# METHODS & TOOLS

**Global knowledge source for software development professionals** **ISSN 1023-4918**
Spring 2000 (Volume 8 - number 1)

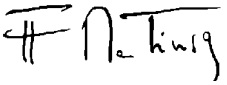
## Post-Y2K Life in Software Development

We have passed the New Year and the February 29 date and no big bug has been signaled. Have we achieved this situation because we have been too alarming about the state of your software portfolios or have we done a good job correcting all bad programs? Surely both answers are correct.

Now that the Y2K problem is mostly behind us, the question is what are we going to do? The current marketing noise is about "B2B e-commerce infrastructure", with new companies like Ariba or Broadvision gaining major contracts and old players like Oracle or SAP shifting their emphasis towards this area. It is sure that Internet is the next big thing and that most of the new projects will be involved with the redeployment of enterprise software on the Web platform, including the new "handheld" phenomena represented by the Palm Pilot or the WAP protocol for Internet access through mobile phones.

This means a strong demand for HTML, XML, Java or ColdFusion developers, but I do not think that this is the end of the need for traditional software development skills. If the software interfaces with customers will be changed in many organizations, the core internal processing software will continue to work on the same technologies. Mainframes, ERP and databases will still be at the heart of information systems. The challenge will be to integrate the new customer relationship management software with the traditional software components like invoicing or accounting. This is where infrastructure software and middleware standards like Corba will play a big role.

So if you have the chance to be trained in a new technology, do not hesitate, but do not panic if you have traditional skills, their value has not disappeared after the Y2K passage.



## Inside

Practical Experience in Automated Testing..... page 2

Understanding Use Case Modeling..... page 10

GUI Testing Checklist.....page 17

Facts, News & Comments .....page 27

# Practical Experience in Automated Testing

Kerry Zallar, zallar@testingstuff.com
http://www.testingstuff.com

## Automated Software Testing - A Perspective

Note from the author: My perspective on most things is that the 'glass is half full' rather than half empty. This attitude carries over to the advice I suggest on automated software testing as well. I should point out, however, there is an increasing awareness from others experienced in this field, as well as from my own experience, that many efforts in test automation do not live up to expectations. A lot of effort goes into developing and maintaining test automation, and even once it's built you may or may not recoup your investment. It's very important to perform a good cost/benefit analysis on whatever manual testing you plan to automate. The successes I've seen have mostly been on focused areas of the application where it made sense to automate, rather than complete automation efforts. Also, skilled people were involved in these efforts and they were allowed the time to do it right.

Test automation can add a lot of complexity and cost to a test team's effort, but it can also provide some valuable assistance if its done by the right people, with the right environment and done where it makes sense to do so. I hope by sharing some pointers that I feel are important that you'll find some value that translates into saved time, money and less frustration in your efforts to implement test automation back on the job.

## Key Points

I've listed the 'key points' up front instead of waiting until the end. The rest of the article will add detail to some of these key points.

- First, it's important to define the purpose of taking on a test automation effort. There are several categories of testing tools each with its own purpose. Identifying what you want to automate and where in the testing life cycle will be the first step in developing a test automation strategy. Just wishing that everything should be tested faster is not a practical strategy. You need to be specific.

- Developing a test automation strategy is very important in mapping out what's to be automated, how it's going to be done, how the scripts will be maintained and what the expected costs and benefits will be. Just like every testing effort should have a testing strategy, or test plan, so should there be a 'plan' built for test automation.

- Many of the testing 'tools' provided by vendors are very sophisticated and use existing or proprietary coding 'languages'. The effort of automating an existing manual testing effort is no different than a programmer using a coding language to write programs to automate any other manual process. Treat the entire process of automating testing as you would any other software development effort. This includes defining what should be automated, (the requirements phase), designing test automation, writing the scripts, testing the scripts, etc. The scripts need to be maintained over the life of the product just as any program would require maintenance. Other components of software development, such as configuration management also apply

- The effort of test automation is an investment. More time and resources are needed up front in order to obtain the benefits later on. Sure, some scripts can be created which will provide immediate payoff, but these opportunities are usually small in number relative to the effort of automating most test cases. What this implies is that there usually is not a positive payoff for automating the current release of the application. The benefit comes from running these automated tests every subsequent release. Therefore, ensuring that the scripts can be easily maintained becomes very important.

- Since test automation really is another software development effort, it's important that those performing the work have the correct skill sets. A good tester does not necessarily make a good test automator. In fact, the job requirements are quite different. Good testers are still necessary to identify and write test cases for what needs to be tested. A test automator, on the other hand, takes these test cases and writes code to automate the process of executing those tests. From what I've seen, the best test automation efforts have been lead by developers who have put their energies into test automation. That's not to say that testers can't learn to be test automators and be successful, it's just that those two roles are different and the skill sets are different.

**Points**

Here are some other important points to consider:

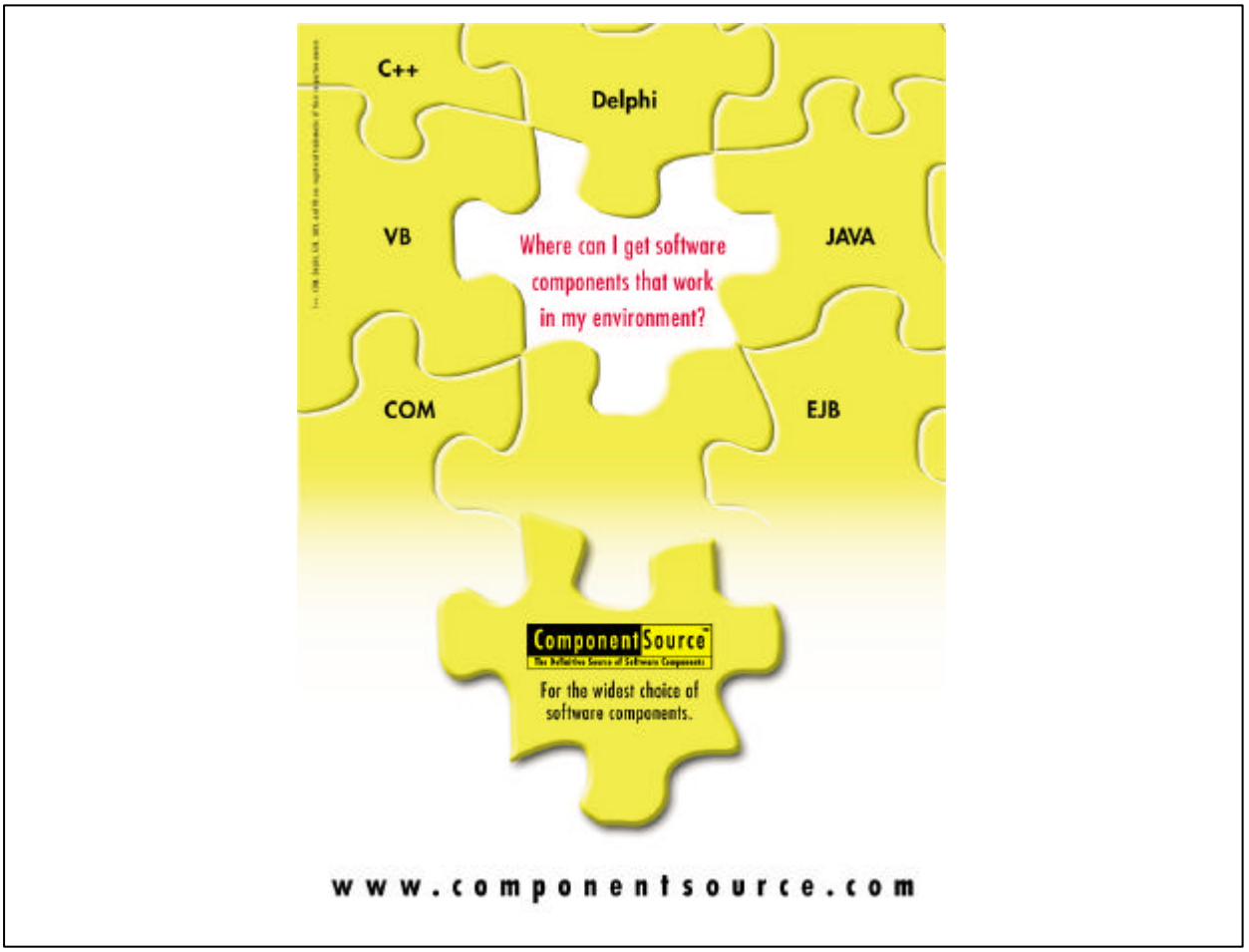
When strategizing for test automation, plan to achieve small successes and grow. It's better to incur a small investment and see what the effort really takes before going gung ho and trying to automate the whole regression suite. This also gives those doing the work the opportunity to try things, make mistakes and design even better approaches.

Many software development efforts are underestimated, sometimes grossly underestimated. This applies to test automation as well, especially if the effort is not looked upon as software development. Test automation is not something that can be done on the side and care should be taken when estimating the amount of effort involved. Again, by starting small and growing, estimating the work can be gauged.

When people think of testing tools, many first think of the 'capture/playback' variety where the application is tested at the end during system test. There are several types of testing tools which can be applied at various points of code integration. Test automation can be applied at each of the levels of testing including unit testing, one or more layers of integration testing, and system testing (another form of integration). The sooner tests can be executed after the code is written, before too much code integration has occurred, the more likely bugs will not be carried forward. When strategizing for test automation, consider automating these tests as early as possible as well as later in the testing life cycle.

Related to this last point is the idea that testers and software developers need to work as a team to make effective test automation work. I don't believe testing independence is lost when testers and developers work together, but there can be some excellent advantages that I'll later point out.

Testing tools, as sophisticated as they have become, are still dependent upon consistency in the test environment. This should be quite obvious, but having a dedicated test environment is absolutely necessary. If testers don't have control of their test environment and test data, the required setup for tests may not meet the requirements of those tests. When manual testing is done testers may sometimes 'work around' test setup issues. Automated test scripts are less flexible and require specific setup scenarios, thereby needing more control.




Test automation is not the only answer to delivering quality software. In fact, test automation in many cases is a last gasp effort in an attempt to find problems after they've been made instead of eliminating the problems as they are being created. Test automation is not a substitute for walkthroughs, inspections, good project management, coding standards, good configuration management, etc. Most of these efforts produce higher pay back for the investment than does test automation. Testing will always need to be done and test automation can assist, but it should not be looked upon as the primary activity in producing better software.

The truth is that developers can produce code faster and faster with more complexity than ever before. Advancements in code generation tools and code reuse are making it difficult for testers to keep up with software development. Test automation, especially if applied only at the end of the testing cycle, will not be able to keep up with these advances. We must pull out all stops along the development life cycle to build in good quality software and test as early and often as possible with the assistance of test automation.

**Benefits**

To many people, the benefits of automation are pretty obvious. Tests can be run faster, they're consistent, and tests can be run over and over again with less overhead. As more automated tests are added to the test suite more tests can be run each time thereafter. Manual testing never goes away, but these efforts can now be focused on more rigorous tests.

There are some common 'perceived' benefits that I like to call 'bogus' benefits. Since test automation is an investment it is rare that the testing effort will take less time or resources in the current release. Sometimes there's the perception that automation is easier than testing manually. It actually makes the effort more complex since there's now another added

software development effort. Automated testing does not replace good test planning, writing of test cases or much of the manual testing effort.

**Costs**

Costs of test automation include personnel to support test automation for the long term. As mentioned, there should be a dedicated test environment as well as the costs for the purchase, development and maintenance of tools. All of the efforts to support software development, such as planning, designing, configuration management, etc. apply to test automation as well.

**Common View**

Now that some of the basic points have been noted, I'd like to talk about the paradigm of testing automation. When people think of test automation, the 'capture/playback' paradigm is commonly perceived. The developers create the application software and turn it over to the testing group. The testers then busily use capture/playback functionality of the testing tool to quickly create test scripts. Capture/playback is used because it's easier than 'coding' scripts. These scripts are then used to test the application software.

There are some inherent problems with this paradigm. First, test automation is only applied at the final stage of testing when it is most expensive to go back and correct the problem. The testers don't get a chance to create scripts until the product is finished and turned over. At this point there is a tremendous pull on resources to just test the software and forgo the test automation effort. Just using capture/playback may be temporarily effective, but using capture/playback to create an entire suite will make the scripts hard to maintain as application modifications are made.

**Test and Automate Early**

From observations and experience, a different paradigm appears to be more effective. Just as you would want to test early and test often if you were testing manually, the same applies to test automation. The first level of testing is the unit testing performed by the developer. From my experience unit testing can be done well or not done well depending on the habits and personality of the developer. Inherently, developers like to develop, not write test cases. Here's where an opportunity for developers and testers to work together can begin to pay off. Testers can help document unit tests and developers can write utilities to begin to automate their unit tests. Assisting in documenting test cases will give a better measurement of unit tests executed. Much success of test automation comes from homegrown utilities. This is because they integrate so well with the application and there is support from the developer to maintain the utilities so that they work with the application. More effective and efficient unit testing, through the use of some automation, provides a significant bang for the buck in trying to find bugs in the testing life cycle. Static analyzers can also be used to identify which modules have the most code complexity and may require more testing.

**Work With Developers**

The same approach should be applied at each subsequent level of testing. Apply test automation where it makes sense to do so. Whether homegrown utilities are used or purchased testing tools, it's important that the development team work with the testing team to identify areas where test automation makes sense and to support the long-term use of test scripts.

Where GUI applications are involved the development team may decide to use custom controls to add functionality and make their applications easier to use. It's important to determine if the testing tools used can recognize and work with these custom controls. If the testing tools can't work with these controls, then test automation may not be possible for that part of the application. Similarly, if months and

months of effort went into building test scripts and the development team decides to use new custom controls which don't work with existing test scripts, this change may completely invalidate all the effort that went into test automation. In either case, by identifying up front in the application design phase how application changes affect test automation, informed decisions can be made which affect application functionality, product quality and time to market. If test automation concerns aren't addressed early and test scripts cannot be run, there is a much higher risk of reduced product quality and increased time to market.

Working with developers also promotes building in 'testability' into the application code. By providing hooks into the application testing can sometimes be made more specific to any area of code. Also, some tests can be performed which otherwise could not be performed if these hooks were not built.

Besides test drivers and capture/playback tools, code coverage tools can help identify where there are holes in testing the code. Remember that code coverage may tell you if paths are being tested, but complete code coverage does not indicate that the application has been exhaustively tested. For example, it will not tell you what has been 'left out' of the application.

**Capture/Playback**

Here's just a note on capture/replay. People should not expect to install the testing tool, turn on the capture function and begin recording tests that will be used forever and ever. Capturing keystrokes and validating data captured within the script will make the script hard to maintain. Higher level scripts should be designed to be modular which has options to run several tests scripts. The lower level test scripts that actually perform tests also should be relatively small and modular so they can be shared and easily maintained. Data for input should not be hard coded into the script, but rather read from a file or spreadsheet and loop through the module for as many times as you wish to test with variations of data. The expected results should also reside in a file or spreadsheet and read in at the time of verification. This method considerably shortens the test script making it easier to maintain and possibly reuse by other test scripts. Bitmap comparisons should be used very sparingly. The problem with bitmap comparison is that if even one pixel changes in the application for the bitmap being compared, the image will compare as a mismatch even if you recognize it as a desirable change and not a bug. Again, the issue is maintainability of the test suite.

Capture/playback functionality can be useful in some ways. Even when creating small modular scripts it may be easier to first capture the test then go back and shorten and modify it for easier maintenance. If you wish to create scripts that will obviously provide immediate pay back, but you don't care if it's maintainable, then using capture/playback can be a very quick way to create the automated test. These scripts typically are thrown away and rebuilt later for long term use. The capture/playback functionality is also good to use during the design phase of a product if there's a prototype developed. During usability testing, which is an application design technique, users sit at the computer using a mock up of the actual application where they're able to use the interface, but the real functionality has not yet been built. By running the capture/playback tool in capture mode while the users are 'playing' with the application, recorded keystrokes and mouse movements can track where the users move on the system. Reading these captured scripts help the designers understand the level of difficulty in navigating through the application.

**Players**

Test automation is not just the responsibility of the testers. As noted, getting developers involved is important as well as getting the understanding and support of management. Since test automation is an investment, it's important that they understand the up front costs and expected benefits so that test automation stays around long enough to show the benefits.

There is the tendency to 'give up' when results are not shown right away.

If the project is just beginning with test automation then having someone who can champion the test automation effort is important. This 'champion' should have skills in project management, software testing and software development (preferably a coding background). This 'champion' is responsible for being the project manager of the test automation effort. This person needs to interact well with both the testers and the application developers. Since this person may also be actively involved with writing scripts as well, good development skills are also desirable. This person should not be involved with the designing of test cases or manual testing other than to review other team member's work. Typically there is not enough time to both design test cases and design test automation. Nor is there time to build test scripts and run manual tests by the same person. Where the testing effort is large the distinction between these two roles apply to teams of automators and testers as well. Too many times test automators are borrowed to performed manual testing never to realize the benefits of test automation in the current or future releases of the application.

This is not to say that the role of testers is reduced. Test planning still needs to be done by a test lead, test cases still need to be designed and manual testing will still be performed. The added role for these testers is that they most likely will begin to run the automated test scripts. As they run these scripts and begin to work more closely with the test automation 'champion' or test automators, they too can begin to create scripts as the automated test suite matures.

Experience has shown that most bugs are not found by running automated tests. Most bugs are found in the process of creating the scripts, or the first time the code is tested. What test automation mostly buys you is the opportunity to not spend valuable man-hours re-testing code that has been tested before, but which has to be tested in any case because the risk is too high not to test it. The other benefit comes from the opportunity to spend these man-hours rigorously testing new code for the first time and identifying new bugs. Just as testing in general is not a guarantee, but a form of insurance, test automation is a method to have even more insurance.

**Some Nuts and Bolts**

When learning to use testing tools it's common to make mistakes. One way to mitigate these mistakes is to create scripts that will provide immediate pay back. That is, create scripts which won't take too much time to create yet will obviously save manual testing effort. These scripts will be immediately useful and it's all right if they're not intended to be part of the permanent test suite. Those creating the scripts will learn more about the tool's functionality and learn to design even better scripts. Not much is lost if these scripts are thrown away since some value has already been gained from them. As experience is gained with the testing tool, a long-term approach to test automation design can start to be developed.

Again, start off small when designing scripts. Identify the functional areas within the application being tested. Design at a higher level how each of these functional areas would be automated, then create a specific automated test design for one of the functional areas. That is, what approach will be used to create scripts using test cases as the basis for automating that function? If there are opportunities to use common scripting techniques with other testing modules, then identify these common approaches as potential standards would be useful in creating maintainable scripts.

Use a similar approach to design and create scripts for some of the other functional areas of the application. As more experience is gained from automation then designing and building scripts to test the integration of these functional areas would be the next step in building a larger and more useful testing suite.

Since the purpose of automating testing is to find bugs, validations should be made as tests are performed. At each validation point there is a possibility of error. Should the script find an error, logic should be built into it so that it can not only report the error it found but also route back to an appropriate point within the automated testing suite so that the automated testing can continue on. This is necessary if automated tests are to be run overnight successfully. This part of test automation is the 'error recovery process'. This is a significant effort since it has to be designed in for every validation point. It's best to design and create reusable error recovery modules that can be called from many validation points in many scripts. Related to this are the reports that get generated from running the tests. Most tools allow you to customize the reports to fit your reporting needs.

It's also important to write documented comments in the test scripts to help those who would maintain the test scripts. Write the scripts with the belief that someone else will be maintaining them.

In the automation test design or documented within the test scripts also identify any manual intervention which is necessary to set up the test environment or test data in order to run the scripts. Perhaps databases need to be loaded or data has to be reset.

**Test Data**

I know of three ways to have the test data populated so that the test environment is setup correctly to run automated tests. If complete control of the test environment is available to testers, then reloading preset databases can be a relatively quick way to load lots of data. One danger in having several preset databases is if a future release requires a reconstruction of data structures and the effort to convert the current data structures to the desired state is a large effort.

Another method of setting up the data is to create tests scripts which run and populate the database with the necessary data to be used in automated tests. This may take a little longer to populate, but there's less dependency on data structures. This method also allows more flexibility should other data change in the database.

Even though I mention 'databases' specifically, the concepts apply to other types of data storage as well.

Other people with test automation experience have used 'randomly' generated data successfully to work with their test scripts. Personally, I have no experience using randomly generated data, but this is another option worth looking into if you're looking for other ways to work with data.

**Potential Risks**

Some common risks to the test automation effort include management and team members support fading after not seeing immediate results, especially when resources are needed to test the current release. Demanding schedules will put pressure on the test team, project management and funding management to do what it takes to get the latest release out. The reality is that the next release usually has the same constraints and you'll wish you had the automated testing in place.

If contractors are used to help build or champion the test automation effort because of their experience, there is the risk that much of the experience and skills will 'walk away' when the contractor leaves. If a contractor is used, ensure there is a plan to back fill this position since the loss of a resource most likely will affect the maintenance effort and new development of test scripts. It's also just as important that there is a comprehensive transfer of knowledge to those who will be creating and maintaining the scripts.

Since the most significant pay back for running automated tests come from future releases, consider how long the application being tested will remain in its current state. If a rewrite of the application is planned in the near future or if the interface is going to be overhauled, then it probably makes sense to only use test automation for immediate pay back. Again, here's where working with application designers and developers can make a difference, especially if internal changes are planned which may not appear to affect the testing team, but in reality can affect a large number of test scripts.

**Summary**

As mentioned earlier, most of the concepts identified here came from experiences and as also noted there are not a lot of facts to back up these ideas. The intent here wasn't to prove any particular technique worked, but, rather just to share methods that appear to be more successful. If nothing else, this information can be used to look at test automation from a little different perspective and assist in planning.

If you have experiences that are different than these that you've found successful, or if you've experienced hardships using some of these recommendations, I'd be grateful to hear from you. Many people, including myself, are interested in finding out what really works in creating higher quality software more quickly.

# Understanding Use Case Modeling

Sinan Si Alhir, salhir@earthlink.net
http://home.earthlink.net/~salhir

## Introduction

Following the "method wars" of the 1970s and 1980s, the Unified Modeling Language (UML) emerged from the unification that occurred in the 1990s within the information systems and technology industry. Unification was lead by Rational Software Corporation and Three Amigos, Grady Booch, James Rumbaugh, and Ivar Jacobson. The UML gained significant industry support from various organizations via the UML Partners Consortium and was submitted to and adopted by the Object Management Group (OMG) as a standard (November 17, 1997).

The UML is an evolutionary general-purpose, broadly applicable, tool-supported, and industry-standardized modeling language for specifying, visualizing, constructing, and documenting the artifacts of a system-intensive process. The language is broadly applicable to different types of systems (software and non-software), domains (business versus software), and methods and processes. The UML enables and promotes (but does not require nor mandate) a use-case-driven, architecture-centric, iterative, and incremental process that is object oriented and component based. The UML enables the capturing, communicating, and leveraging of knowledge: models capture knowledge (semantics), architectural views organize knowledge in accordance with guidelines expressing idioms of usage, and diagrams depict knowledge (syntax) for communication.

System development may be characterized as problem solving, including understanding or conceptualize and representing a problem, solving the problem by manipulating the representation of the problem to derive a representation of the desired solution, and implementing or realizing and constructing the solution. This process is very natural and often occurs subtly and sometimes unconsciously in problem solving.

Models are complete abstractions of systems. Models are used to capture knowledge (semantics) about problems and solutions. Architectural views are abstractions of models. Architectural views are used to organize knowledge in accordance with guidelines expressing idioms of usage. Diagrams are graphical projections of sets of model elements. Diagrams are used to depict knowledge (syntax) about problems and solutions.

Within the fundamental UML notation, concepts are depicted as symbols and relationships among concepts are depicted as paths (lines) connecting symbols.

Use case modeling from the user model view (also known as the use case or scenario view), which encompasses a problem and solution as understood by those individuals whose problem the solution addresses, involves use case diagrams to depict the functionality of a system.

## Use Case Diagrams

To successfully apply use case diagrams, we must first understand the types of elements used in use case diagrams.

### Actors

Actor classes are used to model and represent roles for “users” of a system, including human

users and other systems. Actors are denoted as stick person icons.

They have the following characteristics:

- Actors are external to a system.
- Actors interact with the system. Actors may use the functionality provide by the system, including application functionality and maintenance functionality. Actors may provide functionality to the system. Actors may receive information provided by the system. Actors may provide information to the system.
- Actor classes have actors instances or objects that represent specific actors.

Figure 1 shows a project management system with a project manager actor and a project database actor. The project manager is a user who is responsible for ensuring the success of project and uses the system to manage projects. The project database is a system that is responsible for housing project management data.

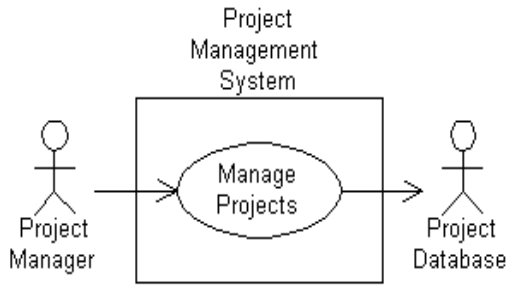



Figure 1

Use Cases

Use case classes are used to model and represent units of functionality or services provided by a system (or parts of a system: subsystems or classes) to users. Use cases are denoted as ellipses or ovals. They may be enclosed by a system boundary or rectangle labeled with the name of the containing system.

They have the following characteristics:

- Use cases are interactions or dialogs between a system and actors, including the messages exchanged and the actions performed by the system. Use cases may include variants of these sequences, including alternative and exception sequences.
- Use cases are initiated by actors and may involve the participation of numerous other actors. Use cases should provide value to at least one of the participating actors.
- Use cases may have extension points that define specific points within an interaction at which other use cases may be inserted.
- Use case classes have use case instances or objects called scenarios that represent specific interactions. Scenarios represent a singe sequence of messages and actions.

Figure 1 shows a project management system which provides the functionality to manage projects in which the project manager and project database participate.

Relationships

Association relationships between actor classes and use case classes are used to indicate that the actor classes participate and communicates with the system containing the use case classes. Association relationships are denoted as solid lines or paths. Arrowheads may be used to indicate who initiates communication in the interaction. If an arrowhead points to a use case, the actor at the other end of the association initiates the interaction with the system. If the arrowhead points to an actor, the system initiates the interaction with the actor at the other end of the association.

Figure 1 shows a project management system that provides functionality to manage projects. A project manager initiates this functionality and the system initiates the communication with the project database in providing this functionality.

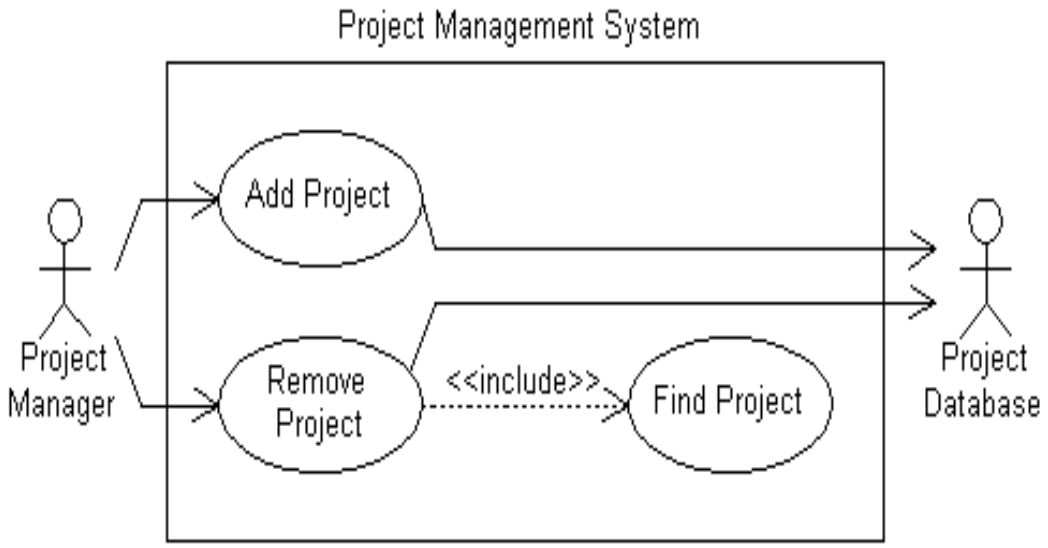



Figure 2

Includes relationships from base use case classes to inclusion use case classes are used to indicate that the base use case classes will contain the inclusion use case classes; that is, the base use case will contain the inclusion use case. A base use case defines the location at which the inclusion use case is included. Includes relationships are denoted as dashed lines or paths with an open arrowhead pointing at the inclusion use case and are labeled with the <<include>> keyword (stereotype). The insertion of the inclusion use case involves the execution of the base use case up to the inclusion point, inserting and executing the inclusion use case, and then continuing with the execution of the base use case.

Figure 2 shows that a project manager may add projects and remove projects using the project management system. When removing projects, the functionality of finding a project is included into removing a project.

Extends relationships from extension use case classes to base use case classes are used to indicate that the base use case classes may be augmented by the extension use case classes; that is, the inclusion use case will augment the base use case if an extension condition is satisfied. A base use case defines the extension point. An extension use case defines the extension condition that must be satisfied in order to insert the extension use case into the base use case.

The insertion of the extension use case involves the execution of the base use case up to the extension point, testing the extension condition and inserting and executing the extension use case if the condition is satisfied, and then continuing with the execution of the base use case. Extends relationships are denoted as dashed lines or paths with an open arrow-head pointing at the extension use case and are labeled with the extension condition in square brackets, the <<extend>> keyword (stereotype), and the extension point name in parentheses. Extension points are identified in a compartment labeled “Extension Points” in the base use case.

Figure 3 shows that a project manager may update projects using the project management system. When updating projects, a project manager may manage tasks if the project manager selects the task option, and a project

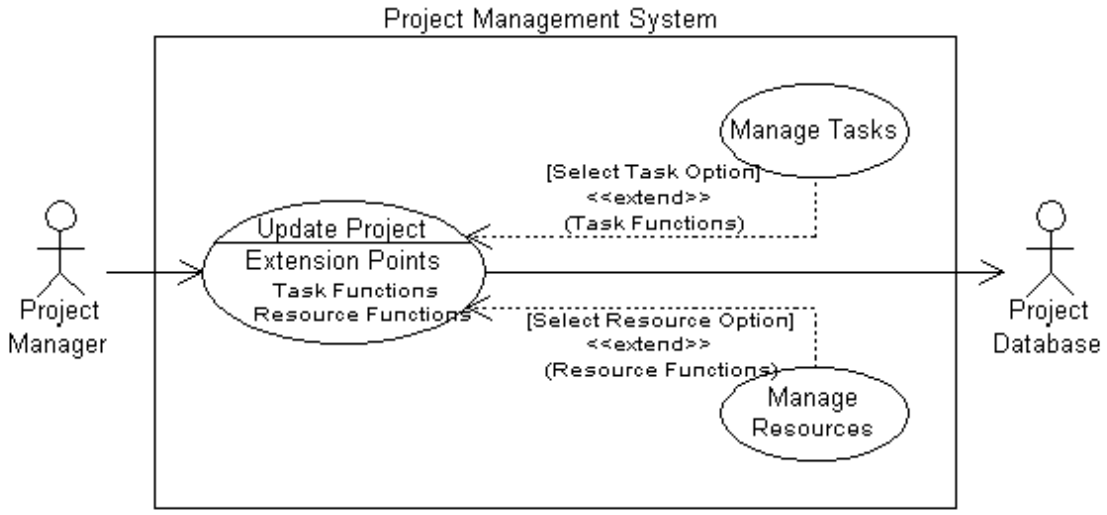



Figure 3

manger may manage resource if the project manager selects the resource option.

Generalization relationships from specialization use case classes to generalized use cases classes are used to indicate the specialization use case classes are consistent with the generalized use case classes and may add additional information. A specialization use case may be used in place of a generalized use case and may use any portions of the interaction of the generalized use case. Generalization relationships are denoted as solid lines or paths with a hollow arrowhead pointing at the generalized use case.

Figure 4 shows that a project manager may publish a project schedule by sending e-mail to project team members using an e-mail system or by generating a web-site on a web-site host. In either case, there will be common functionality used from the generalized use case, for example: inputting project name, extracting the relevant project information from the project database, etc.

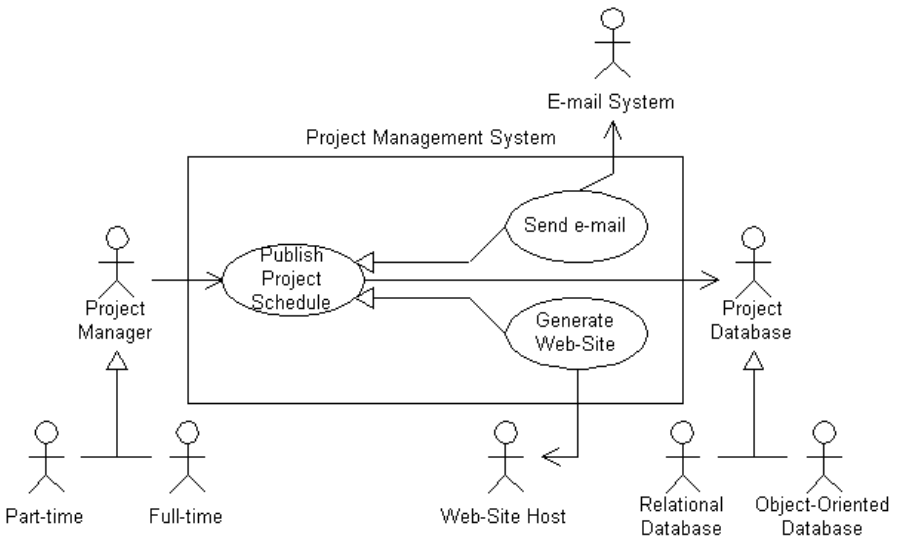



Figure 4

## Lift your SW Development Process to new Heights!

### Enterprise Architecture & DW, DB, Application Modeling

Traditional development methods don't cut it anymore. Quick response to CHANGE is a must. Modeling eliminates all these shortcomings. It quickly shows where you are, where you need to be & how to get there. It is the *blueprint* of IT systems & applications

**Visible Analyst ®** **Visible Advantage™** **Business Templates**

**Benefits**: * Quick Access to '*What Is*'
* Fast adaptation to CHANGE
* Supports collaboration
* Enables integration through XML

### Configuration Management
**Innovative & Affordable**

Too many SW projects are late & show quality problems due to lack of structure in development processes. Large & mid-sized projects need configuration mgmt to stay on the projected course.

Issue/Problem Tracking | Version Control | Release Management

**R A Z O R ®**

**Benefits:** * Increased product quality
* Better control
* Supports collaboration
* Audit trail through life cycle

Visible SYSTEMS CORPORATION

*For more information contact:*
sales@visible.com
or razor_sales@visible.com

www.visible.com/mt

Resellers are welcome!
Excellent business opportunities.


Generalization relationships from specialization actor classes to generalized actor classes are used to indicate the specialization actor classes are consistent with the generalized actor classes and may add additional information.

A specialization actor may be used in place of a generalized actor and receives the characteristics of the generalized actor. Generalization relationships between actors are denoted similarly to generalization relationships between use cases.

Figure 4 shows that there are two types project mangers, full-time and part-time, and that there are two types of project database, relational database management systems (R-DBMS) and object oriented database management systems (OO-DBMS). Any type of project manager may publish a project schedule using any type of project database.

**Use Case Modeling**

To successfully apply use case diagrams in use case modeling, we ought to be aware of various guidelines, and lessons learned from applying this technique.

Actors

When modeling actors, we ought to be aware of the following guidelines:

- Actors should be named using noun phrases.
- Actors should be described, indicating what interests an actor has in interacting with the system. For example, the project manager is responsible for ensuring the success of projects, and the project database is responsible for housing project management data.
- Actors define the scope of a system and identify those elements that reside at the

periphery of the system and those elements on which the system depends. For example, these use case diagrams indicate that the project management system depends on a project database to provide functionality to a project manager, both residing on the periphery of the system.

Furthermore, other guidelines may be applied in addition to those above.

Use Cases

When modeling use cases, we ought to be aware of the following guidelines:

- Use cases should be named using verb-noun phrases.
- Use cases should be described, indicating how they are started and end, any conditions that must be satisfied before the use case starts (pre-conditions), any conditions that must be satisfied when the use case ends (post-conditions), the sequence of exchanged messages and performed actions, the data exchanged, and any non-functional characteristics (reliability, performance, supportability, etc. constraints). This description may be captured using text and other UML diagrams.
- Use cases define the scope of a system and define the functionality provided by the system and those elements on which the system depends in order to provide the functionality. For example, these use case diagrams indicate that the project management system will provide functionality to manage projects to a project manager, and this functionality is implemented using the project database.
- Use cases should facilitate actors in reaching their goals. Use cases are system functionality or responsibilities (requirements) that actors use in order to reach or satisfy their goals. Use cases are not simply actor goals. For example, a project manager is responsible for ensuring the success of projects, and a project database is responsible for housing project management data. The project management system provides functionality to manage projects to a project manager such that the project manger can ensure the success of projects.
- Use cases should facilitate the architecture of a system. Use cases may be organized and partitioned using includes, extends, and generalization relationships to identify, extract, and manage common, optional, and similar functionality. The organization of a set of use cases is not simply the architecture of the system. However, the architecture of a system is based upon the various technology, infrastructure, etc. considerations relevant to satisfying the use cases. For example, the project management system must interface with an e-mail system and a web-site host, thus appropriate subsystem elements must exist within our architecture to facilitate these interfaces.
- Use cases provide flexibility and power throughout the life-cycle process. They provide the freedom to work with a use case as a whole or any subset of a use case via scenarios. The use of includes, extends, and generalization relationships to identify, extract, and manage common, optional, and similar functionality provides further flexibility in working with use cases. Furthermore, use cases may be used to model interactions between actors and systems, subsystems, and classes at various levels of abstraction. This flexibly and power is propagated to every application of use cases. For example, if time, resources, or funding are not sufficient to implement a whole use case, various scenarios may be selected for implementation based upon these factors.
- Use cases may be used as the basis for planning. Time and resource estimates may be associated with use cases. If estimates for a use case cannot be derived, estimates

for each scenario of a use case may be derived and used to potentially estimate the overall time and resource estimates for the use case as a whole. This helps ensure that planning is done with the objective of satisfying the requirements.

- Use cases may be used as the basis for analysis, design, and implementation. The sequence of exchanged messages and performed actions within the description of a use case are analyzed and the system is design and implemented to specifically realize use case interactions. This helps ensure that every element of a system is created and used because it contributes to satisfying the requirements.
- Use cases may be used as the basis for testing. The sequence of exchanged messages and performed actions within the description of a use case may be used as test scripts for validating the functionality of a system. This helps ensure that the system is tested and validated against the requirements.
- Use cases may be used as the basis for documentation since use cases capture how users will use the system.

Furthermore, other guidelines may be applied in addition to those above.

**Conclusion**

As the Unified Modeling Language (UML) is an evolutionary general-purpose, broadly applicable, tool-supported, and industry-standardized modeling language for specifying, visualizing, constructing, and documenting the artifacts of a system-intensive process, by understanding the types of elements used in use case diagrams and being aware of various guidelines (lessons learned) from applying this technique, we have a sound foundation for successfully applying the technique. Furthermore, it is experience, experimentation, and application of the standard and its various techniques that will enable us to realize its benefits.

# GUI Testing Checklist

Barry Dorgan, Bazman @bigfoot.com
http://members.tripod.com/bazman/index.html

**Section 1 - Windows Compliance Standards**

1.1. Application
1.2. For Each Window in the Application
1.3. Text Boxes
1.4. Option (Radio buttons)
1.5. Check Boxes
1.6. Command Buttons
1.7. Drop Down List Boxes
1.8. Combo Boxes
1.9. List Boxes

**Section 2 - Tester's Screen Validation Checklist**

2.1. Aesthetic Conditions
2.2. Validation Conditions
2.3. Navigation Conditions
2.4. Usability Conditions
2.5. Data Integrity Conditions
2.6. Modes (Editable Read-only) Conditions
2.7. General Conditions
2.8. Specific Field Tests
2.8.1. Date Field Checks
2.8.2. Numeric Fields
2.8.3. Alpha Field Checks

**Section 3 - Validation Testing - Standard Actions**

3.1. On every Screen
3.2. Shortcut keys / Hot Keys
3.3. Control Shortcut Keys

## 1. WINDOWS COMPLIANCE TESTING

**For Each Application**

Start the application by double clicking on its icon. The loading message should show the application name, version number, and a bigger pictorial representation of the icon.

No login is necessary.

The main window of the application should have the same caption as the caption of the icon in Program Manager.

Closing the application should result in an "Are you Sure" message box.

Attempt to start application twice. This should not be allowed - you should be returned to main Window.

Try to start the application twice as it is loading.

On each window, if the application is busy, then the hour glass should be displayed. If there is no hour glass (e.g. alpha access enquiries) then some enquiry in progress message should be displayed.

All screens should have a Help button, F1 should work doing the same.

**For Each Window in the Application**

If the window has a minimize button, click it.

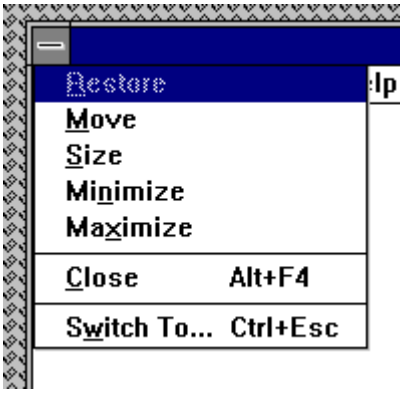
The window should return to an icon on the bottom of the screen.

This icon should correspond to the original icon under Program Manager.

Double click the icon to return the window to its original size.

The window caption for every application should have the name of the application and the window name - especially the error messages. These should be checked for spelling, English and clarity, especially on the top of the screen. Check if the title of the window does make sense.

If the screen has a control menu, then use all ungreyed options. (see below)




Check all text on window for spelling/tense and grammar

Use TAB to move focus around the window.
Use SHIFT+TAB to move focus backwards.

Tab order should be left to right, and up to down within a group box on the screen. All controls should get focus - indicated by dotted box, or cursor. Tabbing to an entry field with text in it should highlight the entire text in the field.

The text in the micro help line should change. Check for spelling, clarity and non-updateable etc.

If a field is disabled (greyed) then it should not get focus. It should not be possible to select it with either the mouse or by using TAB. Try this for every greyed control.

Never updateable fields should be displayed with black text on a grey background with a black label.

All text should be left-justified, followed by a colon tight to it.

In a field that may or may not be updateable, the label text and contents changes from black to grey depending on the current status.

List boxes are always white background with black text whether they are disabled or not. All others are grey.

In general, do not use goto screens, use gosub, i.e. if a button causes another screen to be displayed, the screen should not hide the first screen.

When returning, return to the first screen cleanly i.e. no other screens/applications should appear.

In general, double-clicking is not essential. In general, everything can be done using both the mouse and the keyboard.

All tab buttons should have a distinct letter.

**Text Boxes**

**Program Filename**:

Move the mouse cursor over all enterable text boxes. The cursor should change from arrow to insert bar. If it doesn't then the text in the box should be grey or non-updateable.

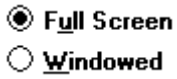
Enter text into the box

Try to overflow the text by typing to many characters - should be stopped Check the field width with capitals W.

Enter invalid characters - Letters in amount fields, try strange characters like + , - * etc. in **All** fields.

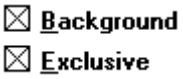
SHIFT and arrow should select characters. Selection should also be possible with mouse. Double click should select all text in box.

**Option (Radio Buttons)**




The left and right arrows should move 'ON' selection. So should up and down. Select with the mouse by clicking.

**Check Boxes**




Clicking with the mouse on the box, or on the text should SET/UNSET the box. SPACE should do the same.

**Command Buttons**

If the command button leads to another screen, and if the user can enter or change details on the other screen, then the text on the button should be followed by three dots.

All buttons except for OK and Cancel should have a letter access to them. This is indicated by a letter underlined in the button text. The button should be activated by pressing ALT+letter. Make sure that there is no duplication.

Click each button once with the mouse - This should activate

Tab to each button - Press SPACE - This should activate

Tab to each button - Press RETURN - This should activate

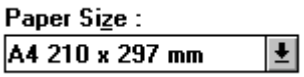
The above are **VERY IMPORTANT**, and should be done for **EVERY** command button.

Tab to another type of control (not a command button). One button on the screen should be default (indicated by a thick black border). Pressing the return key: in ANY CASE no command button control should activate it.

If there is a Cancel button on the screen, then pressing <Esc> should activate it.

If pressing the command button results in uncorrectable data e.g. closing an action step, there should be a message phrased positively with Yes/No answers where Yes results in the completion of the action.

**Drop Down List Boxes**




Pressing the arrow should give list of options. This list may be scrollable. You should not be able to type text in the box.

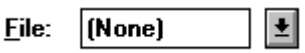
Pressing a letter should bring you to the first item in the list with that start with that letter. Pressing ‘Ctrl - F4’ should open/drop down the list box.

Spacing should be compatible with the existing windows spacing (word etc.). Items should be in alphabetical order with the exception of blank/none which is at the top or the bottom of the list box.

Drop down with the item selected should be display the list with the selected item on the top.

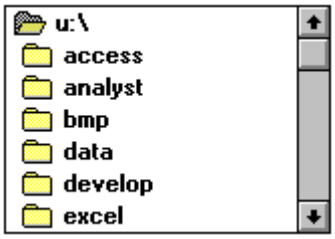
Make sure that only one space appears, you shouldn't have a blank line at the bottom.

**Combo Boxes**




Should allow text to be entered. Clicking the arrow should allow user to choose from list

**List Boxes**




Should allow a single selection to be chosen, by clicking with the mouse, or using the up and down arrow keys.

Pressing a letter should take you to the first item in the list starting with that letter.

If there is a 'View' or 'Open' button beside the list box then double clicking on a line in the list box, should act in the same way as selecting and item in the list box, then clicking the command button.

Force the scroll bar to appear, make sure all the data can be seen in the box.

**2. Screen Validation Checklist**

**Aesthetic Conditions:**

1. Is the general screen background the correct colour?
2. Are the field prompts the correct color?
3. Are the field backgrounds the correct color?
4. In read-only mode, are the field prompts the correct color?
5. In read-only mode, are the field backgrounds the correct color?
6. Are all the screen prompts specified in the correct screen font?
7. Is the text in all fields specified in the correct screen font?
8. Are all the field prompts aligned perfectly on the screen?
9. Are all the field edit boxes aligned perfectly on the screen?
10. Are all groupboxes aligned correctly on the screen?
11. Should the screen be resizable?
12. Should the screen be minimisable?
13. Are all the field prompts spelt correctly?
14. Are all character or alpha-numeric fields left justified? This is the default unless otherwise specified.
15. Are all numeric fields right justified? This is the default unless otherwise specified.
16. Is all the microhelp text spelt correctly on this screen?
17. Is all the error message text spelt correctly on this screen?
18. Is all user input captured in UPPER case or lower case consistently?
19. Where the database requires a value (other than null) then this should be defaulted into fields. The user must either enter an alternative valid value or leave the default value intact.
20. Assure that all windows have a consistent look and feel.
21. Assure that all dialog boxes have a consistent look and feel.

**Validation Conditions:**

1. Does a failure of validation on every field cause a sensible user error message?
2. Is the user required to fix entries which have failed validation tests?
3. Have any fields got multiple validation rules and if so are all rules being applied?
4. If the user enters an invalid value and clicks on the OK button (i.e. does not TAB off the field) is the invalid entry identified and highlighted correctly with an error message?
5. Is validation consistently applied at screen level unless specifically required at field level?

6. For all numeric fields check whether negative numbers can and should be able to be entered.
7. For all numeric fields check the minimum and maximum values and also some mid-range values allowable?
8. For all character/alphanumeric fields check the field to ensure that there is a character limit specified and that this limit is exactly correct for the specified database size?
9. Do all mandatory fields require user input?
10. If any of the database columns don't allow null values then the corresponding screen fields must be mandatory. (If any field which initially was mandatory has become optional then check whether null values are allowed in this field.)

**Navigation Conditions:**

1. Can the screen be accessed correctly from the menu?
2. Can the screen be accessed correctly from the toolbar?
3. Can the screen be accessed correctly by double clicking on a list control on the previous screen?
4. Can all screens accessible via buttons on this screen be accessed correctly?
5. Can all screens accessible by double clicking on a list control be accessed correctly?
6. Is the screen modal, is the user prevented from accessing other functions when this screen is active and is this correct?
7. Can a number of instances of this screen be opened at the same time and is this correct?

**Usability Conditions:**

1. Are all the dropdowns on this screen sorted correctly? Alphabetic sorting is the default unless otherwise specified.
2. Is all date entry required in the correct format?
3. Have all pushbuttons on the screen been given appropriate shortcut keys?
4. Do the shortcut keys work correctly?
5. Have the menu options which apply to your screen got fast keys associated and should they have?
6. Does the TAB order specified on the screen go in sequence from top left to bottom right? This is the default unless otherwise specified.
7. Are all read-only fields avoided in the TAB sequence?
8. Are all disabled fields avoided in the TAB sequence?
9. Can the cursor be placed in the microhelp text box by clicking on the text box with the mouse?
10. Can the cursor be placed in read-only fields by clicking in the field with the mouse?
11. Is the cursor positioned in the first input field or control when the screen is opened?
12. Is there a default button specified on the screen?
13. Does the default button work correctly?
14. When an error message occurs does the focus return to the field in error when the user cancels it?
15. When the user Alt+Tab's to another application does this have any impact on the screen upon return to the application?
16. Do all the fields edit boxes indicate the number of characters they will hold by their length? e.g. a 30 character field should be a lot longer

**Data Integrity Conditions:**

1. Is the data saved when the window is closed by double clicking on the close box?
2. Check the maximum field lengths to ensure that there are no truncated characters?
3. Where the database requires a value (other than null) then this should be defaulted into

fields. The user must either enter an alternative valid value or leave the default value intact.

4. Check maximum and minimum field values for numeric fields?
5. If numeric fields accept negative values can these be stored correctly on the database and does it make sense for the field to accept negative numbers?
6. If a set of radio buttons represent a fixed set of values such as A, B and C then what happens if a blank value is retrieved from the database? (In some situations rows can be created on the database by other functions which are not screen based and thus the required initial values can be incorrect.)
7. If a particular set of data is saved to the database check that each value gets saved fully to the database. Beware of truncation (of strings) and rounding of numeric values.

**Modes (Editable Read-only) Conditions:**

1. Are the screen and field colors adjusted correctly for read-only mode?
2. Should a read-only mode be provided for this screen?
3. Are all fields and controls disabled in read-only mode?
4. Can the screen be accessed from the previous screen/menu/toolbar in read-only mode?
5. Can all screens available from this screen be accessed in read-only mode?
6. Check that no validation is performed in read-only mode.

**General Conditions:**

1. Assure the existence of the "Help" menu.
2. Assure that the proper commands and options are in each menu.
3. Assure that all buttons on all tool bars have a corresponding key commands.
4. Assure that each menu command has an alternative (hot-key) key sequence which will invoke it where appropriate.
5. In drop down list boxes, ensure that the names are not abbreviations / cut short
6. In drop down list boxes, assure that the list and each entry in the list can be accessed via appropriate key / hot key combinations.
7. Ensure that duplicate hot keys do not exist on each screen
8. Ensure the proper usage of the escape key (which is to undo any changes that have been made) and generates a caution message “Changes will be lost - Continue yes/no”
9. Assure that the cancel button functions the same as the escape key.
10. Assure that the Cancel button operates as a Close button when changes have been made that cannot be undone.
11. Assure that only command buttons which are used by a particular window, or in a particular dialog box, are present. - I.e. make sure they don’t work on the screen behind the current screen.
12. When a command button is used sometimes and not at other times, assure that it is grayed out when it should not be used.
13. Assure that OK and Cancel buttons are grouped separately from other command buttons.
14. Assure that command button names are not abbreviations.
15. Assure that all field labels/names are not technical labels, but rather are names meaningful to system users.
16. Assure that command buttons are all of similar size and shape, and same font and font size.

**17.** Assure that each command button can be accessed via a hot key combination.

**18.** Assure that command buttons in the same window/dialog box do not have duplicate hot keys.

**19.** Assure that each window/dialog box has a clearly marked default value (command button, or other object) which is invoked when the Enter key is pressed - and NOT the Cancel or Close button

**20.** Assure that focus is set to an object/button which makes sense according to the function of the window/dialog box.

**21.** Assure that all option buttons (and radio buttons) names are not abbreviations.

**22.** Assure that option button names are not technical labels, but rather are names meaningful to system users.

**23.** If hot keys are used to access option buttons, assure that duplicate hot keys do not exist in the same window/dialog box.

**24.** Assure that option box names are not abbreviations.

**25.** Assure that option boxes, option buttons, and command buttons are logically grouped together in clearly demarcated areas “Group Box”.

**26.** Assure that the Tab key sequence which traverses the screens does so in a logical way.

**27.** Assure consistency of mouse actions across windows.

**28.** Assure that the color red is not used to highlight active objects (many individuals are red-green color blind).

**29.** Assure that the user will have control of the desktop with respect to general color and highlighting (the application should not dictate the desktop background characteristics).

**30.** Assure that the screen/window does not have a cluttered appearance.

**31.** Ctrl + F6 opens next tab within tabbed window.

**32.** Shift + Ctrl + F6 opens previous tab within tabbed window.

**33.** Tabbing will open next tab within tabbed window if on last field of current tab.

**34.** Tabbing will go onto the 'Continue' button if on last field of last tab within tabbed window.

**35.** Tabbing will go onto the next editable field in the window.

**36.** Banner style and size and display exact same as existing windows.

**37.** If 8 or less options in a list box, display all options on open of list box - should be no need to scroll.

**38.** Errors on continue will cause user to be returned to the tab and the focus should be on the field causing the error. (I.e. the tab is opened, highlighting the field with the error on it).

**39.** Pressing continue while on the first tab of a tabbed window (assuming all fields filled correctly) will not open all the tabs.

**40.** On open of tab focus will be on first editable field.

**41.** All fonts to be the same

**42.** Alt+F4 will close the tabbed window and return you to main screen or previous screen (as appropriate), generating "changes will be lost" message if necessary.

**43.** Microhelp text for every enabled field and button

**44.** Ensure all fields are disabled in read-only mode.

**45.** Progress messages on load of tabbed screens.

**46.** Return operates continue.

**47.** If retrieve on load of tabbed window fails window should not open.

**Specific Field Tests**

**Date Field Checks**

Assure that leap years are validated correctly and do not cause errors/miscalculations.

Assure that month code 00 and 13 are validated correctly and do not cause errors/miscalculations.

Assure that 00 and 13 are reported as errors.

Assure that day values 00 and 32 are validated correctly and do not cause errors/miscalculations.

Assure that Feb. 28, 29, 30 are validated correctly and do not cause errors/ miscalculations.

Assure that Feb. 30 is reported as an error.

Assure that century change is validated correctly and does not cause errors/ miscalculations.

Assure that out of cycle dates are validated correctly and do not cause errors/miscalculations.

**Numeric Fields**

Assure that lowest and highest values are handled correctly.

Assure that invalid values are logged and reported.

Assure that valid values are handles by the correct procedure.

Assure that numeric fields with a blank in position 1 are processed or reported as an error.

Assure that fields with a blank in the last position are processed or reported as an error an error.

Assure that both + and - values are correctly processed.

Assure that division by zero does not occur.

Include value zero in all calculations.

Include at least one in-range value.

Include maximum and minimum range values.

Include out of range values above the maximum and below the minimum.

Assure that upper and lower values in ranges are handled correctly.

**Alpha Field Checks**

Use blank and non-blank data.

Include lowest and highest values.

Include invalid characters and symbols.

Include valid characters.

Include data items with first position blank.

Include data items with last position blank.

**3. VALIDATION TESTING - STANDARD ACTIONS**

**On every Screen**

Add
View
Change
Delete
Continue

Add
View
Change
Delete
Cancel

Fill each field - Valid data

Fill each field - Invalid data

Different Check Box combinations

Scroll Lists

Help

Fill Lists and Scroll

Tab

Tab Order

Shift Tab

Shortcut keys - Alt + F

**CONTROL SHORT KEYS**

Recommended CTRL+Letter Shortcuts

| ***Key*** | ***Function*** |
|---|---|
| CTRL+Z | Undo |
| CTRL+X | Cut |
| CTRL+C | Copy |
| CTRL+V | Paste |

Suggested CTRL+Letter Shortcuts

| ***Key*** | ***Function*** |
|---|---|
| CTRL+N | New |
| CTRL+O | Open |
| CTRL+P | Print |
| CTRL+S | Save |
| CTRL+B | Bold* |
| CTRL+I | Italic* |
| CTRL+U | Underline* |

* These shortcuts are suggested for text formatting applications, in the context for which they make sense. Applications may use other modifiers for these operations.

**Shortcut keys & Hot Keys**

| Key | No Modifier | SHIFT | CRTL | ALT |
|---|---|---|---|---|
| F1 | Help | Enter Help Mode | N\A | N\A |
| F2 | N\A | N\A | N\A | N\A |
| F3 | N\A | N\A | N\A | N\A |
| F4 | N\A | N\A | Close Document Window | Close Application Window |
| F5 | N\A | N\A | N\A | N\A |
| F6* | Move clockwise to next pane of active window | Move counterclockwise to next pane of active window | Move to next document window; top window moves to bottom of stack (adding SHIFT reverses action: previous window moves to top. | Move to application's next open non-document window (Adding SHIFT reverses order of movement) |
| F7 | N\A | N\A | N\A | N\A |
| F8 | Toggle extend mode, if supported | Toggle Add mode , if supported | N\A | N\A |
| F9 | N\A | N\A | N\A | N\A |
| F10 | Toggle menu Bar activation | N\A | N\A | N\A |
| F11,F12 | N\A | N\A | N\A | N\A |

## Companies

**Cap Gemini & Ernst & Young**

Cap Gemini, the French-based software house, has reached an agreement with Ernst & Young to acquire its consulting and IT services arm. The completion of the transaction is now subject to the vote of Ernst & Young partners on a country by country basis.

The new company will have a strong position both in Europe and in the USA, eliminating a Cap Gemini weakness. It will however be interesting to see how the cultural differences between a traditional software house and a partnership-based company will influence the fusion of these two organizations.

**Linux my Love**

Corel and Inprise/Borland announced in February that they have entered into a merger agreement, with the goal to become a leader in the Linux market. The combined revenues of the two companies in 1999 were of $418 million.

These two companies have in common a long history in the shade of Microsoft. The office productivity suite of Corel has never been really competitive against Microsoft or Lotus/IBM products. The recent life of Inprise has been agitated also. Its new orientation (and name) toward middleware with the Visigenic's Visibroker acquisition has not been a big success. Its main assets are its competencies in software development environments with products like Delphi or JBuilder. Part of the technology is also licensed to bigger players like Oracle.

The bases of this marriage seem as unstable as the two companies. If the Linux name has a big marketing potential in financial market and a good future as a server or home computing operating system, I could not see a spectacular growth as a professional desktop system, that is the main market for office productivity suite and software development environments. The least we can hope is that the merger process will not damage too much the quality and evolution of Inprise software development products.

**Big Fish gets Bigger**

In mid-February, Computer Associates International (CA) announced an agreement to acquire Sterling Software.

Sterling Software was often compared to CA in its strategy of acquiring a lot of companies to cash on licenses. In the recent years, Sterling has acquired Knowledgeware (ADW), TI Software (IEF/Composer), Synon and Cayenne Software. All these products were remixed in a new line using the "Cool:" prefix. With its acquisition by CA, the probability of having a strong company working on the evolution of software engineering tools seems to vanish. If


# Everlasting Advertisement Available

5000 registered readers receive Methods & Tools via e-mail...
More than 5000 visitors read the online HTML version...
More than 500 times downloads each month for the past issues in PDF version...

**Advertise in Methods & Tools and you will harvest results forever!**
(and this space costs only $60!)

**Interested? Contact us: franco@martinig.ch**

we examine the list of companies and products acquired by CA (IDMS; Ideal/Datacom, Clipper, Ingres, Platinum,...), we can see that none has left a mark as an technology innovator after its absorption. The problem is that customers working in traditional and mainframe environments have now fewer chances to find another alternative except working with CA...

## $9%£ Numbers

### Wanted: Happy Workers

"The average job tenure in IT has shrunk to about 13 months, down from about 18 months two years ago"

Source: "The Joy of Quitting", Fortune, February 7, 2000

Think about your job today. How much time would you say that you spend working or thinking about work?

Too much time ..... 50%
Just the right amount of time ..... 47%
Not enough time ..... 3%

Think again about your job today. How well would you say that you are paid?

More than enough ..... 7%
Just enough ..... 33%
Not enough ..... 60%

Overall, do you think that your job has?

Exceeded your expectations ..... 16%
Met your expectations ..... 52%
Fallen short of your expectations ..... 28%
Been completely disappointing ..... 4%

Source: "Great Expectations?", Fast Company, November 1999

From the same Fortune's article: "So what makes people start looking in the first place? The short answer is bad management 'People don't quit because of money' says Bev Kaye, a retention expert and co-author of *Love 'Em or Lose 'Em* 'People leave bad bosses."

So if you think that people are the key success factors in software project, try to keep them happy.

## In Others' Words

### The Blame Game

"Blaming is a self defense mechanism. People react personally, in a group, or as a corporation, when they are under pressure, make mistakes, are put into uncomfortable situations, or are attacked. They react in two basic ways (with variations); they fight back (attack) or they withdraw. People blame others to deflect a problem, incident, or negative attention away from themselves. We all have been involved in the blame game – on both the receiving and giving ends.

Blaming can have varying degrees of impact, and its intensity and the harm it may cause range widely. Someone may point the finger at us – or we may point it at someone else – for a problem we may or may not have caused. We may become the targets of intimidating grilling tactics intended to have us own up to causing the problem. An individual or organization may try to keep employees constantly under its thumb by telling them how unworthy they are; this is common in environments that do not tolerate mistakes. In perhaps the worst case, a supposedly trusted friend, maybe under pressure, diverts blame toward you and hurts you personally or professionally.

Why do we blame? We have been conditioned to do so. Many of us learn at a young age that owning up to mistakes can have dire consequences – physical punishment, loss of privileges, belittlement, and so forth. We learn defense mechanisms like the blame to cope, and learn from our parents' sometimes less-than-stellar examples that blaming is okay. This

conditioning is reinforced throughout our lifetimes in all facets of society. In the business world, we primarily play the game because it is easy (with few if any apparent repercussions), we are used to playing it, and we feel we need to protect ourselves and escape responsibility for our actions.

[...] The blame game pervades our business culture at the work group, department and corporate levels. Its extreme form, scapgoating, instigates reorganizations, layoffs, mergers and financial manipulations to mask real problems and deflect the true cause of failure should these measures not succeed. A number of organizations never successfully complete these undertakings despite public assertions and appearances to the contrary.

Why many large-scale changes do not succeed is difficult to say. Is it because corporations didn't have the right business motivation, didn't plan well, or didn't have the know-how, patience or perseverance? Because we don't want to analyze and reveal our failures to others, and because we use the blame game well to deflect blame away from the root cause of failure, we may never know why these initiatives fail. If we could examine these failures we might well see the influence of the blame game in them.

[...] We must remember that people are the foundation of our organizations. Processes should support people and make them more effective. Technology exists to integrate with our processes, automate them and make our people more productive, not replace them. As organizational change agents, we must remember this and also demonstrate it to our clients – in our working relationships, our willingness to evaluate and use the best of their practices, and our willingness to stand up to our convictions. It must manifest in our efforts to define and implement organizational process and technology implementation strategies that consider the needs of the people affected.

Some specific organizational remedies for the blame game include the following:

1. Stop deflecting problems – don't run away from finding their root causes.
2. Take responsibility and admit mistakes as a learning experience.
3. Change the blaming culture; "civilize out" the natural instinct to blame.
4. Establish objective performance standards based on a common corporate vision and values.
5. Build effective, adult-to-adult relationships based on trust and respect.
6. Develop teamwork – we all sink or swim together. Do not tolerate blaming on others.
7. Don't overlook personal agendas. Being able to satisfy them helps people overcome individual barriers.

The blame game starts and can be stopped at a personal level – as individuals we can make a big difference in successfully implementing positive change at any organizational level. Lessons I've learned include maintaining a service instead of a self orientation, believing there is no limit to the good we can do if we don't care who gets the credit, an maintaining a sensitivity toward others. To paraphrase Ralph Waldo Emerson, we succeed when we offer the best in ourselves and strive to leave the world a better place. When we stop blaming people and start truly valuing them, and when we are not afraid to take responsibility for our deeds, we sow the first seeds of that success.

Source: Manfred Hein, "The Blame Game", IEEE Software, November/December 1998.

**The Rookie Manager**

"To get better at anything, you have to try, fail, reflect, regroup, and try again, until you succeed. Your first successes are likely to be sporadic and hard to repeat. Over time, and with enough practice and experience, you will improve your performance until you can perform skillfully. It doesn't matter if the skill is creating a good object model in UML, writing exception-safe C++ classes, or negotiating a delivery date for your next release.

[...] Pressman identifies some of the fears a rookie project manager has when assuming a new position. One of the biggest was the "Dilbert Factor", which I interpret as the fear of appearing technically incompetent in front of your technical peers as you exchange nitty gritty technical skills for management skills. This fear is strongest in those who aren't certain they want to be managers. To be blunt, if you really want to be a good manager, you'll accept that increasing your management and leadership skills will mean losing some technical skills. And yes, at times you'll end up looking just as clueless as Dilbert's pointy-haired boss. It is inevitable.

I've found that a different kind of fear is often mistaken for the "Dilbert Factor": the fear of the unknown. Suppose a rookie project manager is genuinely interested in becoming a manager, but harbors a legitimate fear that she may simply not like management. In many companies this fear is compounded by the fact that once your are 'promoted' from the technical ranks into a project management position, there is little hope of going back to a technical position without being labeled a failure.

We take a different approach at my company. Instead of a one-time promotion from development into project management, we try to grow the project and general management skills of interested developers through a series of assignments.

[...] Developers can assume the role of project lead without formally changing job title or role. This allows them to 'try on' management/leadership roles without the usual assumption that once you become a manager you're stuck.

[...] At Aurigin, a developer can work on one project, be a lead on the next, and go back to working on a third. This provides the necessary time to reflect on 'management' experiences. More importantly, we've found that when a rookie manager has once again become a 'normal' developer she is a more effective team member on the next projects. Simply put, to be an effective leader, you must also be an effective follower!"

Source: Luke Hohmann, "Coaching the Rookie Manager", IEEE Software, January/February 1999

**Coming next in Methods & Tools**

- Data Warehouse Design
- How to Sponsor a Successful Project
- Testing Client/Server Applications

## Classified Advertisement

Advertising for a new Web development tool? Looking to recruit software developers? Promoting a conference or a book? Organizing software development related training? This space is waiting for you at the price of US $ 20 each line. Reach more than 5'000 web-savvy software developers and project managers worldwide with a classified advertisement in Methods & Tools. Without counting those that download the issue without being registered! To advertise in the classified section, to place a page ad or to become the distribution sponsor of the next issue, simply send an e-mail to franco@martinig.ch

**METHODS & TOOLS** 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; e-mail franco@martinig.ch
The content of this publication cannot be reproduced without prior written consent of the publisher
**ISSN 1023-4918 Copyright ã 2000, Martinig & Associates**