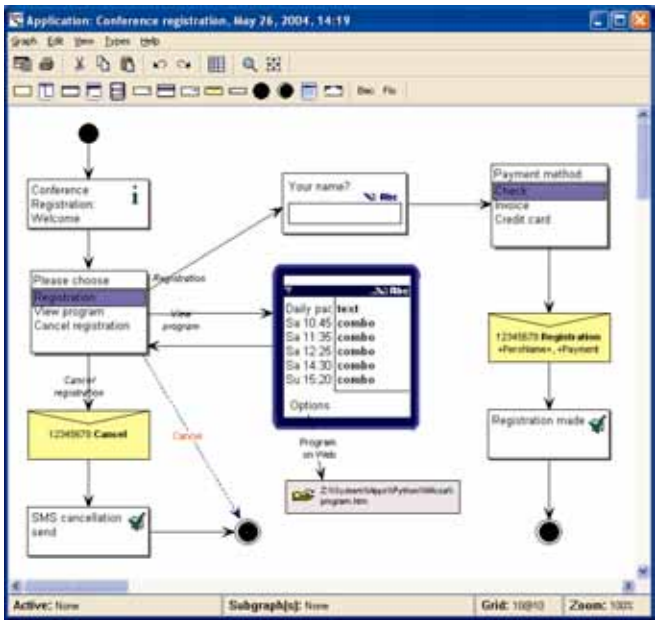


Figure 1a. DSM for designing phone applications

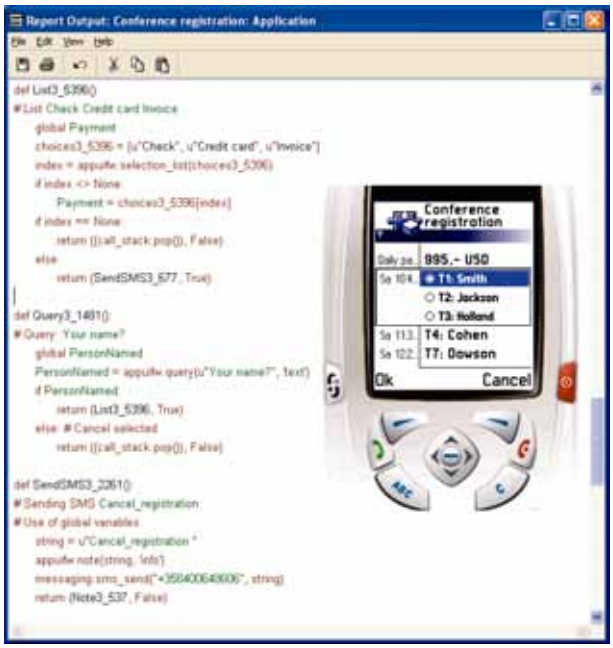


Fig 1b. Generating and executing code on target

Here DSM allows scaling up to significant levels of complexity without losing agility. Even in case fundamental changes in the architecture or platform services occur it is typically adequate to just change the generator or the modeling language.

Business process modeling for workflow execution

This example illustrates how software can be generated based on business process models. Business managers can focus on finding the solution for the company's business processes using natural and well-known concepts. They draw models using concepts like Processes of different kind, Events, Sub-processes, Organizational units etc. as illustrated in Figure 2a. DSM provides these basic modeling concepts along the rules how processes can be defined.

This DSM is also made for generation purposes to produce workflow descriptions in a format that a workflow engine can execute. Making the generator also becomes easier as the specifications don't need to be verified – the DSM language and related constraints have already taken care of that.

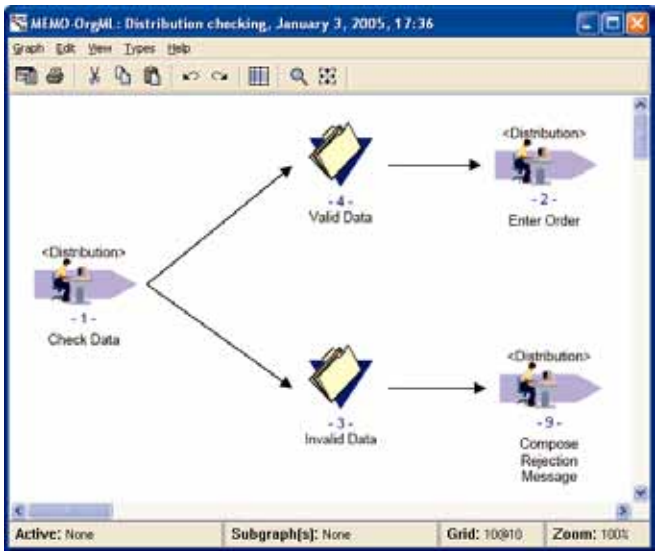


Figure 2a. DSM for business processes

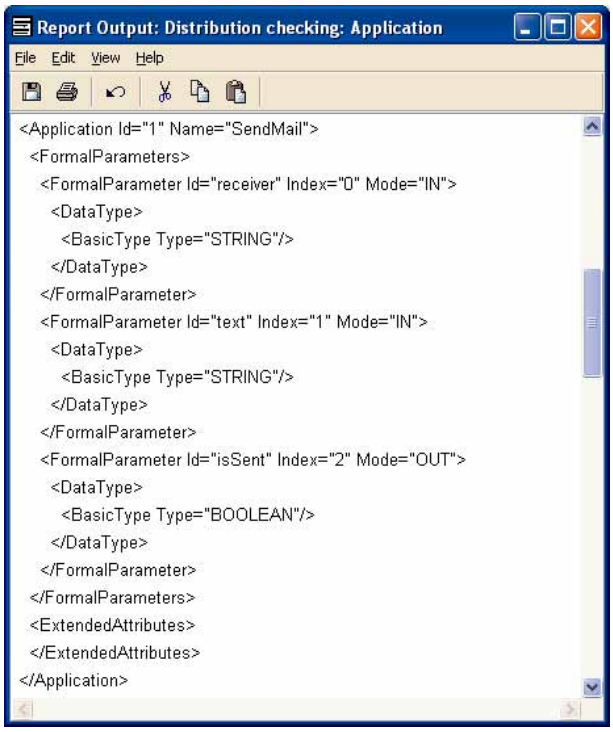


Figure 2b. Generated workflow in XML

Voice menus for an 8-bit microcontroller

This example illustrates a DSM for developing voice menus for a home automation system. Because the target device has limited memory and other resources the DSM aims at optimizing the code. Figure 3a illustrates the flow-like execution of a voice menu system. The DSM provides predefined abstractions that fit to this particular development need. For developers it is far more natural to think about voice communication logic as ‘Menu’, ‘Prompt’, and ‘Voice entry’ than with Assembler mnemonics.

The generator produces assembler with the necessary functionality for memory addressing, calculation, etc. (Figure 3b). It is worthwhile to note that even though the target platform and source code are on a very low level, the DSM language can still operate on pure domain concepts like menu item, voice message and menu selection. Hence, by changing the generator we could produce for instance C from the same designs.

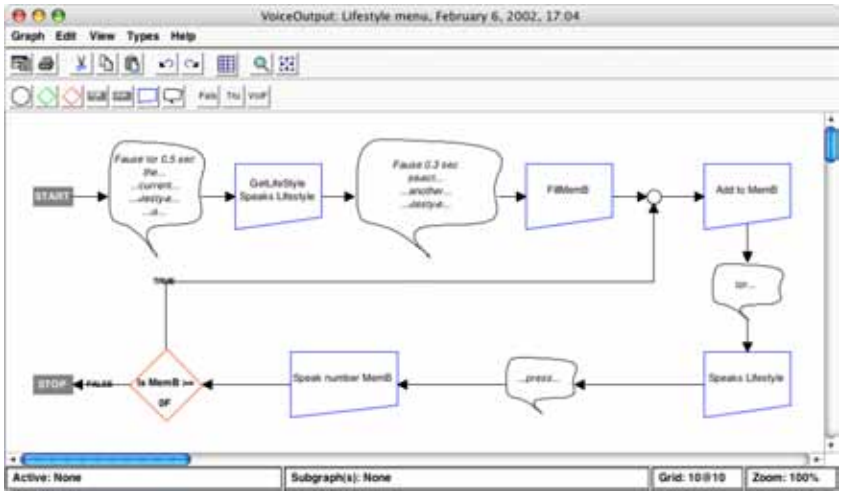


Figure 3a. DSM for voice menu design

Figure 3b. Generated 8-bit code to microcontroller

DSM benefits to developers

The above examples illustrate what DSM looks like and can offer for developers. A higher level of abstraction generally leads to better productivity. This means not only the time and resources to make the designs in the first place, but the also the maintenance resources.

Requirements changes usually come via the problem domain not the implementation domain, so making such changes in a modeling language that uses domain terms is easier. In addition, in some domains, non-programmers can make complete specifications – and run generators to produce the code.

Keeping specifications at a significantly higher level of abstraction than traditional source code or e.g. class diagrams means less specification work. As the language need fit only a specific domain, usually inside only one company, the DSM can be very lightweight. It does not include techniques or constructs that add unnecessary work for developers.

DSM reduces the need of learning new semantics. Problem domain concepts are typically already known and used, well-defined semantics exist and are considered "natural" as they are formed internally. Because domain-specific semantics must be mastered anyway, why not give them first class status? Developers do not need to learn additional semantics (e.g. UML) and map back and forth between domain and UML semantics. This unnecessary mapping takes time and resources, is error-prone, and is carried out by all designers – some doing it better, but often all differently.

DSM leads to a better quality system, mainly because of two reasons. First, DSM can include correctness rules of the domain making it difficult, and often impossible, to draw illegal or unwanted specifications. This is also a far cheaper way to eliminate bugs: the earlier the better. Second, generators provide mapping to a lower abstraction level, normally code, and the generated result does not need to be edited afterwards. This has been the cornerstone of other successful shifts made with programming languages.

Finally, specifications made with domain terms are normally easier to read, understand, remember, validate and communicate with.

How to implement DSM

To get the DSM benefits of improved productivity, quality and complexity hiding, we need a domain-specific language and generators. In the past, you would also have needed to implement the supporting tool set. This was one of the main reasons holding DSM back: after all, implementing CASE tools is hardly a core competence for most organizations.

Today, the work needed is reduced to just defining the language and generators, since open metamodel-based tools for modeling and code generation are available. These tools allow getting a working DSM environment ready within days instead of months, as in case with tools that require additional manual programming. Even more crucial is keeping the development environment responsive to domain changes: these tools can maintain models in synch with language and generator evolution thus making design work safe in real-world industry settings.

We will use the previous Symbian phone application modeling language (Figure 1a) as an example to explain DSM implementation.

Defining the modeling language for a domain

Defining a modeling language involves three aspects: the domain concepts, the notation used to represent these in graphical models, and the rules that guide the modeling process. Defining a complete language is considered a difficult task: this is certainly true if you want to build a language for everyone. The task eases considerably if you make it only for one problem domain in one company.

The key issue for finding domain concepts is the expertise provided by a domain expert or a small team of them. Typically, the expert is an experienced developer who has already developed several products in this domain. He may have developed the architecture behind the products, or been responsible for forming the component library. He can easily identify the domain concepts from its terminology, existing system descriptions, and component services.

In our phone application example, the domain concepts come from the UI elements (e.g. Form, Popup), menu and button structure (user definable keys, menu) and underlying services (e.g. text messages, web browsing). By allocating these concepts to the modeling language and refining them further, we can create the conceptual part of the modeling language. The goal here is to make the chosen concepts map accurately to the domain semantics.

However, pure domain concepts alone do not make a modeling language: we need the domain knowledge of how they can be put together. For this domain, we choose application flow as the basic model of computation. This describes how the domain concepts interact during the application execution (see Figure 1a for an example). We introduce the concepts of Start and Stop and directed flows – with and without conditions. Conditions are used when the user selects one of several possible choices, for example via a list or a menu. We also add concepts for Library code, making it possible to link in third party extensions and call them with parameters.

Next, these basic concepts are enriched with the domain rules. Typically, the rules constrain the use of the language by defining what kinds of connections between concepts are possible. They can specify how certain concepts can be reused and how models can be organized. In our phone

example, rules define which UI elements may have their own menus, which have user-defined buttons, and how different phone services may be called during the application execution. These rules together with the concepts are codified and formalized into a language specification which is often called as a metamodel. A metamodel describes the modeling language, similarly to the way a model describes an application.

To provide the notation part of the modeling language, we define symbols to be the graphical representations of the modeling concepts. As our example domain is user interface related, we can base many of its symbols on the appearance of the phone widgets themselves.

Defining the domain framework

The domain framework provides the interface between the generated code and the underlying platform. In some cases, no extra framework code is needed: generated code can directly call the platform components and their services are enough. Often, though, it is good to define some extra framework utility code or components to make code generation easier. Such components may already exist from earlier development efforts and products, and just need a little tweaking.

In our example, the Symbian/Series 60 enterprise application framework already offers a good set of services, at a higher level than base Series 60. The domain framework thus adds only two functions: a dispatcher to execute the flow of application logic, and view management for multi-view applications.

Developing the code generator

Finally, we want to close the gap between the model and code world by defining the code generator. The generator specifies how information is extracted from the models and transformed into code. The code will be linked with the framework and compile to a finished executable without any additional manual effort. The generated code is thus simply an intermediate by-product on the way to the finished product, like .o files in C compilation.

The key issue in building a code generator is how the models concepts are mapped to code. The domain framework and component library can make this task easier by raising the level of abstraction on the code side. In the simplest cases, each modeling symbol produces certain fixed code, including the values entered into the symbol as arguments. The generator can also generate different code depending on the values in the symbol, the relationships it has with other symbols, or other information in the model.

A simple example of a generator definition for a Note dialog is presented in Figure 4. The Note opens a dialog with information, like “Conference Registration: Welcome” in Figure 1a. Lines 1 and 6 are simply the structure for a generator. Line 2 creates the function definition signature and line 3 a comment. Function naming is based on an internal name that the generator can produce if the developer does not want to give each symbol its own function name.

```
1    report '_Note'
2      'def ';   subreport; '_Internal name'; run;   '():'; newline;
3      '# Note '; :Text; newline;
4      'appuifw.note(u"'; :Text; '", '''; :Note type; ''')'; newline;
5      subreport; '_next element'; run;
6    endreport
```

Figure 4. Code generator for Note dialog

Line 4 produces the call for the platform service. It uses the model data (underlined here for clarity), like the value for the Text property of the Note UI element (in this case, “Conference Registration: Welcome”). Similarly, the modeler chose the ‘Note type’ value in the model from a list of available notification types, like ‘info’ (here) or ‘confirmation’ (used in “Registration made” in Figure 1). The generated line 4 is thus (with model data underlined):

```
appuifw.note(u"Conference Registration: Welcome ", 'info')
```

Since the code generator automates the mapping from model to implementation, every developer makes the Note dialog call similarly: the way the experienced developer has defined it. Finally, line 5 calls another generator definition to follow the application flow to the next phone service or UI element. This generator, named _next element, is also used by other UI elements similar to Notifications.

The generation is not limited to just calling services available from components. The components can also call back to generated code, or generated code can inherit abstract functionality from available code and implement the concrete functionality using design data. The generator can also produce code that the application framework requires but is tedious to design. This minimizes the modeling work and can also hide complexity. In our phone case, for instance, pressing the Cancel button in a dialog need not normally be modeled, as the generator can produce default code that backtracks to the previous element.

As the examples illustrate, DSM only works because it dedicates both the modeling language and code generator to one single domain. Thus model-based code generation can be complete and the generated code efficient. Even if you find code not to be efficient, you most likely can also correct it directly via the generator.

How DSM differs from MDA?

Having demonstrated with examples the main principles of Domain-Specific Modeling we can now compare it to other main model-based development approaches and especially to OMG's Model-Driven Architecture (MDA). At its most basic, MDA involves transforming UML models on a higher level of abstraction into UML models on a lower level of abstraction. Normally there are two levels, platform-independent models (PIMs) and platform-specific models (PSMs). These PIMs and PSMs are plain UML and thus offer no raise in abstraction.

In MDA, at each stage you edit the models in more detail, reverse and round-trip engineer this and in the end you generate substantial code from the final model. The aim the OMG has with MDA is to achieve the ability to use the same PIM on different software platforms and to standardize all translations and model formats so that models become portable between tools from different vendors. Achieving this is very ambitious but also still many years away. This focus however clearly defines the difference between DSM and MDA, and answers the question of when each should be applied.

DSM requires domain expertise, a capability a company can achieve only when continuously working in the same problem domain. These are typically product or system development houses more than project houses. Here platform independence is not an urgent need, although it can be easily achieved with DSM by having different code generators for different software and/or product platforms. Instead, the main focus of DSM is to significantly improve developer productivity.

With MDA, the OMG does not focus on using DSM languages but on generic UML, their own standard modeling language. It is not looking to encapsulate the domain expertise that may exist in a company but assumes this expertise is not present or not relevant. It seems therefore that MDA, if and when the OMG finally achieves the goals it has set for it, would be suitable for systems or application integration projects.

MDA requires a profound knowledge of its methodology, something which is external to the company and has to be gained by experience. Thus whereas the domain expertise needed for DSM is already available and applied in an organization, MDA expertise has to be gained or purchased from outside. In these situations the choice between MDA and DSM is often clear.

Using DSM

Domain-Specific Modeling allows faster development, based on models of the problem domain rather than on models of the code. Our three examples give some illustrations of this. Industrial experiences of DSM show major improvements in productivity, lower development costs and better quality. For example, Nokia states that in this way it now develops mobile phones up to 10 times faster, and Lucent that it improves their productivity by 3-10 times depending on the product. The key factors contributing to this are:

- The problem is solved only once at a high level of abstraction and the final code is generated straight from this solution.

- The focus of developers shifts from the code to the design, the problem itself. Complexity and implementation details can be hidden, and already familiar terminology is emphasized.
- Consistency of products and lower error-rates are achieved thanks to the better uniformity of the development environment and reduced switching between the levels of design and implementation.
- The domain knowledge is made explicit for the development team, being captured in the modeling language and its tool support.

Implementation of DSM is not an extra investment if you consider the whole cycle from design to working code. Rather, it saves development resources: Traditionally all developers work with the problem domain concepts and map them to the implementation concepts manually. And among developers, there are big differences. Some do it better, but many not so well. So let the experienced developers define the concepts and mapping once, and others need not do it again. If an expert specifies the code generator, it produces applications with better quality than could be achieved by normal developers by hand.

Resources

- Pohjonen, R., Kelly, S., Domain-Specific Modeling, Dr. Dobb's Journal, August (2002)
- Greenfield et al. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley (2004)
- Fowler, M., Language Workbenches: The Killer-App for Domain Specific Languages? (http://martinfowler.com/articles/languageWorkbench.html)
- Tolvanen, J.-P, Making model-based code generation work, Embedded Systems, 2004 (http://i.cmpnet.com/embedded/europe/esesep04/esesep04p36.pdf)
- DSMforum.org (www.DSMForum.org)
- Metamodel.com (www.metamodel.com)
- Object Management Group (www.omg.org)

Agile Requirements

Rachel Davies, rachel @ agilexp.com
Agile Experience Ltd, UK, http://www.agilexp.com

On traditional software development projects we talk about *capturing the requirements* - as if we are on a hunt to cage some wild beasts that lurk in the jungle. What is actually meant by this is less exciting - producing documents, which specify a system in enough detail that programmers can work by reference to these documents alone. In theory, it is possible to capture my knowledge in a document from which the reader can extract the same knowledge without distortion. In practice, this may not be an efficient way to communicate requirements for complex software systems.

When setting out our process for developing software, it is important to remember that the primary purpose of a development project is to deliver a software system that generates value to a business. Models and documents are essentially by-products of the development; they do not generate value directly. If we are to optimize the flow of value from software development then we need to think seriously about eliminating unnecessary activities and artefacts.

Agile describes a general approach to software development that is implemented by several methods; the most popular are XP, Scrum and DSDM. The common ground shared by agile methods is that they support values and principles embodied in the Agile Manifesto [1]. Agile software development evolved in response to problems encountered with document-heavy waterfall development.

The way agile teams develop an understanding of system requirements is radically different. The primary artefacts are executable code and tests. Agile teams explore requirements by talking with business people and then developing software for their business customer to try out, in short cycles of weeks (not months or years). Instead of tools to manage requirements documents, what agile teams need are tools that help business users write examples of system use that can be run as tests and tools that support collaborative working.

Limits of Documents

A typical programmer's experience with traditional requirements is working from a document written in dry, formal language, that describes a desired software system. By studying this document, s/he gradually builds a mental model that guides the development of an executable software system. The requirements document acts as both means of communication and data storage (preserving evidence of the request). When you put yourself in the place of this programmer there are several problems with the use of documents as a communication medium that you are likely to encounter. Documents are selective and unidirectional; they may also be ambiguous, vague and conceal gaps.

Unidirectional

Documents are a one-way communication medium; information flows from author to reader. There is no opportunity for the reader to ask questions, offer ideas and insights. The author may try to anticipate questions but cannot be expected to address everything and in an attempt to cover everything, otherwise a clear abstraction may be lost in a forest of pages.

A programmer working from a document is likely to find parts of the document difficult to understand. This may be due to poor choice of words by the author or lack of reader's

background knowledge. How does that programmer get to the bottom of the intended meaning? In a corporate context, s/he may not have met the author and the normal channel for questions will be correspondence via email, which takes time. If working under time-pressure a programmer may make their own conclusion about the intended meaning to avoid delays. Getting answers to questions takes time but guessing the answer can lead to developing the wrong software system.

Selective

The author writes from their personal perspective on the system under development and naturally makes assumptions about the background knowledge of the readers. Requirements documents often describe *what* is required rather than *why* – outlining a desired solution rather than explaining the problem space. Little is said about the needs of the users and business context surrounding the system development. Following traditional development process it is assumed that the programmer does not need to know this information because requirements are chosen purely on business grounds. The job of the programmer is limited to the technical implementation of a system rather than contributing ideas on how to achieve business benefits. However, during the implementation many micro-level decisions will be made by the programmer, an awareness of the system context could help with these.

Cost

When we work in the abstract realm of pure analysis, we are detached from physical constraints such as implementation costs and limits imposed by technology choices. If requirements are considered from a purely business perspective then we may get so creative that we specify a system that exceeds our budget.

When planning a project it is useful to be able to identify the priority of requirements so that we can consider delivering the high value features early and can mark nice-to-have features as potential contingency for schedule slips. However, traditional requirements specifications fix the scope of the entire development, all system features described are equally necessary and their descriptions are intertwined throughout the documents. Expressing requirements in a meshed form makes it difficult to trim scope in response to development delays.

Freezing

The traditional approach is to document the requirements for the whole system and then freeze the requirements to provide a stable base for implementation to proceed. This assumes that we can come to a complete understanding of the system before developing real code. Perhaps this is possible in simple or well-known domains but this is a risky assumption to make for complex software development projects.

This raises a fundamental question of whether the human brain is capable of building a virtual system in our imaginations that we can explore and document. In a traditional waterfall process, requirements documents are the culmination of an analysis phase, during which the problem space has been explored via abstract models. The human mind has limited capacity for complex problems, it is to be expected that when working in the realm of abstract ideas, we may miss scenarios, stakeholders' needs and functions that need to be supported by the system. In his book "Serious Play", Michael Schrage [2] suggests that we can only really determine what we want by interacting with a prototype.

Let's contrast the traditional approach to requirements in software development with how we operate outside the software domain. For example, if we are investing in home improvements such as a new kitchen or bathroom – do we just place an order based on drawings and make no comment until the installation is complete? Most people would be keen to see samples of new fittings and retain the right to change their mind about some aspects as installation proceeds. We would naturally expect that, as the fittings are installed in situ, we are likely to spot some ergonomic issues that were not apparent from the original two-dimensional plans.

Gaps spotted by readers will require the author to update the document. Documents take effort to maintain with significant ceremony to sign-off each new draft. This makes it possible for documents to fall out of step with current agreed working understanding of system requirements and so unreliable sources of information.

The final flaw with freezing requirements early is that we are developing the system for use in a changing world. Time does not stand still during software development, the world around us changes – new business opportunities may arise during the development. When we freeze requirements early we deny the chance to adapt the system to a changing context.

Agile lifecycle

Agile methods are built around a key assumption – we continue to learn about a system and it's operating environment during the project. When we recognize that our understanding develops during the project then we realize that freezing requirements during early project phases will likely cause rework or the delivery of a system that does not meet the customer needs. We need an approach to software development that can embrace change during the project. Agile methods evolve both requirements and the system under development by planning the development in short cycles (weeks rather than months). This is not a new idea [3]. When we develop the system incrementally then this allows the customer/user to try the software developed during the project rather than waiting until the end. Their feedback can be used to refine the system behaviour.

When we move to an iterative lifecycle, we lose the waterfall phases: analysis, design, code and test. These activities now run in parallel. This has an impact on how we organise a project team. Rather than assembling specialist teams dedicated to waterfall phases with handover artefacts, it makes more sense to form a cross-functional project team; a team of individuals whose task is to shape and deliver a software system together. Business people are needed throughout the project to explain how the system under development will generate business value and support the user community through different scenarios. Technical people are needed throughout the project to translate those requirements into executable code and verify the system meets conditions of satisfaction.

Agile communication

When a team works on analysis and software development in parallel, this creates an opportunity to work with requirements in a different way. Agile teams primary means of communicating requirements is conversation. The medium of conversation creates the possibility for information flow between all parties. The problems imposed by communicating via documents drop away.

When agile teams plan development work, this done by the whole team in a workshop. If you listen in, you will hear conversation flowing around the group. The whole team members are partners in this conversation. They discuss scenarios and development options to explore the

requirements before development is planned. The beauty of exploring requirements through team conversation is that we develop a shared understanding and have the opportunity to offer our own ideas. Having a two-way conversation between business and IT and flushing out misinterpretation early may prevent developing code that has to be thrown away.

Stories

The use of Stories is a primary practice of XP [4]. An XP story describes a candidate system feature in language that is meaningful to both customer and developer. Stories have three essential parts: Card, Conversation and Confirmation [5]. Stories must be small enough to implement in a single development cycle (XP teams use a one-week cycle). Index cards provide a low ceremony way to represent stories in development plans. Each story card represents a conversation that has taken place. Automated tests are defined for each story, which are used to confirm that the system supports that story.

Other agile methods also emphasize that requirements descriptions need to be short and loose, the bare-minimum needed to plan development. Typically, requirements are listed as one line description maintained in a prioritized list. Features may be swapped in and out of this backlog and it is assumed details about each requirements will be determined as part of each development cycle.

Requirements specification documents are usually expressed in impersonal and cold language giving us *"whats"* without business motivation. This is quite different from the way we tell stories. Using story form, we start by describing the context and give background to the players in the story. A story takes the players through a chain of events that communicates a desired outcome. A story breathes life into a limp requirement. Through the story, the listener grows to understand the pain experienced by the users and the burning need for the software – this can have a profound effect on motivation within the software development team. By discussing explicit situations and our reactions to them (via the system under construction) we can achieve a shared understanding. Understanding the big picture can help programmers to develop software solutions that are a better fit.

Cards

A document is both a communication medium and a storage mechanism. Although a conversation provides us with a rich communication medium, the words may be lost in the air with no record except in the memories of the participants. During team planning workshops, XP teams use index cards to make notes about stories being discussed. There is no formal form for a story card; it can be written in any way that is useful (although a short name for the story as a heading helps). These cards are used after the weekly planning session to create a visible plan on the wall. Each day the team holds a meeting around the plan to review the work remaining for the week. It's possible that a few days may elapse before I start developing a story but because the team sits together I can replay the conversation with them before I start work.

Agile software development is not a series of mini-waterfalls. The notes on index cards are not the counterparts of requirements documents; this is a different process model. There are several tools available for managing electronic story cards – these seem to miss the point that the story card description is not a requirements document in the traditional sense and treat these as the prime artefact rather than the story tests. Index cards provide a non-intrusive way of documenting a conversation without disrupting the flow. The cards are not a simple input from which a conversation is generated; they are also an output of the conversation. A story card is not a stand-alone document describing the full requirement it is really a brief note that a token that to remind us of key details from a conversation. Our primary source of information during the development is conversation, made possible by co-locating the team.

The return to handwriting on cards appears to be a retrograde step. Surely, people in the software industry should be using the latest software tools for planning? To understand why index cards are preferred for planning with stories, you need to understand that paper has different "affordances" – interactions that it supports – and limitations. For example, *spatial layout* – it is easy to shuffle tasks written on index cards around to create different layouts. When planning it helps to review prioritization from different perspectives – by risk, by value, etc. Another affordance is *visibility* - it is easier for a group of people to read index cards spread out on a table than gather around a computer screen. Index cards can be annotated by grabbing a pen. Grabbing the keyboard in a meeting usually means changing seats.

I have worked with teams who have used data projectors in meetings to make the computer screen visible to all but what happens is the group end up waiting for the operator to keep up with them. The way we interact with today's planning software slows down the meeting dynamics. If we are to use software tools then we need to review how they support group interaction and to remember, recording the conversations is a secondary activity, we don't want it to disrupt the primary activity understanding the desires around the system.

Index cards have some limitations too. If your meeting group is large then some may not be able to read the text on the card. I encourage teams to "write big" using marker pens and move to using sticky-notes on a wall rather than a table if members of the team have problems.

It's worth remembering that the drive for a "paperless" office [6] was motivated by the drive to reduce storage costs of large quantities of data - to remove paper records in filing cabinets. Index cards don't hold a lot of data and can be easily bundled with a rubber band. The notes on the index card are a temporary holder of the requirement. The cards are only used for the purpose of planning; the cards are not intended to become a lasting record. Index cards can be thrown away after they have been translated into acceptance test scripts and committed to version control.

Confirmation

The first task in implementing any story is to create a set of automated system-level tests that clarify the scope of the story. These tests are used to confirm the story has been implemented as expected. These tests remove ambiguity by creating executable tests capturing examples of how business rules should fire.

In 2002, Ward Cunningham published the Fit framework [7] this provides a way for business users to write story tests using desktop tools such as MS Word and Excel. When using a test framework like Fit, tests are typically worked on by a customer-developer pair, fleshing out scenarios to be implemented as part of the story.

The great thing about these tests is that non-technical people can read them and so become living documents that specify the system behaviour and also check that the system continues to support the full suite of stories developed to date. In agile software development it is these customer level story tests that form the requirements repository.

Summary

The software industry has been stuck in an illusion that dry language is less ambiguous and that facts can be isolated from the context and the needs of the business driving the development of a software system. Requirements can be documented in an ivory tower of analysis before engaging with technical labourers who will be building the monumental system. Agile teams have learned that limited abstract ideas ought not to be frozen at the start of software development but need to be refined and explored within an iterative development process.

In agile software development, the sharing of stories binds the team together, as the storyteller offers up their ideas they are woven into the system. People show their interest by asking questions and sharing insights. Suddenly, everyone starts to feel like an equal partner in a project that they are engaged in together. Traditional barriers between departments fall away and people feel energised to make the possibilities discussed a reality.

References

1. Agile Manifesto - http://www.agilemanifesto.org
2. “Serious Play: How the World's Best Companies Simulate to Innovate” by Michael Schrage Harvard Business School Press (1999) ISBN: 0875848141
3. “Agile and Iterative Development” by Craig Larman - Addison Wesley (2003) ISBN: 0131111558
4. “Extreme Programming Explained” 2nd edition by Kent Beck, Cynthia Andres - Addison Wesley (2004) ISBN: 0321278658
5. “Essential XP: Card, Conversation, Confirmation” by Ron Jeffries [http://www.xprogramming.com/xpmag/expCardConversationConfirmation.htm]
6. “The Myth of the Paperless Office” by A.Sellen, R.Harper - The MIT Press (2003) ISBN: 026269283X
7. “Fit for Developing Software” by Rick Mugdrige, Ward Cunningham - Prentice Hall (2004) ISBN: 0321269349

Flexible, web based bug and issue tracking! Woodpecker IT is a completely web-based request, issue and bug tracking tool. You can use it for performing request, version or bug management. Its main function is recording and tracking issues, within a freely defined workflow. Woodpecker IT helps you in increasing your efficiency, lower your costs, integrate your customers and improve the quality of your products.

www.woodpecker-it.com/en/

Load test ASP, ASP.NET web sites and XML web services. Load test ASP, ASP.NET web sites and XML web services with the Advanced .NET Testing System from Red Gate Software (ANTS). Simulate multiple users using your web application so that you know your site works as it should. Prices start from $495.
ANTS Profiler a code profiler giving line level timings for .NET is also now available. Price $295.

http://www.red-gate.com/products/ants_load/index.htm

AdminiTrack offers an effective web-based issue and defect tracking application designed specifically for professional software development teams. See how well AdminiTrack meets your team's issue and defect tracking needs by signing up for a risk free 30-day trial.

www.adminitrack.com

Sparx Systems release Enterprise Architect v5.0. New features include MDA Style Model Transformations, advanced Version Control, RTF Report Generation and Floating License Management (Corporate Edition). Value and power for the whole design team, EA v5.0 is the latest generation UML 2.0 CASE tool. For details and to download the 30 day trial visit:

www.sparxsystems.com

Professional Web-based Issue-Tracking and Project Management System. Looking for the reliable, convenient, secure and completely web-based issue tracking system? BUGtrack allows unlimited amount of users, projects, bugs and unlimited customer support for a low flat rate. Enjoy intentional simplicity of basic operations or take advantage of powerful PRO features like E-mail interface, Open API and many others.

www.ebugtrack.com

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig ISSN 1661-402X
Free subscription on : http://www.methodsandtools.com/forms/submt.php