METHODS & TOOLS

Practical knowledge for the software developer, tester and project manager **ISSN 1661-402X**

Winter 2008 (Volume 16 - number 4) www.methodsandtools.com

From Finger in the Air Estimating to Finger Pointing Tracking

As Giovanni Asproni says it "estimation is a fundamental activity of every project". Or at least it should be. The meaning of "estimation" varies however often for the different participants to software development projects. For the customer, "estimation" means an almost accurate amount of money he will spend and time it will need before he gets what he wants. For the project manager, it is an early commitment to the customer and from the developer to do something for a deadline. For the developer, it is the amount of time that he thinks he could do the stuff, if everything goes right. The experienced developer will often add a 50% cushion to his estimates, knowing that not everything goes right. You can have an additional meaning in the case of outsourcing. For the salesperson, the estimation is the maximum price he can ask, while still having a lot of chances to sign a deal. A lot of young engineers will remind with astonishment hearing in a pre-sales meeting a salesperson answering to the customer that everything he wants is indeed possible... and for a cheap price too ;o)

For a developer, estimation is often equaled to fixing deadlines. He could then feel trapped by the original commitment that created expectations for management and customers. Customers have often two fixed expectations: delivery date and functionalities. Therefore, the effort is the only variable left to adjust the project activity. You can often witness a poor transition from estimation to planning with project managers that dream to have a baby in one month, if only they could manage a team of nine women. Because initial expectations cannot be achieved in many projects, the finger in the air used for estimating get then pointed downwards for blaming people: first from customers to project managers, then from project managers to developers. So print Giovanni's article and give it to your customer before your next project planning meeting.

Methods & Tools wishes to all its readers the very best for a healthy and successful 2009.

[illegible]

Inside

Fingers in the Air: a Gentle Introduction to Software Estimation

Giovanni Asproni, gasproni@asprotunity.com
Asprotunity Limited, www.asprotunity.com

Introduction

Estimation is a fundamental activity of every project. A project has always some goals, and the purpose of estimation is to give a reasonably accurate idea of what is necessary in terms of time, costs, technology, and other resources, to achieve them. Since resources are always limited, an accurate estimate is an invaluable tool for deciding if the goals are realistic, as well as for creating and updating the project plans.

All parties involved in a project need to know something about estimation:

- Customers need a way to check that what they are told is correct, and also if the project is worth their investment
- Managers need to manage the project and also provide estimates to their managers and to the customers in order to set their expectations properly, and enable them to take informed decisions
- Developers need to provide estimates to their managers for the tasks they have to perform, so that managers can produce or update the project plans

Unfortunately, the software development community has a very poor record in estimating anything—in fact is very common for projects to run over-time and over-budget, and deliver poor quality products with fewer features than originally planned.

Part of the problem is that software is quite difficult to estimate. In fact, huge differences in individual productivity, the fact that creative processes are difficult to plan, the fact that software is intangible, and the fact that during the life of the project anything can change - e.g., scope, budget, deadlines, and requirements - make software estimation a challenging task.

However, in my experience, the main cause of poor estimates is that the various stakeholders are often unaware of what estimates are for, and what they should look like. This lack of knowledge means that they have no way to judge if the project goals and the associated expectations are realistic. The result can be either overestimation which causes the allocation of an excess of resources to the project, or, more often, gross underestimation which often leads to “death march” projects [You], which are characterized by “project parameters” that exceed the norm by at least 50%, e.g.:

- The schedule has been compressed to less than half the amount estimated by a rational process
- The staff is less than half it should be in a project of the same size and scope
- The budget and resources are less than half they should be
- The features, functionality and other requirements are twice what they would be under normal circumstances

The rest of the article is an introduction to the software estimation process aimed at project managers, developers and customers who want to get a better understanding of the basics this subject, and avoid to make their projects a death march one.

Some definitions: estimates, targets, and commitments

A typical conversation between a project manager and a programmer talking about estimates often goes along these lines

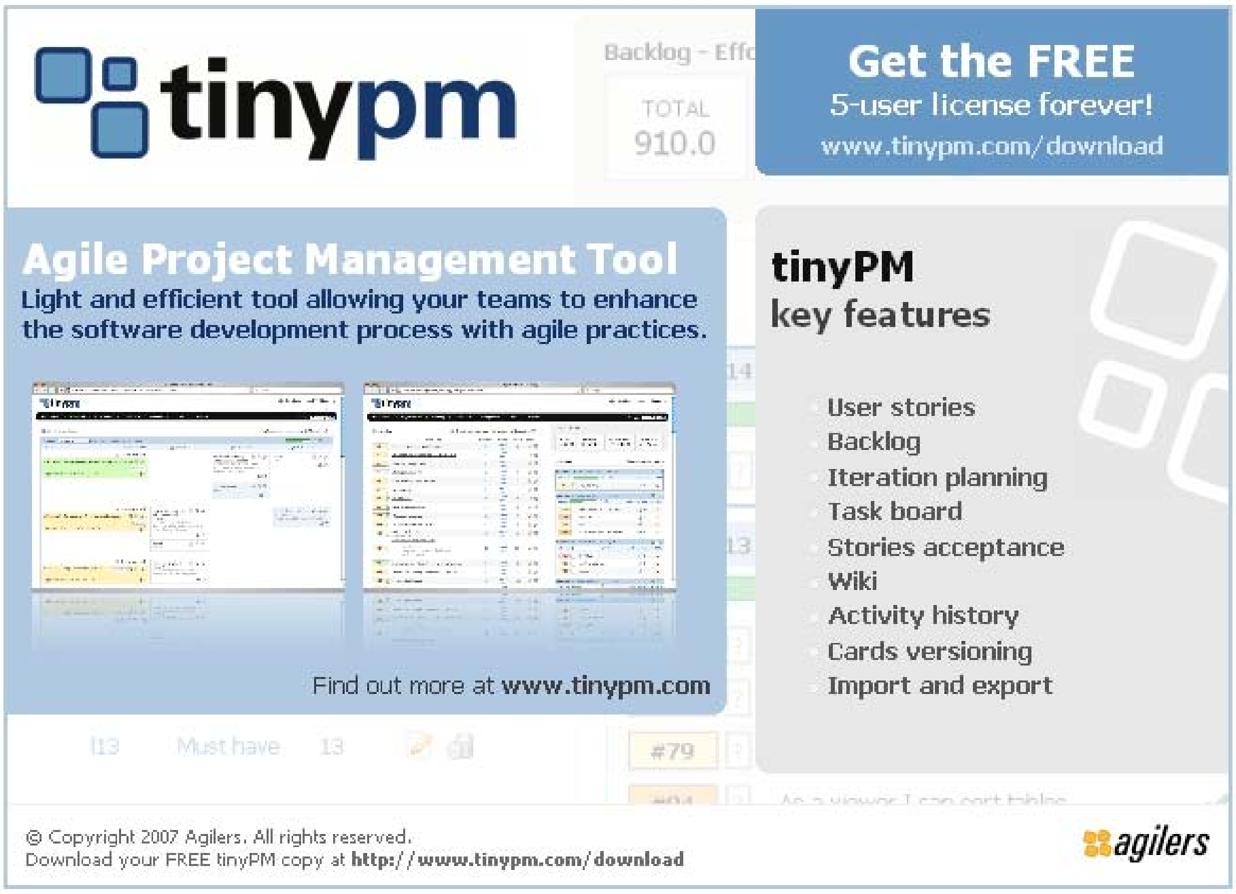
- PM: “Can you give me an estimate of the time necessary to develop feature xyz?”
- Programmer: “One month”
- PM: “That's far too long, we've got only one week!”
- Programmer: “I need at least three”
- PM: “I can give you two at most”
- Programmer: “Deal!”

The conversation above is an example of the “guess the number I'm thinking of” game [You]: the programmer, at the end, comes up with an “estimate” that matches what is acceptable for the manager, but, since it is *his* estimate, the manager will hold him accountable for that. Something is obviously wrong, but what is it exactly?

In order to answer this question, let me introduce three definitions - estimate, target, and commitment.

An **estimate** - as defined in “The New Oxford Dictionary of English” - is “an approximate calculation or judgement of the value, number, quantity, or extent of something”.

Advertisement - Agile Project Management Tool - Click on ad to reach advertiser web site


tinypm
Get the FREE
5-user license forever!
www.tinypm.com/download
Agile Project Management Tool
Light and efficient tool allowing your teams to enhance the software development process with agile practices.
Find out more at www.tinypm.com
tinyPM
key features
User stories
Backlog
Iteration planning
Task board
Stories acceptance
Wiki
Activity history
Cards versioning
Import and export
© Copyright 2007 Agilers. All rights reserved.
Download your FREE tinyPM copy at http://www.tinypm.com/download
agilers

In general we make estimates because we cannot directly measure the value of the quantity because [Stu]:

1. The object is inaccessible (e.g., continuous operational use)
2. The object does not exist yet (e.g., a new software product)
3. Measurement would be too expensive or dangerous

A very important thing to notice is that the definition above implies that an estimate is an objective measure. If I asked you to estimate my height, you would probably have a look at me and come up with a number irrespective of the fact that I wish to be taller than what I actually am - if I told you that your estimate had to be at least 2 metres, you'd probably laugh at me (I'm a bit more that 1.7 metres tall), yet, this is exactly what the manager in the conversation above was doing.

Two examples of estimates from a real project setting could be:

- Implementing the search functionality will require between two and four days
- The development costs will be between forty and sixty million dollars

A **target** is a statement of a desirable business objective. Some examples of targets are:

- Release version 1.0 by Christmas
- The system must support at least 400 concurrent users
- The cost must not exceed three million pounds

A **commitment** is a promise to deliver specified functionality at a certain level of quality by a certain date.

Some examples of commitments are:

- The search functionality will be available in the next release of the product
- The response time will improve by 30% by the end of the next iteration

Estimates, targets and commitments are independent from each other, but targets and commitments should be based on sound estimates, or, as written in [McC]:

“The primary purpose of software estimation is not to predict a project’s outcome; it is to determine whether a project’s targets are realistic enough to allow the project to be controlled to meet them”.

In other terms, the purpose of estimation is to make proper project management and planning possible, and allow the project stakeholders to make commitments based on realistic targets.

Back to the conversation at the beginning of this section. What the project manager was *really* asking the programmer was to make a commitment, not to provide an estimate. Even worse, the commitment was not based on a sound estimate, but only on an unstated target that the manager had in mind. However, it should be clear now that **estimates are not commitments, and they are not negotiable**.

Why does this kind of conversation happen then? In many cases there is a simple answer: more often than not, people are well meaning, but are (especially managers) under heavy pressure to deliver, and they ask questions hoping to receive the answers they are wishing for. On the other hand, the people providing the estimates (like the developer in the previous dialogue), sometimes find it easier to just give the answer that will keep the other party happy, and deal with the consequences later.

It doesn't need to be like that. Every time I find myself in a situation similar to the one above - with someone asking for a commitment and calling it estimate - the first thing I do is to make very clear the difference between the two. This always has the effect of changing the nature of the conversation into a more meaningful one. Now that we have some sound definitions, let's have a closer look at the estimation process itself.

What to estimate

When we talk about estimation, usually the first thing that comes to the mind is time. However, estimation is important for any factor that can impact the success of a project, e.g.:

- Time
- Cost
- Size
- Quality
- Effort
- Risk
- Etc.

Advertisement - MKS Integrity - Click on ad to reach advertiser web site

REALIZE THE VISION FOR UNIFIED ALM

MKS INTEGRITY™

Go Agile with Unsurpassed Collaboration and Flexible Process

Envision a solution with the flexibility to support both Agile and traditional processes.

Envision a team able to collaborate in real-time across the lifecycle.

Envision having full visibility into projects with burn-down charts and reports based on live data.

Turn Vision into Reality. Go Agile with MKS Integrity for Unified ALM.

www.mks.com/solutions/agile | 1.800.613.7535

MKS

Often, some of the factors listed above are actually constrained, i.e., they can vary only between a minimum and a maximum value (time and cost are some typical ones). If that's the case the other factors will have to be estimated accordingly, so that either all the constraints are respected, or, if not possible to achieve that, the project goals are modified, or the availability of some resources is increased.

Accuracy and precision

"It's better to be approximately right than precisely wrong". Warren Buffet.

When dealing with estimates, we necessarily deal with issues of accuracy and precision. In general, estimates should be as accurate as possible, but they can't be precise (after all they are approximate measurements). However, these two concepts are often used wrongly as synonyms - in fact a measurement can be accurate without being precise, or precise without being accurate.

In this context, accuracy refers to **how close** to the real value a number is, while precision refers to **how exact** a number is (number of decimal places). For example:

- Accurate. Task x will take between two and four days
- Precise. Task x will take 2.04 days

If task x above is finished in three days, then estimate 1 was accurate (but not precise), while estimate 2 was precise (but inaccurate, and wrong).

When estimating a quantity it is always important to match the precision to the accuracy of the estimate appropriately. For example, estimating in hours of work a project that is going to take a couple of years is likely to be a big mistake since the error is almost certainly going to be bigger than one hour - it is more likely to be a few months - and the high precision can actually be dangerous due to the false sense of confidence it conveys.

Uncertainty

Estimating software is hard for several reasons: big differences in individual productivity, the fact that creative processes are difficult to plan, the fact that software is intangible and so difficult to measure, and the fact that during the life of the project, anything can change - scope of the product, budget, deadlines, or, as often happen in the software world, requirements.

For these reasons, estimates have always a degree of uncertainty. Uncertainty is usually described and managed using probabilities - an estimate, typically, comes with a best, a worst, and a likely outcome with some probabilities attached. Single point estimates always have a probability less than 100% (usually closer to 0% than to 100%).

The best possible accuracy, according to studies conducted by Barry Boehm [Boe], can be described using the cone of uncertainty, which gives a measure of the estimation error depending on the development phase the project is in. In Figure 1 there is the cone for the case of sequential projects with no requirements volatility (they don't change during the lifetime of a project). The horizontal axis represents the phase the project is in, while the vertical one represents the error factor - e.g., in the initial product definition phase the estimate can be inaccurate by a factor of 4 on either side (over or under estimation). This means that if an initial time estimate for the project is 100 days, it could actually take as many as 400 (4x100) or as few as 25 (0.25x100).

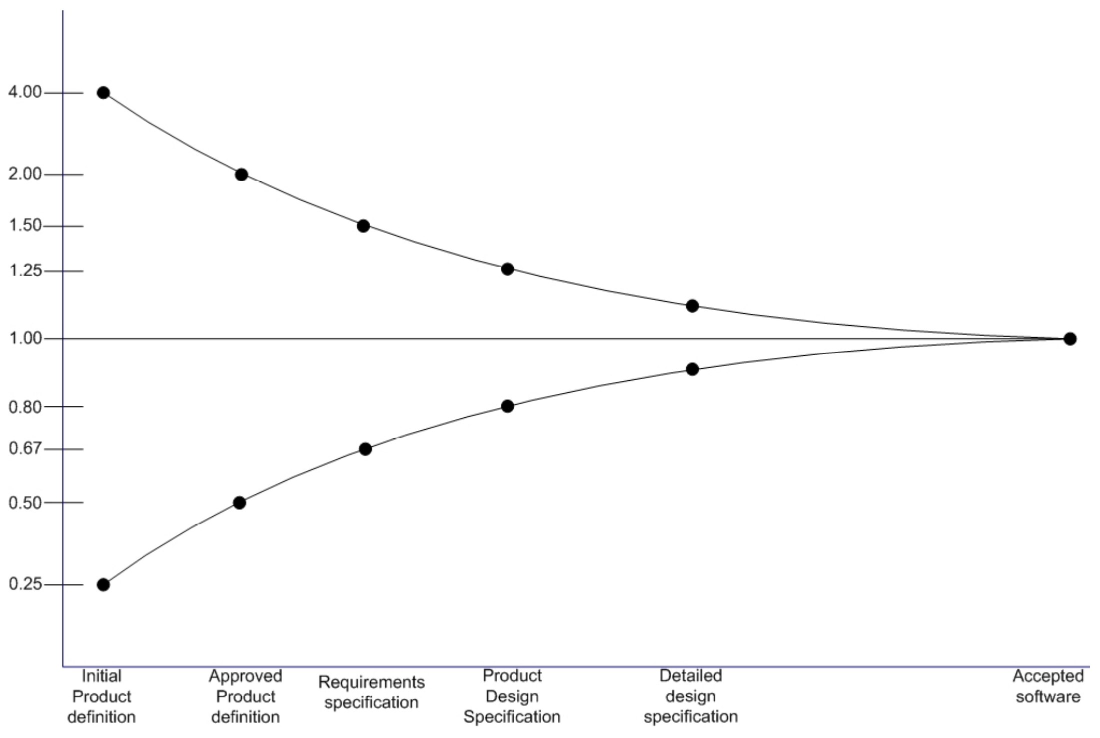


Figure 1: Cone of Uncertainty for a sequential process

As you can see, depending on the phase the software is in, the estimation error can be quite high - up to 400% over or under estimation (and in case of requirement volatility it can be even bigger), but, if the project is well managed, the accuracy increases over time.

The same cone can be used for iterative projects as well, as depicted in Figure 2. The main difference is that there are no phases any more but iterations, and the estimation error goes down iteration after iteration (again, if the project is well managed).

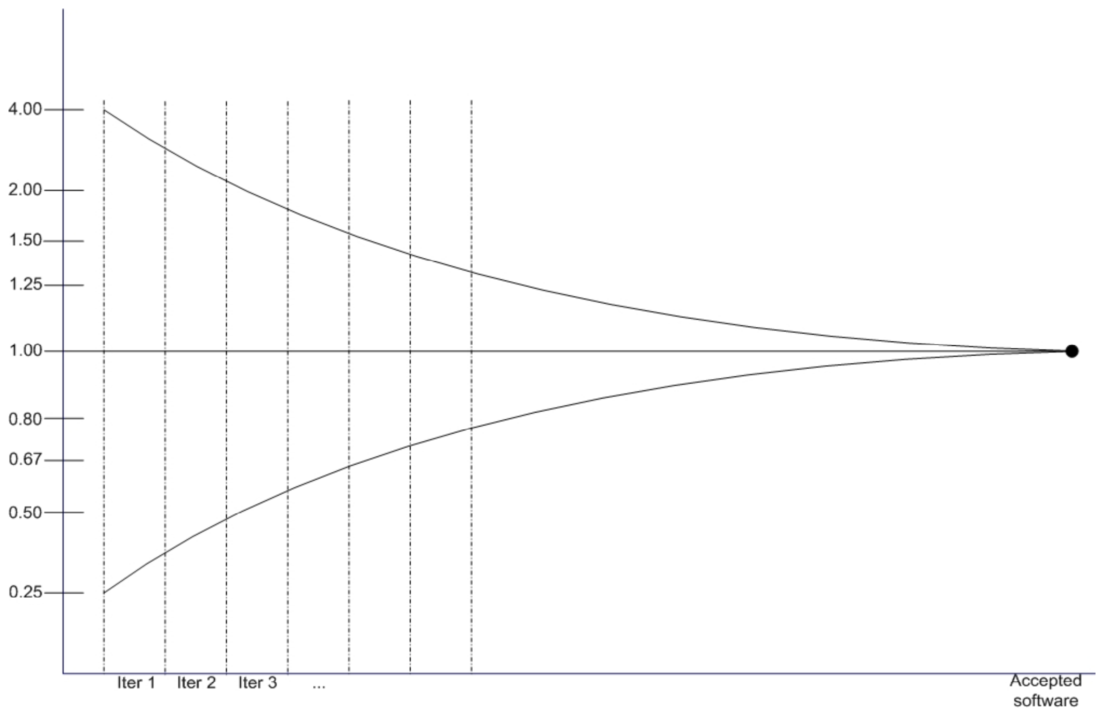


Figure 2: Cone of Uncertainty for an iterative process (Source: [Lar])

It is important to stress a few things.

First, the cone represents best case accuracy. Expert estimators give estimates somewhere inside the cone; less experienced ones can actually go outside the cone. This means that even an expert is bound to over estimate or under estimate a project by 400% during the initial phases of the project.

Second, the cone doesn't narrow itself. The fact that a project is at its 16th iteration doesn't necessarily mean that estimations will be more accurate. In fact, if any of the following is true

- Poor requirements definition
- Lack of user involvement
- Poor design
- Poor coding practices
- Bad planning
- Etc.

Estimates will always have a poor accuracy, and the project will have a high chance of being late and over budget, or, even worse, failing.

Third, estimation is an on-going activity: estimates should be updated whenever there is new information available. This implies that planning is an on-going activity as well. This is independent on the development process used - iterative, waterfall, or something in-between.

Some of you may be wondering why, with this variation, projects are often late and over-budget, and almost never early and under budget - after all the cone of uncertainty says that both things are possible. The answer is, usually, pressure to deliver: in an increasingly competitive world we want to do more in less time and fewer resources. On top of that, it might be difficult to explain to the boss that the project goals are far too ambitious given the available resources, and so people find it easier to say yes to “aggressive” targets.

Making commitments

At some point, even if estimates are only approximations, there will be some decisions to take and commitments to be made. So, when is the best time to commit? As you can see from Figure 3, in the case of a sequential process, a good time is sometime between the completed product definition and the design specification depending on the amount of risk that the stakeholders are willing to take (and the corresponding potential benefits).

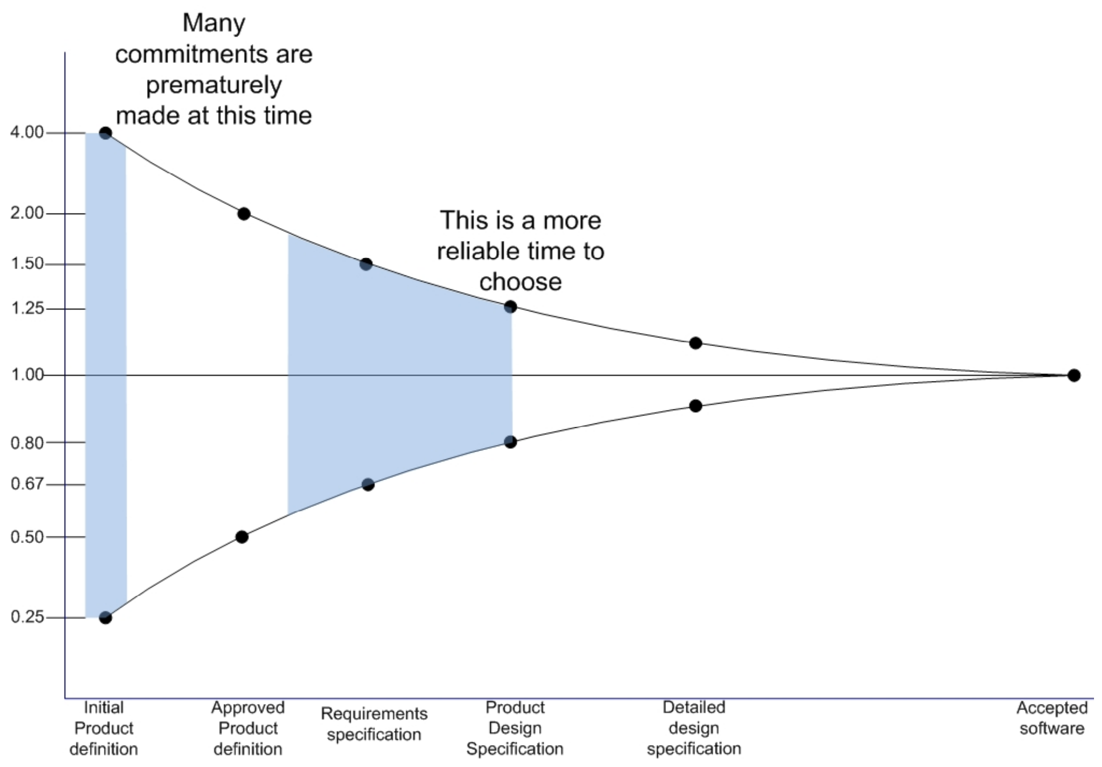


Figure 3: When to commit for a sequential process (adapted from [Lar])

The equivalent example for an iterative process is shown in Figure 4. The main difference is that, in this case, the commitment is made after a few iterations when the stakeholders consider the estimation error acceptable.

In both cases - of the sequential process, and of the iterative one - the initial period of product definition (or the first few iterations) can be used to prototype the most important parts of the product to get a better understanding of what to expect, and to be able to give more accurate estimates, and set more realistic targets and commitments.

Unfortunately, in many cases commitments are made very early when only very few things are known about the product and the potential issues. This is the main reason for time and cost overruns and also one of the biggest source of defects in the final product - when the deadline gets closer people are prone to cut corners and lower quality in the hope to meet the deadline.

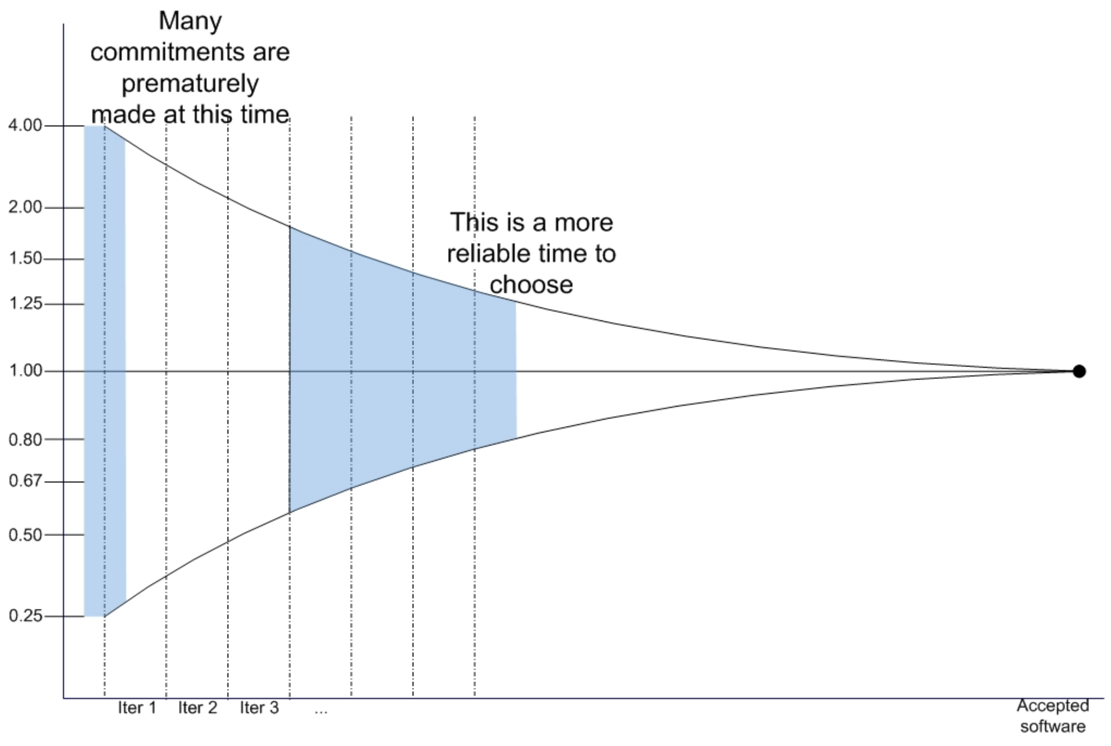


Figure 4: When to commit for an iterative process (adapted from [Lar])

Underestimation and overestimation

Accurate estimates are rare. Is it better to over-estimate or under-estimate?

Both approaches have problems.

Under-estimation tends to reduce the effectiveness of planning, reduce the chance of on-time delivery (developers already tend to be 20%-30% too optimistic), increase the chance of cutting corners, and cause destructive project dynamics which, in turn, will cause overtime, angry customers, and poor quality.

On the other hand, over-estimation tends to cause the cause problems due to Parkinson’s law which says that work will expand to fill available time, and to the student’s syndrome (adapted to software development) - if developers are given too much time, they’ll procrastinate until late in the project, and probably they won’t finish on time.

However, as shown in figure 5, over-estimation tends to cause fewer problems than under-estimation. In fact, the penalty paid for overestimation tends to increase linearly, while the one paid for underestimation tends to increase exponentially.

This makes sense also at an intuitive level. For example, when the time for developing the product is underestimated, as the deadline approaches the pressure mounts, the team starts to work over-time and cut corners, but this causes the occurrence of many defects which are very difficult to fix because corners have been cut, and the developers are tired because of the overtime, and due to this even more problems pop up, and so on and so forth - all the issues impact each-other causing an exponential growth in cost effort and schedule, and an exponential loss in quality. In case of overestimation, even if something bad happens, there is usually time to recover, and there are fewer knock-on effects.

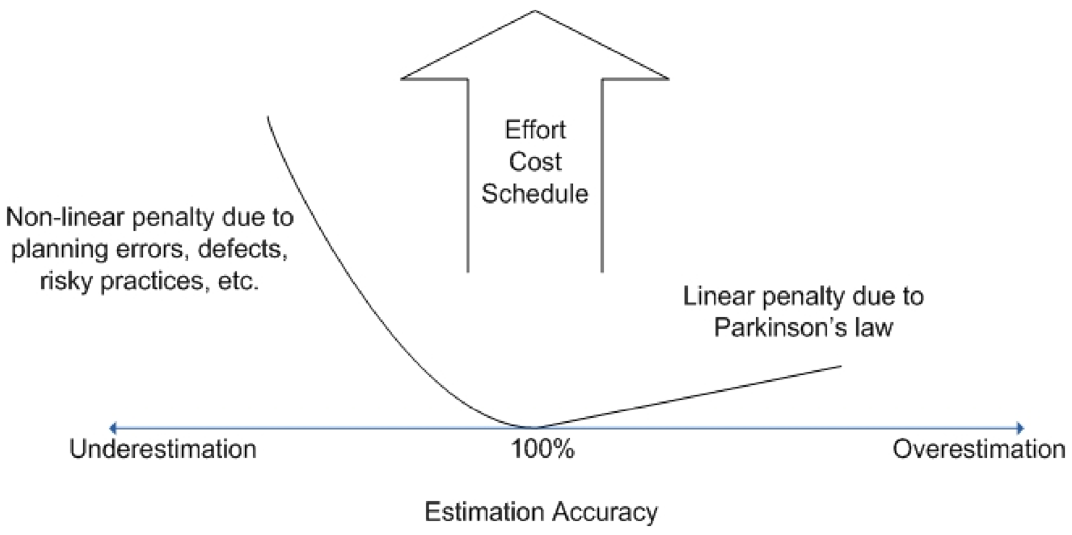

Effort
Cost
Schedule
Non-linear penalty due to planning errors, defects, risky practices, etc.
Linear penalty due to Parkinson's law
Underestimation
100%
Overestimation
Estimation Accuracy

Figure 5: Under-estimation and over-estimation effects (taken from [McC]

Some estimation techniques

Some common estimation techniques include

- Count, compute, and judge. If you can, count, if you can't count compute and use judgement only as a last resort. It can be used during all stages and can have a very high accuracy
- Mathematical models. Based on formulas obtained by studying the data available from past projects, their main advantage is the availability of software tools that can be used for predicting as well as simulating outcomes. Their big disadvantage is that they tend to depend on a very high number of calibration parameters which can be very tricky to configure appropriately. Probably the most famous one is COCOMO II by Barry Boehm [Boe]
- Calibration and historical data. Used to convert counts to estimates - story points to days, lines of code to effort, etc. Calibration uses three kinds of data (depending on what is available): industry data, which refers to data from other companies that develop the same kind of software; historical data, which is data available from the organization running the project; or project data, which is data from the project itself. The availability of historical and project data allows the creation of highly accurate estimates. The main disadvantage is that there has to be some data available to start with
- Analogy. This one is about creating estimates based on a similar (in terms of what is being estimated, e.g., size, complexity, etc.) past projects. Its accuracy is not very high, but it is quite simple to implement. Its main drawback is that it can be highly subjective - different estimators may have different ideas about how similar the projects involved are
- Proxy. Sometimes it is extremely difficult to produce direct estimates, but it is possible to find a proxy correlated to the quantity being estimated which is easier to deal with. A typical example is story points as a proxy for size. This technique can achieve great accuracy, but, to be used effectively, it is necessary to have a fair amount of experience and high discipline
- Expert judgement. This technique is based on the on the judgement of experts - e.g., senior developers who worked on similar systems. It is by far the most widely used, and it comes in various flavours, from individual to group techniques. It can have high accuracy, but it has some important disadvantages - finding the experts may not be easy, and since they are human, they may be subject to external pressure to cut their estimates.

The list above is by no means exhaustive. There are many more listed and explained in much more depth in [McC] and [Stu].

Each technique has its pros and cons. Often, but not always, there is a tension between simplicity, cost, and accuracy - the simpler the technique is to apply the lower are its cost and also its accuracy. Also, the accuracy of each technique, usually, depends on when it is used during the project - early stages, in the middle, or at the end. For these reasons, different techniques can (and should) be used together in order to produce the best results.

A typical approach is to use expert judgement - or, even better, some historical data - at the beginning of the project, and then collect data directly from the project to be used to calibrate and refine the estimates using any of the techniques described above as appropriate. When the project is done, the data collected can be used for estimating future projects in the company, improving the accuracy of the initial estimates and the overall ability of the company to deliver.

It is worth noting that, independently on how you start, the better way to act to improve the accuracy of your estimates is to collect project data as you go and use it to calibrate them. An analogy used by some agile teams to describe this approach is “yesterday's weather” from the observation that today's weather matches yesterday's weather more often than not. In fact, this is a much better approach than the, unfortunately, very common one based on wishful thinking - the we’ll do it better this time syndrome - which assumes that things will go better than in the past only to realize that is not the case when it is already too late.

Final remarks

There are a few very important things which are independent on the estimation techniques used, and are worth remembering when producing estimates.

First of all the quantities being estimated have to bee measurable ones, otherwise the estimation might be impossible. For example how would you estimate the impact on the schedule (or cost, or effort, etc.) of the following requirement (which, incidentally, is not fictional, but comes from a real specification of a system I worked on)?

“The system must be scalable to support future business growth”.

A better one would have been

“The system must be able to support 400 concurrent users with a maximum response time of 1 second”.

Second. The estimates should be provided by the people that have to do the work since they are the ones that are in the best position to know the effort, time, etc. required.

Third. Remember to include everything. For example, time estimates produced by developers should include all technical tasks that should be performed (build environment, computer set-up, supporting the build, etc.); all side tasks like e-mails, meetings, phone calls; time-off due to holidays, possible illness, etc.

Finally, all underlying assumptions should be explicit - availability of resources, expected quality, etc. Even programmers, believe it or not, are able to provide accurate estimates when they remember to include all the things listed above [McC].

Conclusion

In this article I described some of the basics of software estimation.

The most important points to remember are:

- Estimates, targets and commitments are different things
- Estimates are not negotiable
- Estimates cannot be precise, and always have a degree of uncertainty attached
- Estimation is an on-going activity, and new estimates must be made as soon as there is new information available
- There are several estimation techniques, each one with its pros and cons

Of course there is much more to know about this subject - For example, I deliberately avoided discussing all the human issues that can have an impact in the estimation process - and I strongly suggest all the books in the references, in particular [McC] and [Coh] are probably the best ones to get started with.

References

[Boe]: Boehm, Barry W. et al., Software Cost Estimation with Cocomo II, Prentice Hall, 2000

[Coh]: Cohn, Mike, Agile Estimating and Planning, Prentice Hall, 2005

[Lar]: Larman, Craig, Agile & Iterative Development: A Manager's Guide, Addison Wesley, 2004

[McC]: McConnell, Steve, Software Estimation: Demystifying the Black Art, Microsoft Press, 2006

[Stu]: Stutzke, Richard D., Estimating Software Intensive Systems, Addison Wesley, 2005

[You]: Yourdon, Edward, Death March, Prentice Hall, 2004

Behavior Driven Database Development (BDDD)

Pramodkumar J Sadalage, http://www.sadalage.com/
http://www.databaserefactoring.com/

When Behavior Driven Development (BDD) [1] was introduced, some of the key principles were

- Requirements are behavior,
- Provides “ubiquitous language” for analysis,
- Acceptance criteria should be executable.
- Design constraints should be made into executable tests.

All of these principles can be applied to database development. When interacting with the database, we tend to assume certain behavior of the database. Some of this is universal, like when a row is inserted in a table, the same row can later be retrieved. There are other behaviors of the database on every project that are not that universal, like Person table should have at least firstname or lastname. This behavior changes based on the functionality being developed and thus needs to be properly specified and executed. The database lends itself very well to the new way of thinking in the BDD space, where the behavior of the objects is considered. BDD is similar to describing requirements in code.

I am going show how the BDD technique can be applied to database development and how this technique can be used to develop and design databases in an iterative and incremental way.

While designing database objects, we are expecting these objects to behave in a certain fashion and we tend to rely on this behavior. Let's say we make a column NOT NULLABLE. We assume that the database server will throw an exception when a NULL value in inserted in this column. Making the column NULLABLE can alter this behavior of the database. When the behavior of the database is changed this way, all the assumptions that the application made about the database NOT allowing NULL values in the column are no longer true. To avoid these kinds of mistake, we can test the database behavior to assert that the database does throw an exception when NULL values are put in column. In this article I will talk about

- Design a Table
- Design a Primary Key
- Design a Not Null Column
- Design a Constraint on a Column
- Design a Foreign Key Constraint
- Design a Sequence for Object ID
- Design a Unique Index

The example domain being discussed here is Movie rental store. The store wants to track what DVD’s they have and who has rented the DVD’s. For simplicity sake lets assume we are using Java, Hibernate, Oracle and JUnit as some of the technologies. You can substitute these technologies with any others you like.

Design a Table

When starting to work on the feature “As a Store Manager I should be able to select a DVD to rent”. The attributes of the movie Object and movie table have to be decided first. Once you decide the attributes, these attributes need to be mapped in Hibernate mappings. Start with a test first that tries to create a domain object, save the domain object and fetch the domain object back from the database. Let's see the test

```
@Test
public void ShouldBeCreatedAndSavedSuccessfully() {
      Movie movie = new Movie();
      movie.setName(name);
      saveDomainObject(movie);
      assertNotNull("Insert failed", movie);
}
```

The above test, just verifies that it can create a Movie domain object and then save it in the database, the behavior we are driving out here is that a valid Movie domain object can be saved. At this stage the database script for Movie table looks like

```
CREATE TABLE MOVIE (
      MOVIEID NUMBER(18),
      NAME VARCHAR2(64)
);
```

Design a Primary Key

The next task we want do is to make sure the MOVIE table has a Primary Key and there is a value assigned to the primary key when the MOVIE domain object is saved. We can also check the Hibernate mapping behavior to make sure that the correct SEQUENCE is used to assign the next ID to the MOVIE object. The test looks like.

```
@Test
public void shouldAssignPrimaryKeyValuesFromSequence() {
Movie movie = createAndSave("PKASSIGNED");
Long currentPKValue = getCurrentValueForSequence("S_MOVIE");
assertNotNull("Movie should have been assigned a ID", movie.getId());
assertEquals("ID should have been
      same",movie.getId(),currentPKValue);
}
```

The above test “shouldAssignPrimaryKeyValuesFromSequence” checks the behavior of the Hibernate mapping, the Primary Key constraint and makes sure that the correct sequence S_MOVIE is used to populate the ID value for the Movie object. At this stage the database script for the Movie table and S_MOVIE sequence looks like

```
CREATE TABLE MOVIE (
     MOVIEID NUMBER(18),
     NAME  VARCHAR2(64),
     CONSTRAINT PK_MOVIE
     PRIMARY KEY (MOVIEID)
);

CREATE SEQUENCE S_MOVIE;
```

Design a Not Null Column

The next feature to work on is “As a Internal User I should be able to assign a year the movie was made”. Lets also say one of the requirements is to make sure that every movie has to have the year it was made. We will use Make Column Non Nullable [2] database refactoring on the database. Starting with the test as shown below.

```
@Test
public void shouldNotAllowNullYear() {
     Movie movie = new Movie();
     movie.setName(name);
     try {
     saveDomainObject(movie);
     fail();
     } catch (ConstraintViolationException e) {
     assertContains(e,"ORA-01400: cannot insert NULL");
     }
}
```

As can be seen in the above example, the behavior of the database to throw an exception when one of the rules set on the table is not satisfied. If this assumption “That the database does not allow NULL values in the year column” is enforced by the test “ShouldNotAllowNullYear”, when this assumption on the database is changed this test will fail. During refactoring of the database, these tests help to enforce the assumptions made on the database. After this stage the database table looks like

```
CREATE TABLE MOVIE (
     MOVIEID NUMBER(18),
     NAME VARCHAR2(64),
     YEAR VARCHAR2(4) NOT NULL,
     CONSTRAINT PK_MOVIE
     PRIMARY KEY (MOVIEID)
);
```

Design a Constraint on a Column

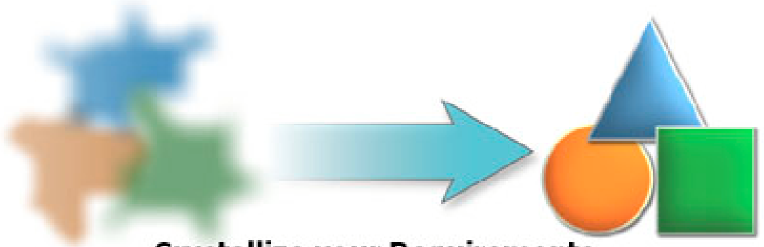
The next feature to work on is “The store does not carry any movies made before 1999”. Lets start with Introduce Column Constraint [3] database refactoring to create a column level constraint on the YEAR column so that it does not allow any value less than 1999.

```
@Test
public void shouldNotAllowMovieYearBeforeYear1999() {
try {
      createAndSave ("1998", "YearBefore");
      fail("Movie year is before 1999");
      } catch (Exception e) {
      assertContains(e,"CHK_MOVIEYEAR_GT_1998");
}
}
```

The behavior of the database to disallow values before 1999 is asserted by the test "ShouldNotAllowMovieYearBeforeYear1999". At this stage the database script looks like

```
CREATE TABLE MOVIE (
      MOVIEID NUMBER(18),
      NAME VARCHAR2(64),
      YEAR VARCHAR2(4) NOT NULL,
      CONSTRAINT PK_MOVIE
      PRIMARY KEY (MOVIEID),
      CONSTRAINT CHK_MOVIEYEAR_GT_1998
      CHECK( YEAR > '1998')
);
```

Advertisement - Visual Use Case - Click on ad to reach advertiser web site



Document Requirements precisely with "Structured Pseudo-Code"

...and then with a single click...

...convert it into a "Flow Chart" that everyone can understand!

Visual Use Case™ 2006

Write better Use Cases in less time. Multiply Productivity, Improve Software.

Click here to download your instant free trial of Visual Use Case.
http://www.TechnoSolutions.com

Design a Foreign Key Constraint

The next feature to work on is “Movie has details about itself that need to persisted”. To store the details about the movie we will Introduce New Table [4] MovieDetail with a MovieId on the MovieDetail table.

So effectively we will have a collection of MovieDetail objects on the Movie object. Having a MovieDetail object created without the MovieId on would create dirty data in the MovieDetail table, so we will Add Foreign Key Constraint [5] on the MoveDetail table. Having a NULL MoveId would also invalidate the MoveDetail object, since a MovieDetail cannot exist without Movie, making the MoveDetail.MovieID not null.

This assertion can be done by the domain object, but I have seen over years of consulting at many different companies that the database gets used eventually without the domain layer (Reporting, Data Extract, Data Import etc.), so it’s a better to move these kinds of constraints on the database. The following test will ensure that the database behavior matches what we expected.

```
@Test
public void shouldNotAllowMovieDetailsToExistWithOutMovie() {
      Movie movie = createAndSave("DoomsDay");
      MovieDetail detail = new MovieDetail();
      detail.setDescription("DoomsDayMovie");
      detail.setMovie(movie);
      detail.setUrl("www.doomsday.com");
      saveDomainObject(detail);
      try {
      removeDomainObject(movie);
      fail();
      }

      catch (Exception e) {
      assertContains(e, "FK_MOVIEDETAIL_MOVIE");
      }
}
```

The above test tries to delete the Movie object from the database when it has the MoveDetail as its child, this forces the database to raise a Foreign Key violation, which is verified by the test, similarly we can write a test where the MoveDetail.MovieId is null.

```
@Test
public void shouldNotAllowMovieDetailsToExistWithNullMovie() {
      MovieDetail detail = new MovieDetail();
      detail.setDescription("DoomsDayMovie");
      detail.setMovie(null);
      detail.setUrl("www.doomsday.com");
      try {
      saveDomainObject(detail);
      fail();
      } catch (Exception e) {
      assertContains(e, "cannot insert NULL into");
      }
}
```

At this stage the database script looks like

```
CREATE TABLE MOVIEDETAIL (
      MOVIEDETAILID NUMBER(18),
      MOVIEID NUMBER(18) NOT NULL,
      DESCRIPTION VARCHAR2(4000),
      URL VARCHAR2(400),
      CONSTRAINT PK_MOVIEDETAIL
      PRIMARY KEY (MOVIEDETAILID)
);

ALTER TABLE MOVIEDETAIL
      ADD CONSTRAINT FK_MOVIEDETAIL_MOVIE
      FOREIGN KEY (MOVIEID)
      REFERENCES MOVIE;
```

Design a Sequence for ObjectId

The next feature to work on is a Technical Story to reduce database trips to get the ObjectID for every new Object created. To reduce the number of database trips for every object created, we can create a sequence (Oracle database specific), which returns values in Increments of 1000. Then we can make the application return one ID at a time from the value that was returned by the database, reducing the number of round trips to the database from the application. When the application exhausts the 1000 IDs it asks the database for the next 1000. Which means that we are relying on the database behavior to return values in increments of 1000. The following test will ensure that the database behavior matches what we expected.

```
@Test
public void shouldIncrementIdBy1000() {
      Long firstValue = getNextValueForSequence("S_MOVIE");
      Long secondValue = getNextValueForSequence("S_MOVIE");
      assertIncrementsBy1000(firstValue, secondValue);
}
```

The above test tries to get the next value from the “S_MOVIE” sequence and compare if the consecutive values returned have incremented by 1000. At this stage the database script looks like

```
CREATE SEQUENCE S_MOVIE
      START WITH 1
      INCREMENT BY 1000;
```

Design a Unique Index

The next feature to work on is “The system should not allow duplicate Movie.Name in the system”, to implement this feature we use the Introduce Index [6] refactoring, the behavior of the database ensures that the Movie.Name cannot be duplicated.

```
@Test
public void shouldNotAllowDuplicateNames() throws Exception {
      Movie movie = createAndSave("NODUPE");
      try {
      Movie dupe = createAndSave("NODUPE");
      fail("Duplicate Movie name is allowed");
      }
      catch (ConstraintViolationException e) {
```

```
      assertContains(e, "UIDX_MOVIE_NAME");
      }
}
```

The above test tries to persist the Movie object with duplicate names and expects that the database throws a Unique Index violation. At this stage the database script looks like

```
CREATE UNIQUE INDEX  UIDX_MOVIE_NAME
ON MOVIE (NAME);
```

Conclusion

These behavior specifications in code make sure that the database provides the specified behavior for the application and that the database cannot be changed inadvertently. These kinds of tests are also important if there is a need to have multiple database compatibility in your application code base. All the example code above can be downloaded from http://www.sadalage.com/bdddexample.zip

References

[1] Introduction to BDD, Dan North
http://dannorth.net/introducing-bdd

Refactoring Databases: Evolutionary Database Design
[2] http://databaserefactoring.com/MakeColumnNonNullable.html
[3] http://databaserefactoring.com/IntroduceColumnConstraint.html
[4] http://databaserefactoring.com/IntroduceNewTable.html
[5] http://databaserefactoring.com/AddForeignKey.html
[6] http://databaserefactoring.com/IntroduceIndex.html

Optimizing the Contribution of Testing to Project Success

Niels Malotraux, niels @ malotaux.nl
http://www.malotaux.nl/nrm/English/

1. Introduction

We know all the stories about failed and partly failed projects, only about one third of the projects delivering according to their original goal [1].

Apparently, despite all the efforts for doing a good job, too many defects are generated by developers, and too many remain undiscovered by testers, causing still too many problems to be experienced by users. It seems that people are taking this state of affairs for granted, accepting it as a nature of software development. A solution is mostly sought in technical means like process descriptions, metrics and tools. If this really would have helped, it should have shown by now.

Oddly enough, there is a lot of knowledge about how to significantly reduce the generation and proliferation of defects and deliver the right solution quicker. Still, this knowledge is ignored in the practice of many software development organizations. In papers and in actual projects I've observed that the time spent on testing and repairing (some people call this *debugging*) is quoted as being 30 to 80% of the total project time. That's a large budget and provides excellent room for a lot of savings.

In 2004, I published a booklet: *How Quality is Assured by Evolutionary Methods* [2], describing practical implementation details of how to organize projects using this knowledge, making the project a success. In an earlier booklet: *Evolutionary Project Management Methods* [3], I described issues to be solved with these methods and my first practical experiences with the approach. Tom Gilb published already in 1988 about these methods [4].

In this booklet we'll extend the Evo methods to the testing process, in order to optimize the contribution of testing to project success.

Important ingredients for success are: a change in attitude, taking the Goal seriously, which includes working towards defect-free results, focusing on prevention rather than repair, and constantly learning how to do things better.

2. The Goal

Let's define as the main goal of our software development efforts:

> *Providing the customer with what he needs, at the time he needs it,*
> *to be satisfied, and to be more successful than he was without it …*

If the customer is not satisfied, he may not want to pay for our efforts. If he is not successful, he *cannot* pay. If he is not more successful than he already was, why should he invest in our work anyway? Of course we have to add that what we do in a project is:

> *… constrained by what the customer can afford and what we mutually*
> *beneficially and satisfactorily can deliver in a reasonable period of time.*

Furthermore, let's define a Defect as:

> *The cause of a problem experienced by the stakeholders of the system.*

If there are no defects, we'll have achieved our goal. If there are defects, we failed.

3. The knowledge

Important ingredients for significantly reducing the generation and proliferation of defects and delivering the right solution quicker are:

- **Clear Goal**: If we have a clear goal for our project, we can focus on achieving that goal. If management does not set the clear goal, we should set the goal ourselves.
- **Prevention attitude**: Preventing defects is more effective and efficient than injecting-finding-fixing, although it needs a specific attitude that usually doesn't come naturally.
- **Continuous Learning**: If we organize projects in very short Plan-Do-Check-Act (PDCA) cycles, constantly selecting only the most important things to work on, we will most quickly learn what the real requirements are and how we can most effectively and efficiently realize these requirements. We spot problems quicker, allowing us more time to do something about them. Actively learning is sped up by expressly applying the Check and Act phases of PDCA.
- **Evolutionary Project Management** (Evo for short) uses this knowledge to the full, combining Project-, Requirements- and Risk-Management into Result Management. The essence of Evo is actively, deliberately, rapidly and frequently going through the PDCA cycle, for the product, the project *and* the process, constantly reprioritizing the order of what we do based on Return on Investment (ROI), and highest value first. In my experience as project manager and as project coach, I observed that those projects, who seriously apply the Evo approach, are routinely successful on time, or earlier [5].

Evo is not only iterative (using multiple cycles) and incremental (we break the work into small parts), like many similar Agile approaches, but above all Evo is about learning. We proactively anticipate problems before they occur and work to prevent them. We may not be able to prevent all the problems, but if we prevent most of them, we have a lot more time to cope with the few problems that slip through.

4. Something is not right

Satisfying the customer and making him more successful implies that the software we deliver should show no defects. So, all we have to do is delivering a result with no defects. As long as a lot of software is delivered with defects and late (which I consider a defect as well), apparently something is not right.

Customers are also to blame, because they keep paying when the software is not delivered as agreed. If they would refuse to pay, the problem could have been solved long ago. One problem here is that it often is not obvious what was agreed. However, as this is a *known problem*, there is no excuse if this problem is not solved within the project, well before the end of the project.

5. The problem with bugs

In a conventional software development process, people develop a lot of software with a lot of defects, which some people call *bugs*, and then enter the *debugging* phase: testers testing the software and developers repairing the *bugs*.

Bugs are so important that they are even counted. We keep a database of the number of bugs we found in previous projects to know how many bugs we should expect in the next project.

Software without bugs is even considered suspect. As long as we put bugs in the center of the testing focus, there will be bugs. Bugs are normal. They are needed. What should we do if there were no bugs any more?

This way, we *endorse* the injection of bugs. But, does this have anything to do with our goal: making sure that the customer will not encounter any problem?

Personally, I dislike the word bug. To me, it refers to a little creature creeping into the software, causing trouble beyond our control. In reality, however, people make mistakes and thus cause defects. Using the word *bug*, subconsciously defers responsibility for making the mistake. In order to prevent defects, however, we have to actively take responsibility for our mistakes.

6. Defects found are symptoms

Many defects are symptoms of deeper lying problems. Defect prevention seeks to find and analyze these problems and doing something more fundamental about them.

Simply repairing the apparent defects has several drawbacks:

- Repair is usually done under pressure, so there is a high risk of imperfect repair, with unexpected side effects.
- Once a bandage has covered up the defect, we think the problem is solved and we easily forget to address the real cause. That's a reason why so many defects are still being repeated.
- Once we find the underlying real cause, of which the defect is just a symptom, we'll probably do a more thorough redesign, making the repair of the apparent defect redundant.

As prevention is better than cure, let's move from *fixation-to-fix* to *attention-to-prevention*.

Many mistakes have a repetitive character, because they are a product of certain behavior of people. If we don't deal with the root causes, we will keep making the same mistakes over and over again. Without feedback, we won't even know. With quick feedback, we can put the repetition immediately to a halt.

7. Defects typically overlooked

We must not only test whether functions are correctly implemented as documented in the requirements, but also, a level higher, whether the requirements adequately solve the needs of the customer according to the goal. Typical defects that may be overlooked are:

- Functions that won't be used (superfluous requirements, no Return on Investment)
- Nice things (added by programmers, usefulness not checked, not required, not paid for)
- Missing quality levels (should have been in the requirements)
  e.g.: response time, security, maintainability, usability, learnability
- Missing constraints (should have been in the requirements)
- Unnecessary constraints (not required)

Another problem that may negatively affect our goal is that many software projects end at "Hurray, it works!". If our software is supposed to make the customer more successful, our responsibility goes further: we have to make sure that the increase in success is *going to happen*.

This awareness will stimulate our understanding of quality requirements like “learnability” and “usability”. Without it, these requirements don’t have much meaning for development. It’s a defect if success is not going to happen.

8. Is defect free software possible?

Most people think that defect free software is impossible. This is probably caused by lack of understanding about what defect free, or Zero Defects, really means. Think of it as an asymptote (Figure 1). We know that an asymptote never reaches its target. However, if we put the bar at an *acceptable level* of defects, we’ll asymptotically approach that level. If we put the bar at zero defects, we can asymptotically approach that level.

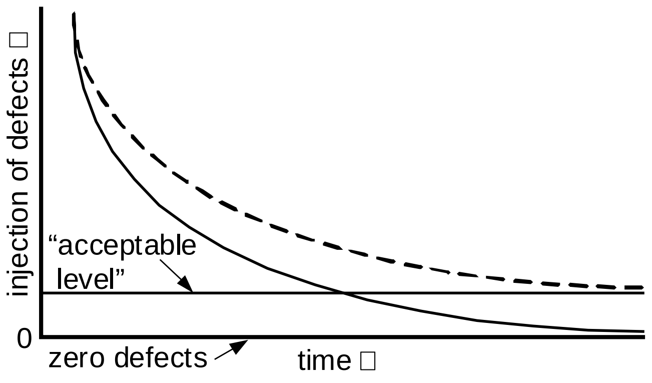


Figure 6: Zero Defects is an asymptote

Philip Crosby writes [6]:

> *Conventional wisdom says that error is inevitable. As long as the performance standard requires it, then this self-fulfilling prophecy will come true. Most people will say: People are humans and humans make mistakes. And people do make mistakes, particularly those who do not become upset when they happen. Do people have a built-in defect ratio? Mistakes are caused by two factors: lack of knowledge and lack of attention. Lack of attention is an attitude problem.*

When Crosby first started to apply Zero Defects as performance standard in 1961, the error rates dropped 40% almost immediately [6]. In my projects I’ve observed similar effects.

Experience: No defects in the first two weeks of use

A QA person of a large banking and insurance company I met in a SPIN metrics working group told me that they got a new manager who told them that from now on she expected that any software delivered to the (internal) users would run defect free for at least the first two weeks of use. He told me this as if it were a good joke. I replied that I thought he finally got a good manager, setting them a clear requirement: “No defects in the first two weeks of use.” Apparently this was a target they had never contemplated before, nor achieved. Now they could focus on how to achieve defect free software, instead of counting function points and defects. Remember that in bookkeeping being one cent off is already a capital offense,so defect free software should be a normal expectation for a bank. Why wouldn’t it be for *any* environment?

Zero Defects is a performance standard, set by management. In Evo projects, even if management does not provide us with this standard, we’ll assume it as a standard for the project, because we know that it will help us to conclude our project successfully in less time.

9. Attitude

As long as we are convinced that defect free software is impossible, we will keep producing defects, failing our goal. As long as we are accepting defects, we are endorsing defects. The more we talk about them, the more normal they seem. It's a self-fulfilling prophecy. It will perpetuate the problem. So, let’s challenge the defect-cult and do something about it.

From now on, we don’t want to make mistakes any more. We get upset if we make one. Feel the failure. If we don't feel failure, we don’t learn. Then we work to find a way not to make the mistake again. If a task is finished we don’t *hope* it’s ok, we don’t *think* it’s ok, no, we’ll be *sure* that there are no defects and we’ll be genuinely surprised when there proves to be any defect after all. In my experience, this attitude prevents half of the defects in the first place. Because we are humans, we can study how we operate psychologically and use this knowledge to our advantage. If we can prevent half of the defects overnight, then we have a lot of time for investing in more prevention, while still being more productive. This attitude is a crucial element of successful projects.

Experience: No more memory leaks

My first Evo project was a project where people had been working for months on software for a hand-held terminal. The developers were running in circles, adding functions they couldn’t even test, because the software crashed before they arrived at their newly added function. The project was already late and management was planning to kill the project. We got six weeks to save it.

The first goal was to get stable software. After all, adding any function if it crashes within a few minutes of operation is of little use: the product cannot be sold. I told the team to take away all functionality except one very basic function and then to make it stable. The planning was to get it stable in two weeks and only then to add more functionality gradually to get a useful product.

I still had other business to finish, so I returned to the project two weeks later. I asked the team “Is it stable?”. The answer was: “We found many memory leaks and solved them. Now it’s much *stabler*”. And they were already adding new functionality. I said: “Stop adding functionality. I want it stable, not *almost* stable”. One week later, all memory leaks were solved and stability was achieved. This was a bit of a weird experience for the team: the software didn’t crash any more. Actually, in this system there was not even a need for dynamically allocatable memory and the whole problem could have been avoided. But changing this architectural decision wasn’t a viable option at this stage any more.

Now that the system was stable, they started adding more functions. We got another six weeks to complete the product. I made it very clear that I didn’t want to see any more memory leaks. Actually that I didn’t want to see any defects. The result was that the testers suddenly found hardly any defect any more and from now on could check the correct functioning of the device. At the end of the second phase of six weeks, the project was successfully closed. The product manager was happy with the result.

Conclusion: after I made it clear that I didn’t want to see any defects, the team hardly produced any defects. The few defects found were easy to trace and repair. The change of attitude saved a lot of defects and a lot of time. The team could spend most of its time adding new functionality instead of fixing defects. This was Zero Defects at work. Technical knowledge was not the problem to these people: once challenged, they quickly came up with tooling to analyze the problem and solve it. The attitude was what made the difference.

10. Plan-Do-Check-Act

I assume the Plan-Do-Check-Act (PDCA- or Deming-) cycle [7] is well known (Figure 2, next page). Because it’s such a crucial ingredient, I’ll shortly reiterate the basic idea:

- We *Plan* what we want to accomplish and how we think to accomplish it best.
- We *Do* according to the plan.
- We *Check* to observe whether the result from the *Do* is according to then *Plan*.
- We *Act* on our findings. If the result was good: what can we do better. If the result was not so good: how can we make it better. *Act* produces a renewed strategy.

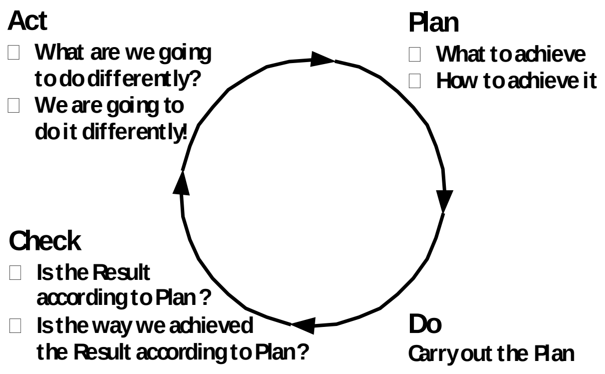

Act
What are we going to do differently?
We are going to do it differently!
Plan
What to achieve
How to achieve it
Check
Is the Result according to Plan ?
Is the way we achieved the Result according to Plan?
Do
Carry out the Plan

Figure 7: PDCA or Deming cycle

The key-ingredients are: planning before doing, systematically checking and above all *acting*: doing something differently. After all, if you don’t do things differently, you shouldn’t expect a change in result.

In Evo we constantly go through multiple PDCA cycles, deliberately adapting strategies in order to learn how to do things better all the time, actively and purposely speeding up the evolution of our knowledge.

As a driver for moving the evolution in the right direction, we use Return on Investment (ROI): the project invests time and other resources and this investment has to be regained in whatever way, otherwise it’s just a hobby. So, we’ll have to constantly be aware whether all our actions contribute to the value of the result. Anything that does not contribute value, we shouldn’t do.

Furthermore, in order to maximize the ROI, we have to do the most important things first. In practice, priorities change dynamically during the course of the project, so we constantly reprioritize, based on what we learnt so far. Every week we ask ourselves: “What are the most important things to do. We shouldn’t work on anything less important.” Note that priority is molded by many issues: customer issues, project issues, technical issues, people issues, political issues and many other issues.

11. How about Project Evaluations

Project Evaluations (also called Project Retrospectives, or Post-Mortems - as if all projects die) are based on the PDCA cycle as well. At the end of a project we evaluate what went wrong and what went right.

Doing this only at the end of a project has several drawbacks:

- We tend to forget what went wrong, especially if it was a long time ago.
- We put the results of the evaluation in a write-only memory: do we really remember to check the evaluation report at the very moment we need the analysis in the next project? Note that this is typically one full project duration after the fact.
- The evaluations are of no use for the project just finished and being evaluated.
- Because people feel these drawbacks, they tend to postpone or forget to evaluate. After all, they are already busy with the next project, after the delay of the previous project.

In short: the principle is good, but the implementation is not tuned to the human time-constant.

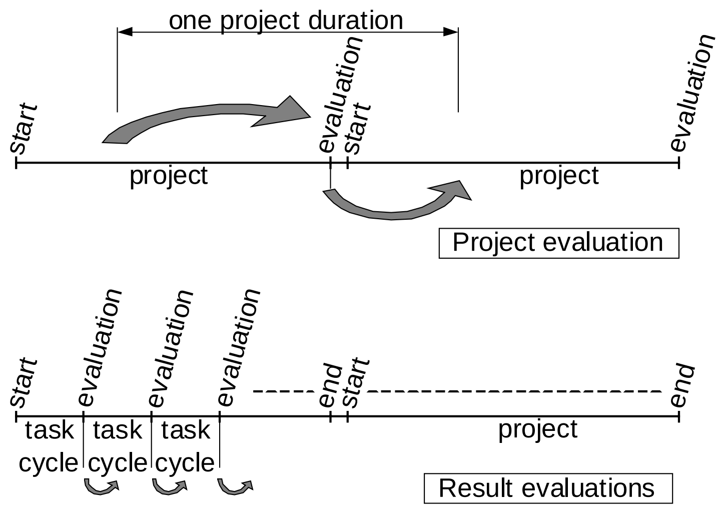


Figure 8: Project and Result evaluations

In Evo, we evaluate *weekly* (in reality it gradually becomes a way-of-life), using PDCA cycles, and now this starts to bear fruit (Figure 3):

- Not so much happens in one week, so there is not so much to evaluate.
- It’s more likely that we remember the issues of the past five days.
- Because we most likely will be working on the same kind of things during the following week, we can immediately use the new strategy, based on our analysis.
- One week later we can check whether our new strategy was better or not, and refine.
- Because we immediately apply the new strategy, it naturally is becoming our new way of working.
- The current project benefits immediately from what we found and improved.

So, evaluations are good, but they must be tuned to the right cycle time to make them really useful. The same applies to testing, as this is also a type of evaluation.

12. Current Evo Testing

Conventionally, a lot of testing is still executed in Waterfall mode, after the *Code Complete* milestone. I have difficulty understanding this “Code Complete”, while apparently the code is not complete, witness the planned “debugging” phase after this milestone. Evo projects do not need a separate debugging phase and hardly need repair after delivery. If code is complete, it is complete. Anything is only ready if it is completely done, *not to worry about it any more*. That includes: no defects. I know we are human and not perfect, but remember the importance of attitude: we *want* to be perfect[1].

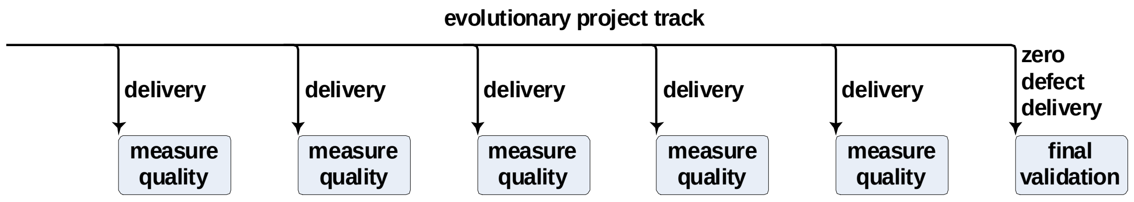


Figure 9: Testing of early deliveries helps the developers to get ready for zero-defect final delivery

Because we regularly deliver results, testers can test these intermediate results (Figure 4). They feed back their findings, which we will use for prevention or optimization. Most issues that are not caught by the testers (I suppose testers are human as well) may be found in subsequent deliveries. This way, most of any undiscovered defects will be caught before the final delivery and, more importantly, be exploited for prevention of further injection of similar defects. Because all people in the project aim for Zero Defects delivery, the developers and testers work together in their quest for perfection. Note that perfection means: *freedom from fault or defect*. It does *not* mean: *gold plating*

13. Further improvement

To further improve the results of the projects, we can extend the Evo techniques to the testing process and exploit the PDCA paradigm even further:

- Testers focus on a clear goal. Finding defects is not the goal. After all, we don’t want defects. Any defects found are only a means to achieve the real goal: the success of the project.
- Testers will select and use any method appropriate for optimum feedback to development, be it testing, review or inspection, or whatever more they come up with.
- Testers check work in progress *even before* it is delivered, to feedback issues found, allowing the developer to abstain from further producing these issues for the remainder of his work.
- “Can I check some piece of what you are working on now?” “But I’m not yet ready!” “Doesn’t matter. Give me what you have. I’ll tell you what I find, if I find anything”. Testers have a different view, seeing things the developer doesn’t see. Developers don’t naturally volunteer to have their intermediate work checked. Not because they don’t like it to be checked, but because their attention is elsewhere. Testers can help by asking. Initially the developers may seem a little surprised, but this will soon fade.
- Similarly, testers can solve a typical problem with planning reviews and inspections. Developers are not against reviews and inspections, because they very well understand the value. They have trouble, however, planning them in between of their design work, which consumes their attention more. If we include the testers in the process, the testers will recognize when which types of review, inspections or tests are needed and organize these accordingly. This is a natural part of their work helping the developers to minimize rework by minimizing the injection of defects and minimizing the time slipped defects stay in the system.
- In general: organizing testing the Evo way means entangling the testing process more intimately with the development process.

14. Cycles in Evo

In the Evo development process, we use several learning cycles:

- The TaskCycle [9] is used for organizing the work, optimizing estimation, planning and tracking. We constantly check whether we are doing the right things in the right order to the right level of detail. We optimize the work effectiveness and efficiency. TaskCycles never take more than one week.
- The DeliveryCycle [10] is used for optimizing the requirements and checking the assumptions. We constantly check whether we are moving to the right product results. DeliveryCycles focus the work organized in TaskCycles. DeliveryCycles normally take not more than two weeks.
- TimeLine [11] is used to keep control over the project duration. We optimize the order of DeliveryCycles in such a way that we approach the product result in the shortest time, with as little rework as possible.

During these cycles we are constantly optimizing:

- The product [12]: how to arrive at the best product (according to the goal).
- The project [13]: how to arrive at this product most effectively and efficiently.
- The process [14]: finding ways to do it even better. Learning from other methods and absorbing those methods that work better, shelving those methods that currently work less effectively.

If we do this well, by definition, there is no better way.

15. Evo cycles for testing

Extending Evo to testing adds cycles for feedback from testing to development, as well as cycles for organizing and optimizing the testing activities themselves.

- Testers organize their work in weekly, or even shorter TaskCycles.
- The DeliveryCycle of the testers is the Test-feedback cycle: in very short cycles testers take intermediate results from developers, check for defects in all varieties and feed back optimizing information to the developers, while the developers are still working on the same results. This way the developers can avoid injecting defects in the remainder of their work, while immediately checking out their prevention ideas in reality.
- The Testers use their own TimeLine, synchronized with the development TimeLine, to control that they plan the right things at the right time, in the right order, to the right level of detail during the course of the project and that they conclude their work in sync with development.

During these cycles the testers are constantly optimizing:

- The product: how to arrive at the most effective product.
- Remember that their product goal is: providing their customer, in this case the developers, with what they need, at the time they need it, to be satisfied, and to be more successful than they were without it.
- The project: how to arrive at this product most effectively and efficiently. This is optimizing in which order they should do which activities to arrive most efficiently at their result.

- The process: finding ways to do it better. Learning from other methods and absorbing those methods that work better, shelving those methods that currently work less effectively.

Testers are part of the project and participate in the weekly 3-step procedure [15] using about 20 minutes per step:

1. Individual preparation.
2. 1-to-1's: Modulation with and coaching by Project Management .
3. Team meeting: Synchronization and synergy with the team.

Project Management in step 2 is now any combination, as appropriate, of the following functions:

- The Project Manager or Project Leader, for the project issues.
- The Architect, for the product issues.
- The Test Manager, for the testing issues.

There can be only one captain on the ship, so the final word is to the person who acts as Project Manager, although he should better listen to the advice of the others.

Testers participate in requirements discussions. They communicate with developers in the unplannable time [16], or if more time is needed, they plan tasks for interaction with developers. If the priority of an issue is too high to wait for the next TaskCycle, the interrupt procedure [17] will be used. If something is unclear, an Analysis Task [18] will be planned. The Prevention Potential of issues found is an important factor in the prioritizing process.

In the team meeting testers see what the developers will be working on in the coming week and they synchronize with that work. There is no ambiguity any more about which requirements can be tested and to which degree, because the testers follow development, and they design their contribution to assist the project optimally for success.

In Evo Testing, we don't wait until something is thrown at us. We actively take responsibility. Prevention doesn't mean sitting waiting for the developers. It means to decide with the developers how to work towards the defect free result together. Developers doing a small step. Testers checking the result and feeding back any imperfections before more imperfections are generated, closing the very short feedback loop. Developers and testers quickly finding a way of optimizing their cooperation. It's important for the whole team to keep helping each other to remind that we don't want to repair defects, because repair costs more. If there are no defects, we don't have to repair them.

Doesn't this process take a lot of time? No. My experience with many projects shows that it *saves* time, projects successfully finishing well before expected. At the start it takes some more time. The attitude, however, results in less defects and as soon as we focus on prevention rather than continuous injection-finding-fixing, we soon decrease the number of injected defects considerably and we don't waste time on all those defects any more.

16. Database for Change Requests and Problem Reports and *Risk Issues*

Most projects already use some form of database to collect defects reported (Problem Report/PR: development pays) and proposed changes in requirements (Change Request/CR: customer pays).

If we are seriously in Prevention Mode, striving for Zero Defects, we should also collect Risk Issues (RI): issues which better be resolved *before* culminating into CR's or PR's.

With the emphasis shifted from repair to prevention, this database will, for every RI/CR/PR, have to provide additional space for the collection of data to specifically support the prevention process, like:

- Follow-up status.
- When and where found.
- Where caused and root cause
- Where should it have been found earlier
- Why didn't we find it earlier
- Prevention plan
- Analysis task defined and put on the Candidate Task List [19].
- Prevention task(s) defined and put on the Candidate Task List.
- Check lists updated for finding this issue easier, in case prevention doesn't work yet.

Analysis tasks may be needed to sort out the details. The analysis and repair tasks are put on the Candidate Task List and will, like all other candidate tasks, be handled when their time has come: if nothing else is more important. Analysis tasks and repair tasks should be separated, because analysis usually has priority over repair. We better first stop the leak, to make sure that not more of the same type of defect is injected.

17. How about metrics?

In Evo, the time to complete a task is estimated as a TimeBox [20], within which the task will be 100% done. This eliminates the need for tracking considerably. The estimate is used during the execution of the task to make sure that we complete the task on time. We experienced that people can quite well estimate the time needed for tasks, *if* we are really serious about time.

Note that exact task estimates are not required. Planning at least 4 tasks in a week allows some estimates to be a bit optimistic and some to be a bit pessimistic. All we want is that, at the end of the week, people have finished what they promised. As long as the *average* estimation is OK, all tasks can be finished at the end of the week. As soon as people learn not to overrun their (average) estimates any more, there is no need to track or record overrun metrics. The attitude replaces the need for the metric.

In many cases, the deadline of a project is defined by genuine external factors like a finite market-window. Then we have to predict which requirements we can realize before the deadline or "Fatal-Date". Therefore, we still need to estimate the amount of work needed for the various requirements. We use the TimeLine technique to regularly predict what we will have accomplished at the FatalDate *and what not*, and to control that we will have a working product well before that date. Testers use TimeLine to control that they will complete whatever they have to do in the project, in sync with the developers.

Several typical testing metrics become irrelevant when we aim for defect free results, for example:

- *Defects-per-kLoC* or *Defects-per-page*
  Counting defects condones the existence of defects, so there is an important psychological reason to discourage counting them.
- *Incoming defects per month,* found by test, found by users
  Don't count incoming defects. Do something about them. Counting conveys a wrong message. We should better make sure that the user doesn't experience any problem.
- *Defect detection effectiveness* or *Inspection yield* (found by test / (found by test + customer)
  There may be some defects left, because perfection is an asymptote. It's the challenge for testers to find them all. Results in practice are in the range of 30% to 80%. Testers apparently are not perfect either. That's why we must strive towards zero defects *before* final test. Whether that is difficult or not, is beside the point.
- *Cost to find and fix a defect*
  The less defects there are, the higher the cost to find and fix the few defects that slip through from time to time, because we still have to test, to see that the result is OK. This was a bad metric anyway.
- *Closed defects per month* or *Age of open customer found defects*
  Whether and how a defect is closed or not, depends on the prioritizing process. Every week any problems are handled, appropriate tasks are defined and put on the Candidate Task List, to be handled when their time has come. It seems that many metrics are there because we don't trust the developers to take appropriate action. In Evo, we do take appropriate action, so we don't need policing metrics.
- *When are we done with testing?*
  Examples from conventional projects: if the number of bugs found per day has declined to a certain level, or if the defect backlog has decreased to zero. In some cases, curve fitting with early numbers of defects found during the debugging phase is used to predict the moment the defect backlog will have decreased to zero. Another technique is to predict the number of defects to be expected from historical data. In Evo projects, the project will be ready at the agreed date, or earlier. That includes testing being done.

Instead of *improving* non-value adding activities, including various types of metrics, it is better to *eliminate* them. In many cases, the attitude, and the assistance of the Evo techniques replace the need for metrics. Other metrics may still be useful, like *Remaining Defects*, as this metric provides information about the effectiveness of the prevention process. Still, even more than in conventional metrics activities, we will be on the alert that whatever we do must contribute value.

If people have trouble deciding what the most important work for the next week is, I usually suggest as a metric: "*The size of the smile on the face of the customer*". If one solution does not get a smile on his face, another solution does cause a smile and a third solution is expected to put a big smile on his face, which solution shall we choose? This proves to be an important Evo metric that helps the team to focus.

18. Finally

Many software organizations in the world are working the same way, producing defects and then trying to find and fix the defects found, waiting for the customer to experience the reminder. In some cases, the service organization is the profit-generator of the company. And isn't the testing department assuring the quality of our products?

That's what the car and electronics manufacturers thought until the Japanese products proved them wrong. So, eventually the question will be: can we afford it?

Moore's Law is still valid, implying that the complexity of our systems is growing exponentially, and the capacity needed to fill these systems with meaningful software is growing exponentially even faster with it. So, why not better become more productive by not injecting the vast majority of defects. Then we have more time to spend on more challenging activities than finding and fixing defects.

I absolutely don't want to imply that finding and fixing is not challenging. Prevention is just cheaper. And, testers, fear not: even if we start aiming at defect free software, we'll still have to learn a lot from the mistakes we'll still be making.

Dijkstra [8] said:

> *It is a usual technique to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.*

Where we first pursued the very effective way to show the presence of bugs, testing will now have to find a solution for the hopeless inadequacy of showing their absence. That is a challenge as well.

I invite testers from now on to change their focus from finding defects, to working with the developers to minimize the generation of defects in order to satisfy the real goal of software development projects. Experience in many projects shows that this is not an utopia, but that it can readily be achieved, using the Evo techniques described.

References

[1] The Standish Group: *Chaos Report*, 1994, 1996, 1998, 2000, 2002, 2004.
www.standishgroup.com/chaos/intro1.php

[2] N.R. Malotaux: *How Quality is Assured by Evolutionary Methods*, 2004.
PNSQC 2004 Proceedings. Also downloadable as a booklet:
www.malotaux.nl/nrm/pdf/Booklet2.pdf

[3] N.R. Malotaux: *Evolutionary Project Management Methods*, 2001.
www.malotaux.nl/nrm/pdf/MxEvo.pdf

[4] T. Gilb: *Principles of Software Engineering Management*, 1988.
Addison-Wesley Pub Co, ISBN: 0201192462.

[5] See cases: www.malotaux.nl/nrm/Evo/EvoFCases.htm

[6] P.B. Crosby: *Quality Without Tears*, 1984.
McGraw-Hill, ISBN 0070145113.

[7] W.E. Deming: *Out of the Crisis*, 1986. MIT, ISBN 0911379010.
M. Walton: *Deming Management At Work*, 1990. The Berkley Publishing Group, ISBN 0399516859.

[8] E. Dijkstra: Lecture: *The Humble Programmer*, 1972.
Reprint in *Classics in Software Engineering*. Yourdon Press, 1979, ISBN 0917072146.

[9] TaskCycle	ref [2] chapter 5.1	ref [3] chapter 3C
[10] DeliveryCycle	ref [2] chapter 5.1	ref [3] chapter 3C
[11] TimeLine	ref [2] chapter 5.5 and 6.8	
[12] Product	ref [2] chapter 4.2	
[13] Project	ref [2] chapter 4.3	
[14] Process	ref [2] chapter 4.4	
[15] 3-step procedure	ref [2] chapter 6.9	
[16] Unplannable time	ref [2] chapter 6.1	
[17] Interrupt procedure	ref [2] chapter 6.7	
[18] Analysis task	ref [2] chapter 6.6	ref [3] chapter 8
[19] Candidate Task List	ref [2] chapter 6.5	ref [3] chapter 8
[20] TimeBox	ref [2] chapter 6.4	ref [3] chapter 3D

Service Components & Compositions

Jeff Davis, http://www.manning.com/davis/

This article is based on Open Source SOA (http://www.manning.com/davis/), to be published in March 2009. It is being reproduced here by permission from Manning Publications (www.manning.com). Manning early access books and ebooks are sold exclusively through Manning. Visit the book's page for more information.

In the previous two excerpts from the forthcoming Manning Publications book *Open Source SOA*, we dissected the technical underpinnings of what constitutes a service oriented architecture (SOA), and selected a set of open source products that can be used to develop what we are calling the OpenSOA Platform. The remaining chapters from the book then begin a detailed examination of how to use these products, and perhaps as important, how to integrate them into a comprehensive solution. This excerpt is from Chapters 3 and 4 of the book and examines how to develop service components and combine them, when appropriate, to form service compositions. The open source product used for this is *Apache Tuscany*, which is an implementation of the *Service Component Architecture* (SCA), a standard now being shepherded by the OASIS group with support by many leading vendors such as IBM and Oracle.

Services, as the “S” in SOA suggests, are instrumental in building an SOA environment. What is a *service*? It is a self-contained, reusable and well-defined piece of business functionality encapsulated in code. Services are indeed the “holy grail” of SOA. If properly designed, publicized and self-describing, they become assets that can be widely reused in a variety of applications. This maximizes the investment in these assets and enables creation of a more agile enterprise since every business process does not have to be re-created from scratch. Other tangible benefits include a reduction in maintenance costs and more bug-free applications.

This excerpt from chapters 3 and 4 will describe, at a high-level, the main concepts around SCA and its related standard, *Service Data Objects* (SDO). The complete chapters found in the book provide extensive code samples to buttress the concepts presented in this excerpt.

The Service Component Architecture

The SCA initiative, as its charter, defines a model for creating, assembling, and deploying service components using heterogonous technologies. This addresses the “S” in SOA. As part of its features it includes an XML-based declarative model for describing how components are assembled and how dependencies between components are linked together. It includes a *binding* model that allows for different types of communication protocols to be used when exposing a component as a service. Lastly, through its *policy* model, advanced infrastructure services such as security, transactions and reliable message can be defined.

SDO, a companion to SCA, is a specification for a language neutral, XML-based data model intended to facilitate the exchange of data between systems and applications. It offers several unique features, including: support for *disconnected* datasets, which uses change logs to identify modifications; an API for dynamically or statically constructing datasets; and built-in metadata support so that a dataset can be interrogated “on-the-fly” to understand its structure and format. The API is designed to facilitate tooling, and some vendors are already moving in that direction. SCA implementations such as Tuscany have placed special emphasis on integrating SDO as the preferred data exchange format.

Armed with this general overview of the OSOA technologies, we will now explore SCA in greater detail, and discover how its innovative assembly model advances the creation of services that are fundamental to an SOA architecture.

Understanding the SCA Assembly Model

The SCA Assembly model represents the core of the specification – it describes how services are constructed. The main ingredients of the model are shown in Figure 1

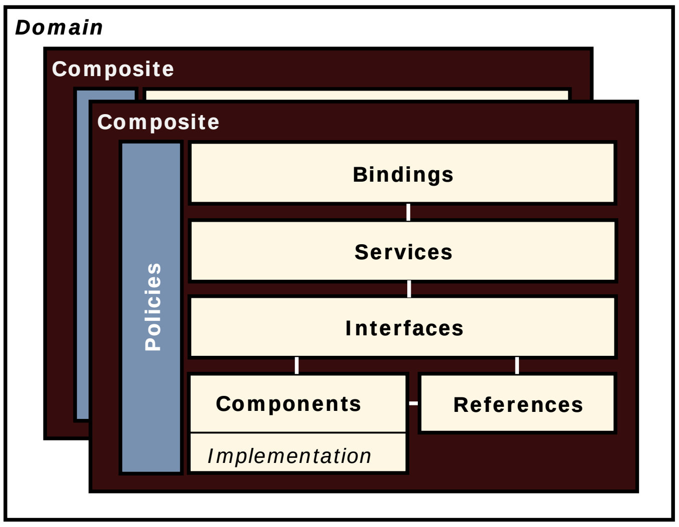


Figure 1 SCA Assembly Model

As figure 1 illustrates, multiple *composites* can reside within a single SCA *domain*. A composite can contain one or more *services*, which are, in turn, created by *components*. A component has concrete *implementation*, and can *reference* other components or services. In order to further demonstrate these concepts, let’s examine each part of the assembly and provide sample illustrations of how they are constructed. Let’s begin by looking at composites, which form the foundation for SCA assemblies.

Composites

A *composite* is a container that is used for defining services, components and their references. It is used to assemble SCA elements into logical groupings of components. For example, all operations that deal with a CRM system’s customer management might be grouped together in a single composite. Interestingly, a composite itself may be treated as a component, which is an example of what SCA calls a *recursive composition* (as you may recall from earlier, multiple composites may exist within a given *domain*). A composite defines the public services that are made available though one of the available bindings (JMS, SOAP etc). Most compositions will likely contain one or more components, references and often at least one service. Property values are useful for inserting default values into component classes, or for configuration options that can be declaratively defined (such as JDBC connectivity parameters, for example). Let’s take a look at the parts of a composite, starting with the most basic building block, the component.

Components

A *component* is business function expressed as code. Components both provide and consume *services*. To be a functional unit, a component must provide an *implementation* that represents the actual code used to provide the functionality. As we discussed in the previous section, a component can also directly specify a *service* by which it can be exposed for external consumption, as well as identify any *references* or dependencies of the component. How is a component defined within a composite? Listing 1 depicts a simple example of a problem ticket service that is comprised of two components: `ProblemTicketComponent` and `CreateTicketComponent`.

Listing 1 Example SCA composite assembly XML

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
   targetNamespace="http://opensoa.book.chapter3"
   xmlns:hw="http://opensoa.book.chapter3"
   name="ProblemManagementComposite">

   <component name="ProblemTicketComponent">                        |#1
      <implementation.java
 class="opensoa.book.chapter3.impl.ProblemTicketComponentImpl" />
       <service name="ProblemTicketComponent">                      |#2
            <binding.ws
           uri="http://localhost:8085/ProblemTicketService"/>    |#5
          </service>
       <reference name="createTicket" target="CreateTicketComponent"/>|#3
      </component>

   <component name="CreateTicketComponent">                         |#4
      <implementation.java
          class="opensoa.book.chapter3.impl.CreateTicketComponentImpl"/>
   </component>
</composite>
```

Notice the two components elements that are defined. The `ProblemTicketComponent` ([#1]).is exposed as a SOAP-based web service by the embedded service element ([#2]) that is present. In addition, the component defines a dependency/reference to the `CreateTicketComponent` ([#4]) by virtual of the child reference ([#3]). A dynamic WSDL is being generated by the service element, since an actual WSDL was not specified (one of the features of SCA). The WSDL location would be: http://localhost:8085/ProblemTicketService?wsdl, as that is the `binding.ws@uri` attribute ([#5]).

What listing 1 demonstrates is how a component can be exposed a web service (`ProblemTicketComponent`). In turn, this component uses the services of another component, the `CreateTicketComponent`. This should give you some idea of how components can be defined and used with SCA (the book chapters provide the actual Java implementation used for each component, and how they are setup for use by SCA). What should be noted is that there is no SOAP or web service specific code anywhere within the component implementation class, so the component itself is not bound directly to a specific protocol. Instead, the binding of the protocol to the component is done declaratively through the service element definition. This form of lose coupling is very appealing, because as we'll see next, we can now expose services through any number of different protocols, or bindings, without having to change any implementation code. This is truly exciting stuff!

Services

We have already demonstrated some of the capabilities of the service element using the problem ticket example introduced in the previous section. The service element is used to expose a component's functionality for external consumption through any number of communication protocols such as SOAP or JMS. The consumer of the service can be another component or an external client running outside of the SCA framework. An example of such a client could be one using SOAP-based web services or perhaps depositing a message into a JMS queue.

Properties

Up to this point we have shown the basics of components and services -- the building-blocks to SOA. The SCA standard also provides convenient run-time configuration capabilities through properties and references. Properties, the subject of this section, are much like their namesakes in the Spring framework, and are used for populating component variables in a setting injection-style fashion. This is obviously very useful where environment-specific values need to be set without having to resort to using a external property file. More impressively, properties can be complex XML structures, which in turn can be referenced via XPATH locations by the components using it. What does a simple example of a property definition look like? Listing 2 shows a simple usage of a property embedded within a component definition:

Listing 2 Example of properties used within SCA component definition

```
<component name="ProblemTicketComponent">
     <implementation.java
         class="opensoa.book.chapter3.impl.ProblemTicketComponentImpl" />

    <service name="ProblemTicketComponent">
         <binding.ws uri="http://localhost:8085/ProblemTicketService"/>
      </service>

    <property name="username">jdoe@mycompany.com</property>
    <property name="password">mypassword1</property>
    <reference name="createTicket" target="CreateTicketComponent"/>
</component>
```

As the book discusses, there are many different ways by which properties can be used. In listing 2, the Java implementation class ProblemTicketComponentImpl would simple have specified a member variable of the same name as assigned to the property in order to receive its value.

The next "essential" SCA technology we will address is implementation. It is how SCA provides for multi-language support and is the mechanism by which recursive compositions are created, which is the nesting of composites to create higher-level assemblies. As you may recall from our SOA discussion in chapter 1, the ability to easily create components that are themselves comprised of one or more sub-components greatly facilitates code reuse, which is a key objective behind SOA.

Implementations

The implementation node, as we have seen in previous examples, is a child element of component. It is used to define the concert implementation for the service. In the examples thus far, we have used the Java implementation, as specified by using `implementation.java`. However, as we have pointed out, SCA is designed to support multiple languages.

Which languages are supported is driven by the SCA implementation. Apache Tuscany, the open source SCA offering we are using, supports the following:

Table 1 Apache Tuscany SCA Implementation Types

TYPE	DESCRIPTION
Java Components	We have been using Java components in the examples thus far.
Spring assemblies	You can invoke Java code exposed through Spring.
Scripting – JavaScript, Groovy, Ruby, Python, XSLT	Uses JSR 223 to support a wide variety of scripting languages
BPEL	Integration with Apache ODE for BPEL support.
XQuery	Supports Saxon XQuery.
OSGi	Supports Apache Felix OSGi impelemtation.

One implementation that is anticipated to be supported by all SCA-compliant products is the composite implementation. Using `implementation.composite`. you can construct a hierarchy of services which are layered upon each other. At the lowest level you would presumably have finer-grained components that perform very specific, narrow functions. As you move up the hierarchy, you could construct more courser-grained services that incorporate or build upon the lower-level ones. You could also wrap components that are designed primarily for one purpose and re-purpose them for other users.

References

In SCA, reference are somewhat analogous to properties in that they allow run-time configurations to be made without requiring implementation code changes. In the case of references, you insert dependent class instances in much the same way that properties can be used to insert static data into a class. Such flexibility contributes to the agility so often touted as a major beneficiary of moving to SOA. In listing 1 we demonstrated a simple use of a reference called `createTicket`. This inserted a reference, or handle, to the `CreateTicketComponent` and made it available to the `ProblemTicketComponent`. In chapter 3 in the book, it shows multiple ways in which references can be used.

References are one of the most powerful features of SCA, and they likely will play a central role in how you design assemblies. We will now turn to the last of the core SCA technologies – bindings.

Bindings

Bindings truly represent one of the most innovative and value-added aspects of SCA, for it allows you to create and reference services over a variety of different communication protocols, all-the-while the underlying implementation classes are oblivious to it. Table 2 lists the protocol bindings that are available through Apache Tuscany.

Table 2 Apache Tuscany SCA Bindings

TYPE	DESCRIPTION
Webservice (SOAP)	Based on Apache Axis, which is one of the most popular SOAP engines available. Known for its high performance, standards-compliance and architecture (<binding.ws/>). You can expose components as service or using binding for referencing.
JMS	Tested using Apache ActiveMQ, which has become the most popular JMS solution available (<binding.jms>). Can be used for services and references.
JSON/RPC	This support for Javascript's JSON protocol is useful when crafting solutions that use a web browser interface (<binding.jsonrpc/>). RPC-style support is available through Ajax binding (binding.ajax). JSON binding exists for the service level only (not reference).
EJB	Enables SCA components to access existing stateless session beans (<binding.jms/>). Reference support only.
Feed (Atom, RSS)	Service support for RSS (<binding.rss/>) and Atom (<binding.atom/>) newsfeed protocols.
RMI	Support for Java's Remote Method Invocation protocol for service or references (<binding.rmi/>).

As you can see, there is an ever-growing list of protocol bindings.

Advanced SCA Features

You should now have a sense of the capabilities offered by SCA, and with the code examples provided in Chapter 3 of the book, would possess sufficient knowledge to begin using SCA within your own organization. However, there are many advanced features of SCA that are described in much greater detail in Chapter 4. A brief summary of these advanced features follows.

Component Types

While in this except we have not demonstrated the specific implementation classes used for the SCA components we have discussed, suffice it to say that SCA leverages the Java annotation features that first appeared with the Java 1.5 release. SCA introduces its own set of annotations which make it very simple to define references, services, properties etc. Within your own environment, you may not always have ready access to modify existing legacy code to include such annotations. Or, perhaps, your coding standards frown upon annotations. In any event, SCA does provide an alternative known as *Component Types* that act an alternative to annotations.

To use this capability, you create what is known as a component type XML file for each class/object that you are working with, and give the file name a .componentType extension, located in the same classpath location as the source object. Listing 3 is an example of such a file:

Listing 3 Example of a Component Type XML descriptor file

```
<componentType
   xmlns="http://www.osoa.org/xmlns/sca/1.0"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <service name="ProblemTicketComponent">
      <interface.java
         interface="opensoa.book.chapter41.ProblemTicketComponent" />
   </service>

   <reference name="createTicket">
      <interface.java
         interface="opensoa.book.chapter41.CreateTicketComponent" />
   </reference>

   <property name="username" type="xsd:string" />
   <property name="password" type="xsd:string" />
</componentType>
```

The `service` and `reference` nodes identify the interface classes used for each. When using annotations, these would be defined within the implementation classes directly using the `@Service` and `@Reference` annotations. In a similar fashion, the `@Property` is normally used to indicate which class variable or method is used to receive inbound properties in lieu of specifying it within a component type file.

Conversations

Although best practices suggest that services be made stateless, as this has immediate advantages in scalability, in practical terms, this is not always feasible, or desired. What is an example where we might want a stateful service? In the example we are building, we will be using Salesforce.com SOAP-based web services. Salesforce's API, like many, requires that you first call a login service with your provided username and password. If successfully logged-in, you are returned a `sessionId` that must be used in subsequent calls. The `sessionId` will become invalid after some period of inactivity (2 hrs), however, and has to be refreshed. Retrieving a `sessionId` is an expensive operation, and can be relatively slow, so fetching a new one for each-and-every call is not advisable. SCA supports 4 types of conversation scopes:

- **COMPOSITE**. All requests are dispatched to the same class instance for the lifetime of the composite. The example above with the `sessionId` is this type of scope.
- **CONVERSATION**. Uses correlation to create an ongoing conversation between a client and target service. The conversation starts when the first service is requested, and terminates when an end operation is invoked.
- **REQUEST**. Similar to a web servlet request scope, whereby a request on a remotable interface enters the SCA runtime and processes the request until the thread completes processing. Only supported by SCA clients.
- **STATELESS**. The default – no session is maintained.

Chapter 4 provides examples of each of these conversational scopes, and it is likely that you will encounter the need for such an ability as you begin to use SCA in non-trivial type implementations. Another somewhat related technology is a callback.

Callbacks

Callbacks are a form of bidirectional communications whereby a client requests a service and the provider or server returns back the results by “calling-back” the client over a given interface or operation. It is considered an asynchronous form of communication because the server does not return the results through its synchronous reply, as most commonly exemplified by web services or RFC-style communications. Callbacks are often used when the processing time required to fulfill a request is unpredictable or slow.

What is an example of how it might be used in the examples we’ve been building thus far? Let’s say that we want to capture all create ticket events so that they can be fed into a business activity monitoring (BAM) dashboard that executives would monitor. For example, they might want to be apprised of any unusual spikes in ticket creation activity. To illustrate this, let’s create a new component that is called in an asynchronous fashion, and this component would, in real life, send an event to the BAM or complex event processing system. We do want to ensure that the BAM system was able to process the event successfully, so we will use a callback that just indicates whether the message was processed correctly.

Callbacks are a fairly advanced feature, and like conversations, require a fair amount of explanation that falls outside of the scope of this excerpt. However, it is important you understand such an ability exists if you are currently evaluating the merits of SCA.

Production Deployment

By default, Tuscany includes a built-in Jetty engine for surfacing the bindings that we have been using thus far (such as web services). In most production scenarios, however you will likely use a servlet containers such as Apache Tomcat, at least for exposing HTTP-based services such as SOAP. In part, this is because, by using the embedded container, each domain would require its own dedicated IP port. It is easier to use a servlet container, as you can run multiple domains within that single instance (the book describes setting this up in detail). You can also mix containers/domains in a distributed fashion, as is illustrated in figure 2.

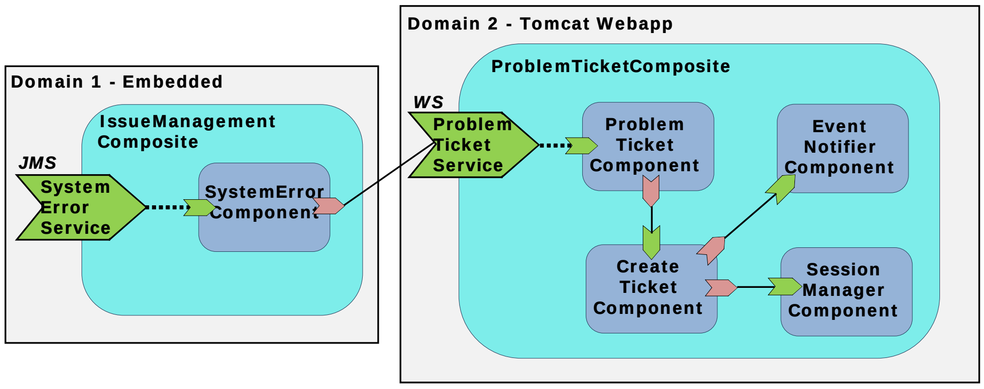


Figure 2 Example of a distributed SCA architecture

In this example, an embedded domain is used to house a JMS service, which then references a component running within a separate domain running within Tomcat. This gives you a sense of the flexible, distributed capabilities inherent in the SCA architecture.

Scripting Language Support

One of the main selling points of SCA is that it supports multiple languages (the particular SCA implementation, such as Apache Tuscany, determines which individual languages are supported). The days are likely past when an organization is entirely homogenous in their selection of programming languages. In particular, scripting languages such as Python, Perl and Ruby have become increasingly popular. Their agile development cycles and, in some cases, simplified language structures have contributed to their success. Thus, supporting multiple languages is becoming a requirement for enterprises adopting SOA. Tuscany's SCA reference implementation supports the following scripting languages: JavasScript, Groovy, Ruby, Python, XSLT and XQuery. Once you've created the script using one of the aforementioned languages, you then define the component's implementation by identifying the script file location (notice the tuscany namespace). For example

```
<component name="EmailServiceComponent">
   <tuscany:implementation.script script=
    "opensoa/book/chapter4/impl/Email.rb"/>
 </component>
```

In this example, a Ruby script is being called. Because Java annotations cannot be used by most scripting languages, a component type file is required if you wish to pass references or properties to the script. The last advanced topic of SCA we'll cover is its companion specification, *Service Data Objects* (SDO).

Service Data Objects

SDO is designed to simplify working with complex, XML-based data structures, such as a purchase order or invoice. Upon first examination of SDO, an inevitable question often arises – how is this binding technology different than the multitude of others that exist, such as Castor, JiBX, XMLBeans or JAXB? To be honest, it does share many characteristics with these technologies, but offers extended functionality that does not exist in those binding solutions. Specifically, it was designed to support "off-line" processing, where changes to the dataset are automatically captured into change summaries that indicate any new, modified or deleted data. SDO also supports a rich set of metadata which allows the client to retrospectively examine the "data graphs" for their structure and form. Last but not least, SCA was designed to work seamlessly with SDO, and this becomes apparent when working with more complex XML structures.

SDO, like the other XML binding technologies mentioned, has a variety of different configuration options, and these are addressed in Chapter 4 of the book. In that chapter, we describe how a top-down approach can be used, whereby a complex data structure is first defined using XML Schema. From there, the follow steps are demonstrated:

1. How to develop and run an Ant target using `XSD2JavaGenerator` to generate Java classes from the WSDL XML schema definition.
2. How to create a new a new service interface and implementation that uses the new SDO generated classes.
3. How to create a client composition and components so that we can submit test SOAP requests.

Upon finishing chapter 4 in the book, you will have a solid understanding of how SDO can be used in-tandem with SCA for creating non-trivial services. This concludes our summary coverage of the advanced features of SCA and SDO.

Summary

The Service Component Architecture initiative, which is sponsored by a virtual “who-s who” of vendors, offers a framework for building reusable, multi-language, components that can be exposed using a variety of communication protocols. In this excerpt from Chapters 3 and 4 from the forthcoming Manning Publications book *Open Source SOA*, we provided a brief overview of the main elements of SCA assembly model, including composites, components, services and references. We also touched upon some of the advanced features such as conversational services and callbacks. While some of these features may not be immediately used by newcomers to SCA, any widespread SCA rollout will likely involve them to some degree. Other exciting capabilities we described were how to use languages other than Java to create and consume services. The ability to use increasingly popular languages such as Ruby is a real selling point of SCA.

The complete chapters in the book provide greater coverage of core and advanced features, and are illustrate with numerous code samples. Follow-up chapters build upon the foundation of SCA/SDO to create complex business processes; expose business rules as services; and monitor, in real-time, your enterprise pulse through event stream processing.

Advertisement – Software Development Videos - Click on ad to reach advertiser web site

Software Development Links

http://www.softdevlinks.com/

SoftDevLinks.com is a general directory for software developers, testers and managers. If you have a blog, a web site, distribute a tool, work on an open source project or for a consulting company related to software development, do not hesitate to improve your visibility by adding a (free) link in this directory.

White Paper: Driving Software Quality Through Requirements-Based Testing Research shows that the root cause of 56 percent of errors identified in projects are introduced in the requirements phase. Reduce your risk by implementing a requirements-based testing methodology. This white paper details how MKS Integrity ensures complete requirements test coverage, establishing traceability between all artifacts in the application lifecycle (such as requirements, specifications, test cases and code), to improve software quality and increase efficiency.

http://www.mks.com/mtdc-rbt

Open Source .NET eXchange 2009, January 22 2009, London, UK. In this mini-conference, members of the ALT.NET community will talk about their favourite alternative .NET tools that increase programmer productivity and help us do our work better but are still not mainstream. The sessions will be aimed at .NET developers at all levels. http://skillsmatter.com/event/open-source-dot-net/open-source-dot-net-exchange

---

Spring eXchange 2009, January 29 2009, London, UK. Come join us at this 1-day, 1 track event and hear from SpringSource CEO and Spring creator Rod Johnson, and key committers to the Spring project, including Russ Miles, Dave Syer, Sam Brannen, a joint talk by Adrian Colyer and Rob Harrop and more!
http://skillsmatter.com/event/java-jee/spring-exchange

Get yourself informed on the software development conferences schedule for 2009 on http://www.softdevconferences.com/

Advertising for a new Web development tool? Looking to recruit software developers? Promoting a conference or a book? Organizing software development training? This clasified section is waiting for you at the price of US $ 30 each line. Reach more than 50'000 web-savvy software developers and project managers worldwide with a classified advertisement in Methods & Tools. Without counting the 1000s that download the issue each month without being registered and the 50'000 visitors/month of our web sites! To advertise in this section or to place a page ad simply http://www.methodsandtools.com/advertise.php

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