
METHODS & TOOLS

Global knowledge source for software development professionals

ISSN 1023-4918

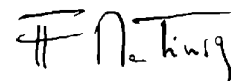
Summer 2004 (Volume 12 - number 1)

How Expensive is a Developer?

There are some special occasions in your life as a developer when you have to negotiate the salary or the hourly rate you will work for. This happened to me recently and gave me the idea to write this editorial. Like IT in general, software development is considered by the majority of organisations as a cost. Therefore, the numbers that will be discussed are often determined by salary surveys or current market rates for the activities that the developer will perform. As many companies are working on a project-basis, another assumption is that you should look at the margin between the developer's cost and what can be charged to the customer. The problem with this approach is that developers are rarely considered for the results that they can produce.

This situation is created by the fact that there are few opportunities to measure the productivity of a developer. In the famous "bang per buck" measure of productivity described by Tom de Marco more than 20 years ago, the majority of companies are still struggling to figure out the bucks (the cost of a project) and do not care about measuring the bangs (the deliverables). According to some studies, a programmer's productivity can vary largely, without any relation to the quality of the code produced, but few companies measure this aspect. It is true that software development productivity is difficult to define and measure, but this situation is also caused by the reluctance of developers to incorporate anything that looks like productivity measure into their activities. Until this attitude changes, we will remain locked in a situation where a developer is evaluated mainly on a cost basis and not on the delivered results. This situation provides companies many reasons to consider developers as expensive.

In this issue you will find an article on how to use metrics to analyse the software development process. There is also an inspiring article on precise use cases. I would like to quote just one part of it that you can remember in your next analysis phase: " Note that software development always gets to formality and precision if code is produced. So the question is not if formality and precision, but when and by whom it is introduced and who can review it. Since ambiguity often masks the seeds of failure, introducing formality and precision earlier in the development process can reduce the risk of critical information being missed or guessed at by the developers."



Inside

Software Process: Identifying Best Practices..... page 2

UML: Precise Use Cases..... page 7

Identifying your Organization's Best Practices

David Herron, dh@davidconsultinggroup.com
and David Garmus, dg@davidconsultinggroup.com
The David Consulting Group, www.davidconsultinggroup.com

Characterizing an organization's best practices can easily be defined as those software development practices that yield favorable results. Favorable results may be measured in relation to customer satisfaction, reduced time of delivery, decreased cost, better quality, etc. This article will explore opportunities for an organization to improve their ability to identify best practices and to ultimately achieve more positive results.

We are most interested in the results that meet business goals and objectives. A basic set of business goals related to software application development and maintenance typically includes the following objectives:

- Decrease project costs
- Reduce time to market
- Minimize defects delivered
- Improve performance

Advertisement – Enterprise Change Management Success - Click on ad to reach advertiser web site



Merant + Serena =
*Dimensions + Professional + Tracker + Version Manager +
Collage + Build + Meritage + Modello + Mover +
TeamTrack + ChangeMan + StarTool =*
???

**Change
is easy.**

MKS Integrity Solution +
Outstanding Customer Service =
**Enterprise Change
Management Success**

mks®
Build Better Software®

Phone: 1 800 265 2797
Email: info@mks.com
Web: <http://www.mks.com>

"I have just migrated my company from PVCS to MKS using
MKS Source Integrity Enterprise and MKS Integrity Manager.
I'm very pleased with our decision."
Mark Andrews, Senior Vice President and CIO,
Check Technology, Certegy Inc.

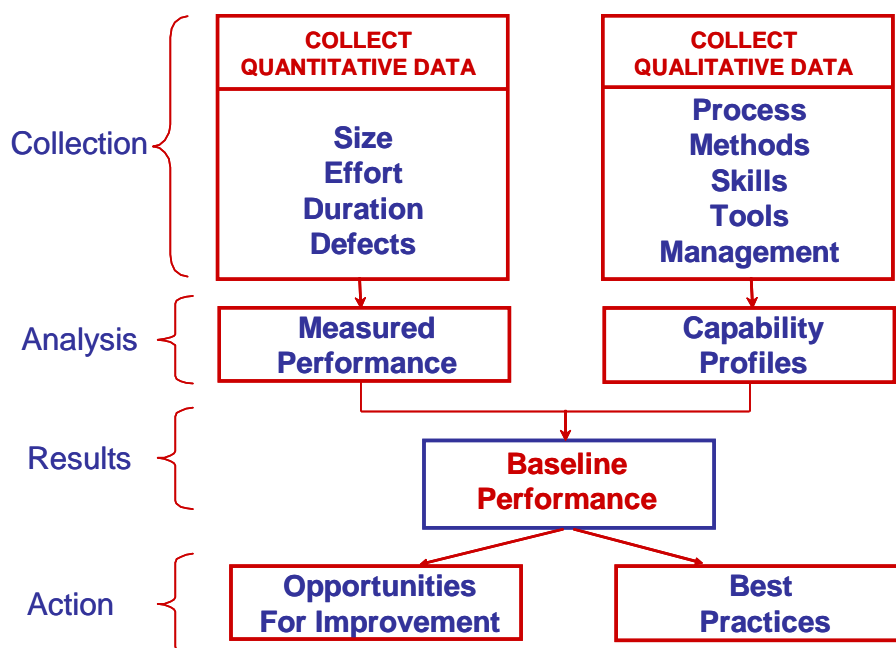
Frequently, the strategy to achieve these goals is formulated around quick-fix approaches. Cost reduction often tops the list and can be the driving force behind the decision to outsource software development to an offshore provider. Time to market is often reduced by delivering fewer features to the end user, thus reducing the development work load. Defect minimization is too often ignored. However, regardless of the means, the good news here is that all of these goals are measurable and can be achieved as we deliver and maintain software.

The key to successful performance management is performance measurement. As the software industry grows out of its infancy into a more mature set of practices, the inclusion of performance measurement to manage and direct decisions is becoming a mainstream practice. It is obviously important to establish proper goals and objectives; equally important, however, is that effective measurement provides the key to recognizing and realizing those goals and objectives.

A basic measurement model that is advanced by the Practical Software and Systems Measurement (PSM) program suggests that an organization follow these steps:

- Identify the needs of the organization.
- Select the measures.
- Integrate measurement into the process.

Building upon this basic measurement model, we will further examine the key elements necessary in selecting your appropriate measures. The selected measures must have a balanced view and, therefore, must consider both quantitative and the qualitative information. Using the model depicted below, we can begin to see how the quantitative and qualitative measures can together provide a complete organizational view.



Advertisement - European SEPG 2004, 14th - 17th June 2004, London- Click on ad to reach advertiser web site

Ninth Annual European Systems &
Software Engineering Process Group
Conference

EUROPEAN SEPG 2004

Organisation for improvement

PEOPLE • PROCESS • TECHNOLOGY • MEASUREMENT • PROJECT MANAGEMENT



14th - 17th June 2004, London

www.espi.org

The European SEPG conference is the premier process improvement event in Europe, where leaders, innovators and practitioners meet to explore methods, tools and process improvements that aim to increase business performance through quality and productivity gains. In four intensive days of tutorials, presentations, panel discussions and informal meetings, delegates and presenters learn and share experience on the practical issues of making process improvement work on every level: for the business, IS organisation and individuals. An exhibitor showcase features some of Europe's leading providers of tools and services to support process improvement initiatives.

This year's conference features:

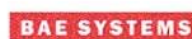
Measurement Symposium • Project Management Symposium • Culture & People • Team Software ProcessSM • Business Focus • CMMI[®] Appraisal Methods • Outsourcing & Supplier Relationships • Technology & Tools • Security & Process

The European SEPG[™] is a joint
initiative between:



Carnegie Mellon
Software Engineering Institute

Sponsors:



[™] European SEPG is a European Community registered trademark of the ESPI Foundation.
[®] CMMI is registered in the U.S. Patent and Trademark Office. SM Team Software Process is a service mark of the Carnegie Mellon University

This model depicts the collection of quantitative data (e.g., deliverable project size, effort required, duration of the project, pre and post defects) along with the qualitative data (e.g., processes, methods, skill levels, development tools, management practices). Collected on a project basis, the quantitative data results in a measured profile that indicates how well a project is performing. The qualitative data results in a capability profile, which identifies the attributes that are contributing to high or low yields of performance.

These two elements (quantitative and qualitative) come together to form what is commonly viewed as an organization's baseline of performance. The baseline values are compiled from a selection of measured projects and represent the overall performance level of the organization.

As you can well imagine, the results vary significantly. Some projects perform very well (i.e., they have low cost and high quality), and other projects do not perform nearly as well. The quantitative data gives us a picture of performance; the qualitative data provides us the opportunity to examine the attributes of the projects to determine why some projects have performed better than others. This can often lead to the identification of an organization's best practices and opportunities for improvement.

A few case studies follow that indicate how this data can be applied.

Case Study 1

A large organization wants a complete baseline performed consisting of both quantitative data collection and analysis and qualitative data collection and analysis. Data is collected on sixty-five completed projects, and productivity rates are then calculated based upon the findings. The results are divided into two categories (high performing projects and low performing projects), and an average is calculated for each category. See the note below regarding the measures.

	High Performers	Low Performers
Average project size in function points (FPs)	148	113
Average duration (in calendar months)	5.0	7.0
Average rate of delivery in FPs/person month (PM)	22	9
Average number of resources	2.4	1.8

The quantitative data demonstrated that high performing projects produced (on average) more functionality in a shorter timeframe with a modest increase in staffing levels.

Qualitative data (attributes about each project) was also collected so profiles of performance could be developed that identify characteristics, which were present in the high performing projects but absent from the lower performing projects. These sets of attributes are then considered to be the leading factors contributing to higher performing projects.

About the measures

FP Size – represents the average function point size of the delivered product. Some organizations use this as a measure of value (functionality) being delivered to the end user.

Duration – represent the overall duration of the project from requirements through to customer acceptance

Defect Density – is a function of defects per function points. A lower number represents fewer defects in the delivered product.

Case Study 2

The second case study involved an organization that wanted to compare its level of performance to industry benchmarks. Once again, the organization collected quantitative data and calculated performance indicators such as those in the first case study. After determining the current level of performance, a comparison to industry average and industry best practices benchmarks was accomplished.

The results were as follows:

	Client	Ind. Avg.	Best Practices
FP Size	567	567	567
Productivity (FP/PM	6.9	7.26	22.68
Duration (months	12	14	10
Defects/	.24	.12	.02

As we examine these data points, we learn the following. For projects of equal size, the client's productivity rate was close to the industry average (6.9 vs. 7.26), but the best practices value indicates that there is a great deal of room for improvement. The client is actually delivering products (on average) in a shorter time frame than industry average, but again is not as good as best practices. Finally, the level of quality (defect density) is significantly below industry data points.

By looking at this picture, we can imagine an organization that is getting the product out the door quickly by increasing staffing levels and short-cutting quality practices.

Case Study 3

The final case study involves an organization that wants to estimate the impact that an SEI CMM Level 3 process improvement initiative will have on their performance. In order to model this improvement, the organization must first determine its current baseline of performance and profile of performance.

Project data is collected and analyzed. Averages for size, productivity, duration, and quality are computed. Using the profile data, a mapping of current project attributes is completed and compared to a similar mapping of CMM Level 3 project attributes. A model is developed that will modify the current attribute profile to more closely reflect Level 3 practices. Using a modeling tool such as Predictor from DDB Software, a series of calculations are performed, and projected models of performance are produced. Let's look at the results:

	Client Performance	Level 3 Model	
FP Size	456	456	
Productivity	6.9	10.6	+ 53%
Duration	12	12	--
Defect Density	.12	.06	- 50%

The client performance levels are marked, and we can see the impact of CMM Level 3 practices for products of similar size. Productivity is projected to increase by 53%, and defect density will be improved by 50%. This modeling technique helps organizations to evaluate the potential benefits with process improvement programs or other strategic initiatives.

In Summary

We have seen, in these three case studies above, a variety of ways in which measurement data may be used to learn more about:

- An organization's level of performance,
- Key factors that contribute to high or low yields of productivity,
- The level of performance as compared to industry data points, and
- The impact of strategic initiatives through the use of performance modeling.

Precise Use Cases

David Gelperin, dave@livespecs.com
LiveSpecs Software, www.livespecs.com

Abstract

*This paper describes a precise form of use case that promotes the specification of inter-actor options and alternative course conditions. Precise cases use a description language with a Structured English grammar to specify interactions. The design goals for the notation are to maximize understandability by application experts, not familiar with formal object-oriented concepts, and to supply sufficient information and structure to support both manual and automated test design. In addition to a detailed example, a development process, and guidelines for modeling and test coverage are provided. **The reader is assumed to have a basic understanding of use cases.** [Editor: look at our Spring 2000 issue for an article written by Sinan Si Alhir introducing use case modeling]*

1. Introduction

Use Cases can be described by diagrams [16] and text [6]. Diagrams emphasize relationships between inter-actors and cases and between cases. Text describes inter-actor/system interaction at different levels of detail ranging from short summaries to longer precise specifications. Diagrams and summaries are useful during early elicitation, while more precise text is useful during detailed analysis and functional specification.

While use cases, in general, and their current detailed formulations, in particular, provide useful information to manual test design, no current use case formulation is adequate for automated test design that includes result checking. This paper describes a test-adequate text-based formulation, Precise Use Cases. Because this formulation makes more details explicit, it lowers the risk of misunderstandings and provides a firmer foundation for system development and manual testing as well.

Constantine [4, 5] has proposed approaches to some of the same issues. Although our work focuses on testing rather than usage-centered design, Precise Use Cases conform to Constantine's definition of essential cases and therefore are likely to be effective at supporting user interface design.

In addition, using precise cases should provide a firm foundation for usage-based reading [25], for building operational profiles [23], for generating user documentation [17], and for increasing the accuracy of test and development effort estimation [24].

The remainder of the paper is organized as follows: Sections 2 and 3 describe the information needs of testing and an approach to satisfying those needs. Section 4 describes the structure of a Precise Use Case, Section 5 provides a substantial example, Section 6 comments on aspects of the example, and Section 7 specifies the Structured English constructs used to precisely describe courses of action. Section 8 discusses alternative formats for course descriptions, Section 9 provides a few modeling guidelines, and Section 10 describes a development process for suites of cases. Finally, Section 11 provides test coverage criteria and Section 12 discusses related work.

2. A testing need

Effective testing must include usage scenarios. Use cases can be a valuable source of usage information and usage testing ideas.

The following information associated with a single usage scenario (i.e., complete path through a use case) is necessary (but not sufficient) to design a usage-based test:

- Sequence of user actions and system actions in the scenario
- Scenario pre, invariant, and post conditions
- Alternative initial states of the domain entities (e.g. order) associated with the scenario
- Alternative scenario trigger events
- Alternative inputs to the scenario
- Alternative Boolean values of the predicates determining the flow of the scenario

If use cases are specified informally or semi-formally [3, 6, 16, 19, 23], then high quality, usage-based test design remains a manual process and some of the information identified above will always be missing, unclear, or incorrect. Formality and precision are prerequisites for automating test design and for automatically verifying use cases as well.

Advertisement – Feature Rich UML Modeling Tool - Click on ad to reach advertiser web site

Enterprise Architect

INTUITIVE, FLEXIBLE, POWERFUL

NEW!

Version 4.00

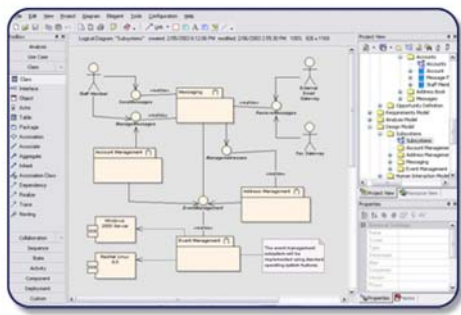
is now available...


The
Feature-Rich UML Modeling Tool
for the **Entire Development Team**

Object-oriented development is much more than developing a class model - it now embraces the full lifecycle of system development - business process analysis, use case requirements, dynamic models, component and deployment, system management, non-functional requirements, user interface design, testing and maintenance.

Enterprise Architect is the ideal UML tool to enable you to tie all of these elements together in a user-friendly, intuitive environment. **Take a Free Test Drive** of Enterprise Architect and discover why thousands of analysts, designers, architects, developers, testers, project managers and maintenance staff have come to rely upon the power and benefits of this robust, integrated project management and modeling tool.


Outstanding software at modest prices.
Leverage your advantage.





ENTERPRISE ARCHITECT
www.sparxsystems.com

A flexible, powerful and complete
UML modeling tool for the full development
lifecycle - at a competitive price.



Cockburn [7] reports that people do not like to write formal cases. He argues that, formal or not, use cases can not automatically produce system designs, UI designs, or feature lists. Formal cases seem to be pain, without gain. However, formal cases can support automated design of (manual or automated) test scripts. In addition, attempts at precision can bring to light critical issues hidden in informal descriptions. Sometimes formality is warranted due to the nature and extent of product risk.

Note that software development always gets to formality and precision if code is produced. So the question is not if formality and precision, but when and by whom it is introduced and who can review it. Since ambiguity often masks the seeds of failure, introducing formality and precision earlier in the development process can reduce the risk of critical information being missed or guessed at by the developers.

3. Satisfying the testing need

Precise Use Cases lie at the heart of a larger-scale description of application usage called a Precise Usage Model. In addition to use cases, these models contain:

- Application name & abstract
- Population description - including population diagram and inter-actor profiles
- Application domain description - including essential class descriptions, states, transition rules, class relationships, domain invariants, and function limits
- Definitions of derived attributes and conditions - optional
- Dependencies between conditions – optional
- Non-functional requirements – optional
- Work breakdown structure - optional
- Action contracts – optional [12]

This additional information provides context for the precise cases including background on the inter-actors involved as well as descriptions of the application entities, their relationships, their state changes, and their constraints. This additional information provides semantics for the conditions and actions in the use cases. Precise Usage Models are consistent with the specifications described by Larman [21]. A sample Precise Usage Model for a Library Management System can be found in [9].

The twin goals that drove the design of Precise Use Cases were to maximize the understandability of the cases by application experts, not familiar with formal object-oriented concepts, and to supply sufficient information and structure to support both manual and automated test design.

4. Structure of a Precise Use Case (PUC)

The formulation of Precise Use Cases was greatly influenced by Cockburn's book on Use Cases [6], but violates several of his guidelines. It also draws on DeMarco's Structured English [8].

A *Precise Use Case* contains the following core information:

1. Case Id, Name and optional Abstract/Context Diagram

2. Risk Factors

Frequency of occurrence: [frequency range per time interval or event]

Impact of failure, likely & worst case [high, medium, low]

3. Case Conditions

Invariants and Pre-conditions

4. Courses of Action

Basic course

Performed course(s)

Alternative course(s)

Exception handler (EH)

Case-exiting utilities (CE)

Required selections (RS)

Optional actions (OA)

User-invoked interrupts (UI)

A Precise Use Case contains one *basic course* and zero or more *performed* and/or *alternative courses*. Additional information such as version, modeler, sources, duration, trigger events (e.g., specific times), undoing cases, or likely successors may also be included in the case. Additional candidates for inclusion can be found in [4] and [6].

A basic course description contains the following core information: (1) name, (2) course conditions i.e., invariant, pre, and post conditions, (3) inter-actors, and (4) interaction steps. Each usage scenario (i.e., complete path through a use case) begins at the first step of the basic course and must satisfy the case invariants and pre-conditions. Success scenarios, including those taking alternative courses, successfully exit the last step of the basic course.

A performed course is invoked with a “does <performed course name>” action and contains the following core information: (1) name, (2) course conditions, (3) inter-actors, and (4) interaction steps. Performed courses are used to simplify the basic course description as well as to encapsulate common course segments.

There are four types of alternative courses. All alternative course descriptions contain the following core information: (1) name, (2) insertion site location or range, (3) alternative course conditions, (4) inter-actors, and (5) interaction steps. The types of alternative courses include:

1. **Exception handlers** deal with abnormal conditions that block the successful completion of a system process. An exception condition is checked and if TRUE, the corresponding handler is invoked. The handler may compensate for the abnormal condition or take other steps to deal with the situation.
2. **Required selections** are elements of a set of two or more mutually exclusive alternative courses, one of which must be followed e.g., payment alternatives.
3. **Optional actions** are guarded by a set of guard conditions i.e., invariants and pre-conditions. If all the guards conditions are TRUE, then the option is included in the flow e.g., using discount coupons during checkout.
4. **User-invoked interrupts** are included goal-level use cases from the same application or a different application or utilities (e.g., help, save, print).

5. A sample PUC

The following example is derived from a specification appearing in [22]. Additional examples can be found in [10].

Case Name: Get [new] Seat on Reserved Flight

Risk Factors: Frequency of occurrence: 0 to 2 times per reservation

Impact of failure: *likely case* – **low**, open seating is a workaround

worst case – **medium**, open seating in a large plane with many expensive seats is likely to anger important passengers

Case Conditions:

Invariants: None

Preconditions: For reservation system, status is active

For passenger, system access status is signed on

Interactions

Basic course:

Passenger	Web-based Airline Reservation System
1. requests seat assignment	2. requests a reservation locator
3. provides a (corrected) reservation locator alternative	4. searches for reservation
Until (reservation located or all reservation locator strategies tried), repeat 3 to 4	
	5. offers seating alternatives, unless <ul style="list-style-type: none"> a. reservation not located or b. seat previously assigned or c. no seats are available or d. no seats are assignable
6. selects a seating alternative	7. assigns selected seat unless <ul style="list-style-type: none"> a. no seating alternative selected
	8. If (For reservation, seat previously assigned) <ul style="list-style-type: none"> returns previous seat to inventory <i>Post-conditions</i> -- For flight, previously assigned seat is available <ul style="list-style-type: none"> Endif
	9. confirms assignment SUCCESS EXIT

Basic course Conditions

Invariants:

For reservation system, status is active

For passenger, system access status is signed on

For passenger & flight, a reservation exists and can be located

Pre-conditions:

For flight, some seats are assignable

Passenger wants to get or change seat assignment

Post-conditions:

For flight, seating alternative was selected

For reservation, selected seat is assigned

*Alternative Courses:*Exception Handlers (EH):**EH 1** - 5a (reservation not located)

EH1 Invariants:

- For reservation system, status is active
- For passenger, system access status is signed on
- For passenger, reservation not located

Passenger	Web-based Airline Reservation System
	1. Offers help making reservation FAILURE EXIT

EH 2 - 5b (seat previously assigned)

EH2 Invariants:

- For reservation system, status is active
- For passenger, system access status is signed on
- For passenger & flight, a reservation exists and can be located
- For reservation, seat previously assigned

Passenger	Web-based Airline Reservation System
	1. offers to change seat assignment
2. wants a. to change seat assignment b. no change	3. If (Passenger wants to change seat assignment), CONTINUE Else, provides help and SUCCESS EXIT

EH3 – 5c or 5d (no seats are available or assignable)

EH3 Invariants:

- For reservation system, status is active
- For passenger, system access status is signed on
- For passenger & flight, a reservation exists and can be located
- For flight, no seats are assignable

Passenger	Web-based Airline Reservation System
	1. If (check in), places passenger on standby <i>Post-cond</i> -- For passenger, name on standby list Else, advises when assignment will be possible Endif FAILURE EXIT

EH4 - 7a (no seating alternative selected)**EH4 Invariants:**

For reservation system, status is active
 For passenger, system access status is signed on
 For passenger & flight, a reservation exists and can be located
 For flight, some seats are assignable
 For reservation, no seating alternative selected

Passenger	Web-based Airline Reservation System
	1. If (For reservation, seat previously assigned) SUCCESS EXIT Endif
	2. If (day of flight), advises to get assignment at check in Else, advises to try later Endif FAILURE EXIT

Figure 1. Example of the reserved seat functionality in a web-based airline reservation system

6. Comments on the example

The use of explicit exception conditions assures that each exception handler has a visible *point of origin*. A point of origin is marked by an individual exception condition preceded by the keyword *unless*. For example in step 7 of the basic course, some alternative must be selected before the system can assign a seat. An exception handler must deal with the situation when no seat is selected. A point of origin may be in any course of action. In the example above, steps 5 and 7 in the basic course contain points of origin, while the exception handlers do not.

Precise use cases encourage the explicit identification of **valid input alternatives**. When there are only a few simple alternatives, they may be provided directly within the step as they are in the second step of exception handler 2. Otherwise, the alternatives should be specified in an associated glossary of input alternatives. The alternatives referenced in step 3 of the basic course would be found in such a glossary.

7. Structured English in PUC courses

Structured English was originally developed to describe the actions of individual analysis entities [8]. The version used in PUCs is intended to describe interactions between multiple entities.

7.1. Courses of action

The types of courses that can appear in a PUC are listed below:

- **Basic course** -- each PUC has exactly one of these and every scenario begins at its first step.
- **Performed Course** -- refers to action sequences used to specify common or simplifying course segments.

- **Alternative Course** -- performed under specific conditions.
- **Exception Handling Macros** -- analogous to macros in programming and classes in object-oriented analysis. Macros are used to group exception handlers that have predictable differences. They use a common set of steps and handler specific information to generate a specific exception handler. Macros make the common structure of multiple handlers explicit.

7.2. Actions

<action> [, **unless** <exception conditions [N%]>]

An action has the structure <verb> <direct object [modifiers]> [<indirect object>]. See the many examples in this paper.

does <performed course name>

[**using** <parameter1>, <parameter2>, ...]

[, **unless** <exception conditions [N%]>]

selects/wants

a. <valid input alternative [N%]>

b. <valid input alternative [N%]>

In this paper, specific action names signal inputs, outputs, and state changes.

Inputs are signaled by: requests, provides, enters, selects, and wants

Outputs are signaled by: advises, approves, confirms, displays, prints, offers, requests, and provides

State changes are signaled by: accepts, deposits, places, records, returns, stores, and updates

The N% annotation of input alternatives and exception conditions is the (integer) probability that the alternative will be selected or the condition will be TRUE. The sum of probabilities of all alternatives must be 100%. The sum of probabilities of all exception condition must be less than 50%.

Examples:

- places passenger on standby
- approves drop request, **unless** course not added 1%
- **does** “adds a book” **using** “Writing Effective Use Cases”,
- **unless** book is already present
- **wants** aisle seat
- **selects**
 - a. credit or debt card 38%
 - b. sufficient cash or equivalent 62%

7.3. Handling intermediate failure

We try, we fail, we compensate or try something different, and we succeed. Intermediate failure is a normal component of successful goal-directed behavior. There are three basic approaches to handling such failures: (1) report, (2) compensate, and (3) retry using a different approach.

<action>, **unless** <exception conditions [N%]>

Exception handlers can effectively model reporting and compensating behavior. Exception conditions in an **unless** clause are Booleans that all must evaluate to FALSE in order for an action or performed course to be performed. Each exception condition has a corresponding handler that is invoked if its condition is the first to evaluate to TRUE. The exception handler may attempt to compensate for the condition and may succeed or fail in this attempt. If compensation is not attempted or the attempt fails, then following the termination style [15], the handler will record/report the condition, possibly do other things, but will always cause a FAILURE EXIT from the course containing its invoking action-unless statement i.e., the course raising the exception. If correction succeeds, then (following the resumption style [15],) control returns to the point it would have reached initially had the exception condition been FALSE. This permits the evaluation of conditions to continue.

does alternative strategies

```

<strategy1 course>
    [using <parameter1>, <parameter2>, ...]
<strategy2 course>
    [using <parameter1>, <parameter2>, ...]
[<failure preparation course>
    [using <parameter1>, ...]]
FAILURE EXIT

```

End

A different construct (i.e., does alternative strategies) is needed to model the use of heuristic alternatives that are expected to fail occasionally. This construct contains an ordered sequence of performed courses for different strategies that are performed until one strategy succeeds or the final FAILURE EXIT clause is executed. Each strategy course, except failure preparation, must contain a SUCCESS EXIT and may contain one or more FAILURE EXITS. The failure preparation course must not contain a SUCCESS EXIT.

7.4. Selection

```

If (<selection conditions [N%]>)
    <action> or does <...>
[Else
    <action> or does <... >]
[Endif]

```

Examples:

```

If ((customer offers credit or debit card)
and (payment amount is approved) 33%)
    accepts card payment
Endif

```

```

If (customer rejects all card payments) does "handles cash payment",
    unless cash or equivalent amount is insufficient
Endif

```

```

    If (check in time and no assignable seat 2%),
        places passenger on standby
    Else
        advises when assignment will be possible
    Endif

```

Avoid directly embedding *if* statements within *if* statements. The embedded *if* should be placed in a performed course.

7.5. Repetition

Until or **While** <conditions> or **For each** <item name>,
 repeat <step range>
 or **repeat** <performed course name>

The step range must be immediately above or below the repetition statement.

Repetition must always encompass both inter-actor and system actions.

3. provides a (corrected) reservation locator alternative	4. searches for reservation
Until (reservation located or all reservation locator strategies tried), repeat 3 to 4	

Figure 2. Example of using a repetition construct

7.6. Scenario flow

Each usage scenario starts with the ENTER in the basic course and ends with an EXIT. The EXITs may be SUCCESS or FAILURE.

CONTINUE means a return to the invoking course. If all conditions in the set of conditions (e.g., exception conditions) that triggered the CONTINUEing course have not been evaluated, then evaluate the unevaluated conditions, otherwise execute the next step.

ITERATE only appears in steps that are being performed within a repeat loop. It means that the flow should continue at the repeat statement and proceed normally depending on the evaluation of the repetition conditions.

7.7. Ambiguity of order in a step

For the following action:

- inter-actor provides
 - a. name
 - b. address
 - c. phone number

the order in which inputs are provided is **not** specified.

For the step actions: “requests checkout and provides items”, the order of actions is **not** specified. If order is important, write multiple steps.

For the following action:

- system approves add request, **unless**
 - a. course is full
 - b. prerequisites are not satisfied
 - c. conflict in schedule
 - d. course load is unacceptable

the order in which exception conditions are evaluated is **not** specified. Therefore, if multiple conditions are TRUE, any of the corresponding exception handlers might be invoked.

8. Alternative formats for courses

The activity within courses can be in one-column, two-column, or three-column format. In a **one-column format**, the actions of all inter-actors and the system, appear in a single stream. Therefore, to avoid confusion, each action description must explicitly name its associated inter-actor or system.

The example above shows a **two-column format**. This is useful for describing dialogs between one or more inter-actors and a fully automated system. System actions appear in the second column and the actions of all inter-actors appear in the first. All inter-actors are named in the heading of the first column. If there are multiple inter-actors, each action in the first column should explicitly name its inter-actor.

A **three-column format** is useful for describing trialogs i.e., when inter-actors interact with a combination of technology and people. For example, at an airport or supermarket, we find ticket agents and cashiers as well as technology. The people and the technology are inside the system from the perspective of the inter-actor. As with the two-column format, if there are multiple inter-actors or multiple people or both, they should all be named in the headings and explicitly named with the actions. This format should not be used for multiple inter-actors and a fully automated system.

9. PUC modeling guidelines

- a) A use case may require access to specific resources (e.g., document files or applications) to be successful and an inter-actor may be able to enable this access (e.g., by opening a file or signing onto an application). In this situation, define an optional action to be inserted at the start of the use case that specifies resource enabling if the resources are not accessible e.g. inter-actor log on.

Using this approach removes most dependencies between use cases, except for multiple cases that reference the same application entity. For example, if one use case creates a document named “bob” and the next one tries to create a document with the same name, the inter-actor should get a different response than if the first use case had not occurred.

- b) Sometimes, there may be confusion in choosing between an “<action>, **unless**” and an “if (<selection condition>)”. The guidelines are:
 - 1. when an action is always to be done, unless some exception condition is TRUE, use an

“action, **unless**” . An exception condition is one that is **intended** to be exceptional. Sometimes it may actually be the rule, e.g., the printer may usually be inoperative.

2. when an action is to be done sometimes, but not other times, use an “if (<selection condition>)” e.g., for adding some courses and deleting others use an if.

- c) Model the flow of information, not user interface design nor navigation details. Use input actions like - enters, provides, or requests rather than fills in a form, clicks a button, or links to a specific web page. See the discussion of essential use cases in [4].

Additional recommendations can be found in [11].

10. A development process for suites of PUCs

PUCs are too detailed for effective use during early interactive elicitation. Their development can be characterized as the descriptive (as opposed to prescriptive) programming of interactive processes. Therefore, development of PUCs should start after some requirements have already been captured using a less formal style of use case [26]. Initial forms can be as informal as the user stories used in XP development [1] or any of the styles described in [6] or [21]. PUC development and informal elicitation can co-occur.

The initial PUCs should be developed by stakeholder teams. This precise modeling process will expose misunderstandings, disagreements, and confusion. After a solid foundation of common understanding has been established, PUCs can be developed “off line” by an analyst or tester, but they will require stakeholder support to answer the inevitable questions and to review the results. The issues uncovered during PUC development, if uncovered before system design, could be PUC’s most important contribution to quality.

A suite of use cases can be developed as follows:

1. Identify system, clarify scope, and list functions
2. **Identify inter-actors and relationships**
3. **Identify goals of primary inter-actors**
4. Identify domain and system entities (e.g., an order) associated with the goals
5. **Develop use cases for [critical] goals**
6. Factor *common operations* into their own cases and *similar cases* into parameterized cases
7. Organize the suite

Identify inter-actors with the following questions:

- For the new system or change:
- Who/What directly benefits?
- Who/What is directly impacted?
- Which services are needed to enable successful functioning and who/what supplies them?

Identify goals of the primary inter-actors as follows:

- Identify those directly helped by the proposed system or change (these are the primary inter-actors)
- Identify primary inter-actor job responsibilities, with priorities

- Identify strategies for carrying out the relevant responsibilities
- Identify goals (and sub-goals) associated with the selected strategy
- For each goal, identify which inter-actors share the goal; when, how often, and why the goal is important; and what happens if the goal is not met

Develop a use case as follows:

1. Create an id and goal-based title
2. Write a minimal basic course
3. Add exception conditions to the course
4. Develop a rich set of inter-actor friendly alternative courses
5. Factor similar alternatives into course macros
6. To shorten a course, factor out a group of steps into another course or into a use case (operation-level)
7. **Add course conditions (invariant, pre, and post) to each course and macro**
8. Add supplemental info to the case

Identify course conditions as follows:

Post-conditions & Invariants

- For each step, ask: What conditions caused by this step will always be TRUE at the end of the course?
- Identify any invariants among the post-conditions
- For each variable in a post-condition, identify any relevant invariants for that variable

Pre-conditions & Invariants

- For each step, ask: What conditions must be TRUE from the beginning of the course to enable this step?
- Identify any invariants among the pre-conditions
- For each variable in a pre-condition, identify any relevant invariants for that variable

Review the suite of PUCs with stakeholders and subject matter experts and modify as needed

11. Test coverage criteria for PUCs

To describe test coverage, we need a few definitions. *Single-goal use cases* accomplish one and only one user goal. **Multiple-goal** cases accomplish two or more user goals. **Transitional use cases** cause entity state transitions e.g., returns a book, while **non-transitional cases** do not e.g. display copies borrowed. An ordered pair of transitional cases referencing the same entity may be valid or invalid. For example, a customer record can only be updated, after it is created.

For a set of PUCs, a test suite should cover:

- every (valid) ordered pair of single-goal cases where one or both are non-transitional
- every (valid & invalid) ordered pair of transitional cases that reference the same entity

- every multiple-goal case

If this is too expensive, then either cover every ordered pair of “high or medium risk” use cases or just cover every ordered pair of “high risk” use cases. In either case, cover every multiple-goal case.

Within a single-goal case, a test suite should cover every step in the basic course, performed courses, and alternative courses.

A use case will normally contain multiple scenarios i.e., complete execution paths, because it contains repetition cycles and alternative courses. A scenario may have multiple interpretations, because it contains input alternatives and derived conditions for inputs (e.g., invalid) or states (e.g., unavailable). There are usually many distinct ways to interpret a derived condition e.g., many distinct ways to be invalid or state.

For a use case, the test suite should cover every:

1. repetition cycle 0 (if possible), 1, and an even number of times
2. unique cause + 1 interpretation [13] of selection, repetition, alternative course, and derived conditions
3. input alternative

In addition, the test suite should cover:

1. output boundaries
2. valid and invalid input boundary values [18].

If one or more of these criteria causes the test suite to be “too big” (i.e., unaffordable because criteria are too strong), risk information must be used to weaken or delete criteria.

Consider adding the “every pair of input partition values” [28] criterion, if you have an adequate budget surplus and its incremental cost-effectiveness has been shown.

12. Related work

Others have proposed adding rigor to usage descriptions. Proposals have entailed Message Sequence Charts and Process Algebra [2], Abstract State Machine Language [14], Activity Diagrams [20], and Modular Petri Nets [29]. These alternatives, however, do not present the application choices, essential to effective test design, as clearly and completely as PUCs. More importantly, these other approaches have not evolved from informal forms of the same technique. Effective informal forms make transition to formality much easier.

The Object Constraint Language [27] has been proposed for specifying conditions, but does not satisfy the understandability requirement.

13. Acknowledgements

This work owes an enormous debt to Alistair Cockburn’s book [6]. I am also grateful to Ian Alexander, James Bach, Sofia and Stanislov Passova and Natan Aronshtam for helpful comments.

14. References

- [1] Extreme Programming info available at www.extremeprogramming.org/rules/userstories.html
- [2] Andersson, Michael and Bergstrand, Johan “**Formalizing Use Cases with Message Sequence Charts**” MS Thesis, Lund Institute of Technology May 1995 Available at www.efd.lth.se/~d87man/EXJOB/BB/Formal_Approach_to_UC.html
- [3] Armour, Frank and Miller, Granville **Advanced Use Case Modeling** Addison Wesley 2001
- [4] Constantine, Larry and Lockwood, Lucy “**Structure and Style in Use Cases for User Interface Design**” Available at www.foruse.com/Resources.htm#style
- [5] Constantine, Larry and Lockwood, Lucy **Software for Use** ACM Press Addison Wesley 1999
- [6] Cockburn, Alistair **Writing Effective Use Cases** Addison-Wesley 2001
- [7] Cockburn, Alistair “**Use Cases, Ten Years Later**” STQE, SQE, Vol. 4, No. 2, March/April 2002
- [8] DeMarco, Tom **Structured Analysis and System Specification** Prentice-Hall 1978
- [9] Gelperin, David “**Precise Usage Model for Library Management System**” Available at www.LiveSpecs.com
- [10] Gelperin, David “**Precise Use Case Examples**” Available at www.LiveSpecs.com [also included as an appendix]
- [11] Gelperin, David “**Modeling Alternative Courses in Detailed Use Cases**” Available at www.LiveSpecs.com
- [12] Gelperin, David “**Specifying Consequences with Action Contracts**” Available at www.LiveSpecs.com
- [13] Gelperin, David “**Testing Complex Logic**” Available at www.LiveSpecs.com
- [14] Grieskamp, Wolfgang, Lepper, Markus, Schulte, Wolfram, and Tillmann, Nikolai “**Testable Use Cases in the Abstract State Machine Language**” in *Proceedings of Asia-Pacific Conference on Quality Software (APAQS'01)*, December 2001.
- [15] Garcia, Alessandro F., Rubira, Cecilia M. F., Romanovsky, Alexander, Xu, Jie. “**A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software**”. *Journal of Systems and Software*, Elsevier, Vol. 59, Issue 2, November 2001, pp. 197-222.
- [16] Ivar Jacobson et al *Object-Oriented Software Engineering* Addison-Wesley 1992
- [17] Jungmayr, Stefan and Stumpe, Jens “**Another Motivation for Usage Modeling: Generation of User Documentation**” *Proceedings of CONQUEST '98*, Nuernberg, Germany, September 28-29, 1998

- [18] Kaner, Cem, Falk, Jack, Nguyen, Hung Quoc **Testing Computer Software** John Wiley 1999
- [19] Kulak, Daryl and Guiney, Eamonn **Use Cases: Requirements in Context** ACM Press Addison-Wesley 2000
- [20] Kisters, Georg, Six, Hans-Werner, and Winter, Mario “**Coupling Use Cases and Class Models as a Means for Validation and Verification of Requirements Specifications**” Requirements Engineering Journal, Vol. 6, No. 1 2001 pp 3-17
- [21] Larman, Craig **Applying UML and Patterns** Prentice-Hall PTR 2002
- [22] Rumbaugh, James “**Getting Started : Using Use Cases to Capture Requirements**” Journal of OO Programming, SIGS Publications, Vol. 7, No. 5, Sept 1994 pp 8-10, 12, & 23
- [23] Runeson, Per and Regnell, Bjorn “**Derivation of an integrated operational profile and use case model**” 9th Symposium on SRE IEEE press Nov. '98 pp. 70-79
- [24] Schneider, Geri and Winters, Jason P. **Applying Use Cases** Addison Wesley 1998
- [25] Thelin, Thomas, Runeson, Per, and Regnell, Bjorn “**Usage-based reading – an experiment to guide reviewers with use cases**” Information and Software Tech. 43 (2001) pp. 925-938
- [26] Wiegers, Karl E. “**Listening to the Customer’s Voice**” Software Development, March 1997
- [27] Warmer, Jos and Kleppe, Anneke. **The Object Constraint Language** Addison Wesley 1999
- [28] Williams, Clay and Paradkar, Amit “**Efficient Regression Testing of Multi-Panel Systems**” IEEE International Symposium on Software Reliability Engineering, Nov. 1999
- [29] Woo, Jin Lee, Sung, Doek Cha, and Yong, Rae Kwon “**Integration and Analysis of Use Cases Using Modular Petri Nets in Requirements Engineering**” IEEE Transactions on Software Engineering, Vol. 24, No. 12, December 1998

Appendix: Additional PUC Examples

This appendix contains a substantial use case example that includes a performed course, two optional action courses, and exception handling macros. An example of the three-column, trialog format is also included.

1. An Example with Performed and Optional Action Courses, as well as Exception Handling Macros

Case Name: Build a Course Schedule

Risk Factors: Frequency of occurrence: 1 to 5 (1.3 avg.) times per student per quarter

Impact of failure:

likely case -- high

worst case -- high

Case Conditions:

Invariants:

None

Preconditions:

University Web site is active

For student, system access status is signed on

Interactions

Basic course:

Student	University Web Site
ENTER 1. requests "Create a Schedule"	
	2. If (For student, a schedule already exists 23%) displays existing schedule Else displays new schedule Endif
	3. displays list of available offering from the Course Catalog System, unless course registration is closed 8% or Course Catalog System is unavailable 3%
For each course request, actors repeat do "Add or Drop Courses"	
	4. If (initial schedule has changed 80%) stores status, displays and prints schedule and schedule confirmation number Else displays invitation to return until registration close date Endif SUCCESS EXIT

Success course conditions

Invariants:

- University Web site is active
- Course registration is open
- For Course Catalog system, status is available

Preconditions:

None

Post-conditions:

- For student, system access status is signed on
- For student, a schedule exists
- For student, schedule, and course requests,
 - all approved requests are recorded, displayed, and printed along with associated confirmation number
 - all disapproved requests are explained

Performed Courses:

“Add or Drop Courses” (ADC):

ADC Invariants:

- University Web site is active
- For student, system access status is signed on course registration is open
- For Course Catalog system, status is available
- For student, a schedule exists

Student	University Web Site
1. requests course be <ul style="list-style-type: none"> a. added 88% b. dropped 12% 	
	2. If (add request) <ul style="list-style-type: none"> approves add request, unless <ul style="list-style-type: none"> a. course is full 18% b. prerequisites are not satisfied 7% c. conflict in schedule 2% d. course load is unacceptable 3% Else (drop) <ul style="list-style-type: none"> approves drop request, unless <ul style="list-style-type: none"> e. course is not in schedule 2%
	3. displays approved adds & drops as well as

ADC Post-conditions:

- For student, schedule, and course request, request is approved or disapproved with reasons

Alternative Courses:

Optional Action

Log On after basic course ENTER

Log On Invariants:

University Web site is active

Log On Preconditions:

For student, system access status is signed off

Student	University Web Site
1. requests logon	2. requests logon info
3. provides logon info	4. If (logon info ok) displays welcome message <i>Post-cond</i> – For student, system access status is signed on Else requests correct logon info Endif
Until (logon info ok or three unsuccessful tries), actors repeat 3 to 4	
	5. If (logon info ok) CONTINUE Else FAILURE EXIT Endif

Optional Action

Quit Registration

Quit Registration Invariants:

University Web site is active

Quit Registration Preconditions:

For student, system access status is signed on

Student	University Web Site
1. requests "Quit"	2. offers to save registration status
3. chooses a. save 93% b. no save 7%	4. If (save selected) stores status <i>Post-cond</i> – For student, approved courses, disapproved requests with reasons, and unprocessed requests are stored Endif
	5. offers to continue, instead of quit
6. chooses a. continue 7% b. quit 93%	
	7. If (continue selected) CONTINUE <i>Post-cond</i> – For student, system access status is signed on Else SUCCESS EXIT <i>Post-cond</i> – For student, system access status is signed off Endif

Exception Handlers:

Exception Handling Macro 1 – Displays exception condition message

EHM1 Invariants: For system, <exception condition>

EHM1 Preconditions: None

Student	University Web Site
	1. displays <exception condition> message FAILURE EXIT

EHM1 Post-conditions:

For system, <exception condition> message is displayed

EHM1 applies to:

EH1 – SC 3a: course registration is closed

EH2 – SC 3b: catalog system is unavailable

Exception Handling Macro 2 – Add request disapproved

EHM2 Invariants: For student, schedule, and add request, <exception condition>

EHM2 Preconditions: For student and schedule, add request is pending

Student	University Web Site
	1. records request denial and displays <exception condition> message CONTINUE

EHM2 Post-conditions: For student and schedule add request is disapproved

EHM2 applies to:

EH3 - ADC 2a: course is full

EH4 - ADC 2b: prerequisites are not satisfied

EH5 - ADC 2c: conflict in schedule

EH6 - ADC 2d: course load is unacceptable

Exception Handler 7 - ADC 2e

EH7 Invariants: For student, schedule, and drop request, requested course is not in schedule

EH7 Preconditions: For student and schedule drop request is pending

Student	University Web Site
	1. records request denial and displays “only approved courses can be dropped” message CONTINUE

EH7 Post-conditions: For student and schedule drop request is disapproved

This example is derived from the example on page 123 in Alistair Cockburn’s **Writing Effective Use Cases** (Addison-Wesley 2001).

2. Example of Three-Column Course Format

Case Name: Checkout items

Risk Factors:Frequency of occurrence: 10 to 200 per hour

Impact of failure:

likely case – **low**, checkout at another station

worst case – **high**, all stations down

Case Conditions

Invariants:

For system, a clerk available

Preconditions:

For customer, items to be checked out

For system, POS subsystem status is operational

Interactions

Basic course:

Customer	Checkout System	
	Clerk	POS subsystem
ENTER 1. requests checkout and provides items		
	Until all items are entered, actors repeat 2 to 4	
	2. enters next item a. by scanning b. manually	3. displays line-item details
		4. displays running total
	5. requests payment	
6. chooses a. credit or debt card b. cash or equivalent		7. If (credit or debit card is offered and payment amount is approved) accepts card payment Endif
	8. If (no card payment is accepted) accepts and deposits cash or equivalent, unless a. amount is insufficient Endif	9. records sale
		10. updates inventory
		11. prints receipt SUCCESS EXIT

Basic course Conditions

Invariants:

For system, a clerk available

For system, POS subsystem status is operational

Preconditions:

For customer, items to be checked out

For purchase amount, card payment or cash or cash equivalent is sufficient

Post-conditions:

For customer transaction, sale of all desired items is recorded and receipt is printed

For sold items, inventory updated

Click on ad to reach advertiser web site

Now with support for UML 2.0, Enterprise Architect is your intuitive, flexible and powerful UML analysis and design tool for building robust and maintainable software. From requirements gathering, through the analysis, design models, implementation, testing, deployment and maintenance, Enterprise Architect is a fast, feature-rich multi-user UML modelling tool, driving the long-term success of your software project.

<http://www.sparxsystems.com.au?source=Methodsandtools>

European SEPG Conference 14th-17th June 2004 London. PEOPLE - PROCESS - TECHNOLOGY - MEASUREMENT - PROJECT MANAGEMENT. The European SEPG conference is the premier process improvement event in Europe, where leaders, innovators and practitioners meet to explore methods, tools and process improvements that aim to increase business performance through quality and productivity gains. The conference provides practical assistance to those involved in or planning improvement in the fields of People, Process, and Technology. Measurement and Project Management Symposia will run parallel with the main programme on the first two days of the conference. The European SEPG is the annual showcase for CMMIR, it is a joint venture between ESPI Foundation and the Software Engineering Institute, Carnegie Mellon University, Pittsburgh.

<http://www.espi.org>

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

<http://www.adminitrack.com>

XMLBooster: the fastest XML parsing solution available today. It statically generates application-specific parsers, which are 5 to 50 times faster than generic parsers, such as DOM or SAX, and which require 60 to 90% less memory. It supports various languages: Java, C,C#, C++, COBOL, Ada. XMLBooster GUI also generates Swing-based GUIs.

<http://www.xmlbooster.com>

Confluence, the professional wiki, is designed to make it easy for a team to share information with each other and with the world - all in a single web-based location. Includes features rarely seen in wiki software: security, blogging, refactoring, HTML/PDF exporting, a remote API, easy installation. As used by Boeing, Telenor Networks, Thoughtworks and many more! Read more, try Confluence online or download an evaluation for your organisation today:

<http://www.atlassian.com/confluence/>

Merant PVCS customers, are you confused about recent changes?
Looking for a single vendor committed to SCM and outstanding customer service? Learn how easy change can be with MKS - the proven choice for Enterprise SCM. Certegy Inc. recently moved from Merant PVCS to MKS and was certified CMM Level 2 in just six months time.

<http://www.mks.com/go/changeiseasytm>

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

<http://www.red-gate.com/dotnet/summary.htm>

Advertising for a new Web development tool? Looking to recruit software developers? Promoting a conference or a book? Organizing software development training? This space is waiting for you at the price of US \$ 30 each line. Reach more than 31'000 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 7000 visitors/month of our web sites! To advertise in this section or to place a page ad in a PDF issue simply go to

<http://www.methodsandtools.com/advertise.html>

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 1023-4918

Free subscription on : <http://www.methodsandtools.com/forms/submt.php>

The content of this publication cannot be reproduced without prior written consent of the publisher

Copyright © 2004, Martinig & Associates