

# Writing Autonomic Software

By Steve Mastrianni, IBM Research  
January 2003

## Abstract

Computer users are just interested in getting their work done. Whether surfing the Web, editing photographs, preparing architectural drawings, or monitoring the weather, computer users expect the computer system and software to do what is expected. They don't want to see cryptic error messages, warnings about a new critical update, or program execution errors that cause all their work in progress to be lost. While those in the software engineering profession are used to such behavior and tend to tolerate it, most users find it annoying and even intimidating. As engineers, haven't done a good job in isolating the user from things they shouldn't have to understand, and moreover, things that we can fix automatically.

## Introduction

When customers buy telephone sets and plug them into the wall jack, they pick up the receiver and expect to hear a dial tone if the line is active. They don't need to know anything about the modulation techniques used, nor do they have to adjust any voltages or enter special codes, it just works. The personal computer is much more complex than an analog telephone, but the users are often one and the same. In spite of this, we continue to write software as if users were skilled engineers who understand what a General Protection Fault is, or what an illegal memory reference means. We fill applications with meaningless dialog and message boxes. We make it a point to trap every exception, and then present the user with this complex and meaningless information as if they should be able to understand what happened, and then leave them to infer what to do next. These messages and dialogs are interesting and informative to us as developers, but mean very little to the average user. Even a technically innocuous message can be

intimidating, so the application or system should do its best to fix the problem without the user's knowledge or intervention.

In this paper, we summarize the ever too often frustrating experiences encountered in using personal computers. We then present a taxonomy of causes (faults) of the frustration and finally discuss some remedies that a self-healing, autonomic personal computing system could use to combat these faults.

## Autonomic Applications

An autonomic application is an application that relieves users from the drudgery of dealing with such things as update notifications and error messages, and just lets them get their job done. For example, if a critical update is available, that update should be downloaded and installed automatically. If it is not a critical fix, it should not be marked as such. Of course, there are a few caveats.

First, developers and other similar users should have the ability to hold off even critical updates because they need to control their development environment. Second, there must be a rollback capability in the event that the critical update causes the system to work incorrectly. These items should not be buried in something called a Control Panel, and then Add/Remove Programs, but should be easily accessible with a one-button Help key. The user should be presented with a button next to a caption that says something like "My computer does not appear to be working correctly since I installed the critical update on 04/10/2002 at 13:40. I'd like to restore the system to the way it was before I installed the critical update on 04/10/2002 at 13:40." If the critical update was a device driver, the user should be able to click a button next to the caption "My computer does not seem to be working correctly since I installed the sound

card drivers on 03/12/2002 at 17:50. I'd like to restore the sound card driver to the way it was before the upgrade I performed on 03/12/2002 at 17:50."

During the removal of software, the user should never be presented with dialog boxes asking if it is "OK" to remove certain DLLs that might not be in use or that might be shared with other applications. Most users are afraid to say yes or no, and may abandon the process, leaving the software in a corrupt state. An autonomic system should figure out which option makes the most sense and do it automatically with no operator intervention.

In the case of an error during program execution, the user should never be presented with something like "the application performed an illegal access to location 0x340003ab". A better description would be "The program you were running, "[Program Name]", encountered a programming error that caused the program to stop working. You did nothing wrong, nor did you lose any work, as work that you were doing was saved automatically. You can access this saved work in the Save folder. Check the software manufacturer's Web site to see if there any updates for your software or answers to the problem you're having."

One of the side effects of improperly-written C++ programs is the memory leak. In this case, the program requests a block of memory but never releases it. Over time, the offending program consumes the available memory, causing the system to slow to a crawl or become unresponsive. Instead of fixing this problem, the user is presented with a message that says something like "system memory is low, you must shutdown running programs." Here the user is supposed to know what "system memory" is, and what to do about the problem. No other useful information is usually given. Yet the system knows what program is causing the problem, and even how fast that program is using up resources. The application can be instrumented to determine what resources have been requested and when they are returned to the resource pool. Over time, the system can determine which application is likely causing the problem, and can

automatically stop that program from being loaded the next time the system is restarted. The system can then report to the user that the program has a problem and has been using up valuable resources and not releasing them correctly. The user can then be directed to check the program manufacturer's Web site for updates to that program or asked not use the program until the problem has been identified and corrected.

Another area that users find difficult is configuring their system with the correct drivers, protocols, and networking software to get connected to a network or the Internet. We expect users to understand such things as IP addresses, TCP/IP, NetBIOS, DNS, and WINS. Most users don't know what these acronyms mean, let alone understand how to configure them. An autonomic application or connection wizard should not make the user traverse a set of dialogs, or be forced to enter dozens of meaningless parameters just to surf the Web. Configuration should be automatic, the user should not have to enter any parameters, nor should they be forced to understand the details of network communications.

The use of DHCP went a long way to remedy some of these problems, but this is still not an autonomic solution. In order to use DHCP, the system first has to have a valid IP connection. In the case of a wired LAN connection, this means that the network adapter card must be installed, the correct drivers must be loaded for the card, the network card must be working properly with no resource conflicts, the TCP/IP protocol must be loaded and associated with the adapter, and the cable from the network card must be plugged into an operational LAN. In a wireless situation, almost all of these conditions must be met, but instead of having to be plugged into a LAN, the computer must be able to dial a server and establish an authenticated connection. If the network supports DHCP, the system should be able to use the DHCP protocol to locate most of the settings.

While this scenario works well for a networked home or small business, it does not work well in an enterprise environment.

In the corporate environment, users are often required to access the Internet or email by using a proxy server. The purpose of this proxy server is to route IP traffic around the firewall while maintaining security. Users must know the name of the local proxy server and they must configure their system to allow access outside the firewall. There is no industry-wide standard for naming proxy servers, and thus no mechanism similar to DHCP to allow the automatic configuration of the user's system.

Because of the complexity and number of parameters required to configure a system for network access, autonomic connectivity is an important part of autonomic computing. When a user plugs the network cable into the wall jack, the system should work just like the earlier example of the telephone. The user should not have to enter any parameters, install any drivers or protocols, or enter the names of local servers. The user just needs to get work done, and should not be distracted with dialogs, messages, or connections that are never made. This is even more important to the 'road warrior', who is often seen trying to retrieve his or her email between connecting flights. These users don't have time to configure their systems before the next important meeting.

Once connected, we can enhance the autonomic computing experience by providing server-based content, such as help, tips, regional alerts, regional configuration changes, access to local resources, printers, storage, and services. Using the server, the client system can determine its location based on information in, say, an LDAP server, measuring network hops or network traffic speed, or if multiple access points are available, using some type of triangulation. Once the location is known, the server can help the user get connected. The server could install a small agent on the client, and that agent could then query the server for connection information. Using the connection information, the agent can configure the user's system, making sure all the correct software is installed. If software needs to be installed, the server can download it to the client and have it installed. Web services can and should be used to provide

connectivity information to the client from specialized web sites.

In an enterprise environment, the system can contact other systems and peers. With the proper authentication, the peer system could provide the client system with the configuration parameters and software necessary to perform various tasks.

The most important aspect of this procedure is that it should be done completely automatically, and should require no user intervention. One of the factors limiting this type of dynamic configuration has been the inability to change network parameters without the requirement to reboot the system. Configuration of this type requires the ability to build a network stack dynamically, allowing parts of the software stack to come and go without the need to reboot. Windows XP shows great promise in this area by allowing many of the network parameters to be changed without rebooting. It provides the ability to roll back a driver, although this feature still requires a great deal of user intervention. Windows XP introduced side-by-side DLLs, allowing DLLs to be used for a particular application without replacing an existing DLL with the same name that is also used by other applications.

### **Building Autonomic Systems and Software**

Autonomic software falls into the following categories, each of which can be implemented at some level using current hardware and software technologies. The current release of Windows XP addresses some of these issues, but goes only part-way in providing true autonomic behavior. The categories are:

- Break/Fix
- Proactive monitoring
- Device instrumentation
- Application instrumentation
- On-Demand Wizards
- Collaboration

## Break/Fix

Break/Fix is the category that handles errors in real time, as they occur. Break/Fix implementation needs to be done locally, on the system that experienced the fault. An example of Break/Fix would be what happens when a PCMCIA card or USB device is inserted. Windows detects the insertion, identifies the device, and checks to see if a driver is already loaded for the device. If the driver exists, the driver's entry points are called to configure the resources for the new device. If the driver is not resident, the system prompts the user to insert a driver disk or allows the user to search the web for a suitable driver. If the driver installation files and binaries are located in a specific folder, that folder can be automatically searched. If found, the driver is installed, the configuration entry points called, and the device becomes available. The user should not be involved in this process unless a critical error is encountered. All error logging should be to a database of actions and results that is linked to the device and device driver. Telling the user that the system installed the device and that the device is now available may be somewhat self-gratuitous on the part of the system, and the user may not want to hear about it anyway.

Break/Fix requires instrumentation to detect, identify, and fix problems. Microsoft Windows provides instrumentation for system objects such as devices and applications through the Windows Management Instrumentation subsystem. Devices that are Windows Management Instrumentation Query Language (WQL) certified provide instrumentation methods to allow the device drivers to be queried or controlled via the WMI APIs. Applications can also be instrumented to provide better Break/Fix information. While the instrumentation of these objects does come with a slight performance penalty, the ability to fix problems automatically is worth taking the hit.

To fix a problem, the system may have to connect to a peer or server to retrieve new software or parameters to fix it. However, the fault may be due to a problem with connectivity. In this case, it is likely that the

system cannot contact a server or peer system to get help, so the problem must be fixed locally. Once a connection is established, the autonomic services available to the client system can be extended with a server.

## Proactive Monitoring

While Break/Fix solves problems when they occur or immediately after, proactive monitoring attempts to identify problems before they happen to avoid having to invoke the Break/Fix mechanism. Proactive monitoring can detect when certain network connections are slow or where the connection quality had degraded over time, and attempt to repair those connections. It can detect when system resources are being used up by certain programs, and attempt to fix the problem before the system grinds to a crawl. Proactive monitoring can detect when a disk drive is getting full, or when a user appears to be stuck figuring out how to do something.

In Microsoft Office, the Office Assistant attempts to guess what the user needs help with, and frequently offers to perform an operation automatically, such as adding bullets to a list. For example, when the user is observed entering text in Microsoft Word, the Rocky object wags its tail and sniffs the ground. If data is being entered by a fast typist, the Rocky object pants as if it was working hard. If the user stops entering data for an extended period of time, the Rocky object lies down and goes to sleep.

A simple example of proactive monitoring can be found in Microsoft Office. When the user performs a repetitive operation two or more times, the Rocky object will suggest a shortcut to performing the same operation if one exists. This simple form of proactive monitoring could be extended to periodically check the size of the Word file in memory against the available disk space to make sure there's enough space available to save the document. It could monitor the mouse movement to determine if the user is having trouble locating a small object with the mouse, such as a single pixel in a CAD program. The system could automatically adjust the mouse positioning by introducing a filter or adjusting the mouse resolution to

provide more accurate pointing. When the system observes that some time has elapsed since the user required this feature, it could automatically remove the filter or revert back to the original resolution. The user should never have to traverse a half-dozen windows and dialogs to find the mouse settings, and then do it again while trying to remember what the last settings were.

The system could monitor network traffic, watching for bandwidth degradation or bottlenecks. For example, when the response time from a particular name server becomes a problem, the system could attempt to locate another name server with less traffic and route our requests through that name server. At a later time, the system would revert back to the original name server to see if things had improved.

Another benefit of proactive monitoring is that the system can become more user-friendly by adapting to the way the system is being used. User who perform operations a certain way could have their user interface “custom fit” to their habits. For example, some users open up an application by launching the application using a shortcut, then open a file for the application using the standard File->Open menu option. The user interface for the installed programs could be modified to always include a File-Open menu item whether it exists in the program or not. Other users may never start applications directly, but may invoke them indirectly by clicking on the file type registered for that program. In this case, the system could hide those options and not expose them in the application’s menu. While this is not a compelling feature, it’s easy to imagine other features which could be implemented based on the user’s work style.

### **Device Instrumentation**

Windows instruments devices using WMI. Device manufacturers that provide WQL compatible devices are required to supply device drivers that support WMI. Using WMI, Windows and applications can query the state of any managed device, and can perform operations on those devices such as disabling a device or changing a device

parameter. WMI generates events which can be used to determine what action to take. A program registers for certain events or classes of events, and then acts upon those events or passes them on to the next event handler in the chain.

WMI is primarily a reactive Break/FIX mechanism in that it doesn’t provide any proactive monitoring of a device that might cause some other action to be taken. It will report errors or problems with a device but will not undertake any corrective action by itself. Almost all of the parts are there to extend WMI to be more proactive. While WMI is rich with features, it is based on the outdated COM architecture that has been superseded by the new Common Language Runtime (CLR) model. WMI is difficult to learn, and requires a great deal of time to master. It requires programmers to learn yet another underlying technology which should be part of the system infrastructure and exposed through standard frameworks. This instrumentation should be represented with design patterns and as provide as standard application development templates as part of the development tools.

Generating events is only part of the story, however, because unless there’s a mechanism in place to analyze and correlate those events, the event information is not very useful. What’s missing in WMI is a rules-based event monitor that can determine if something if something will likely fail based on the sequence and proximity of certain events, and what can or should be done about it. The ability to predict a problem or failure before it occurs can save users valuable time and will help the lower the cost of ownership.

### **Application Instrumentation**

Like devices, applications can also be instrumented. Application programmers can create custom application events that can be handled by WMI and passed on to custom event handlers. This requires extra time to be allocated for designing, coding, and testing the event generation and handling features of an application. This makes for a lengthier debugging process to make sure that all of the events are handled correctly. Instrumentation is something that should be

designed in to an application at the beginning of the process rather than as an afterthought. Applications can get notified when a device goes away, or when a resource is freed up. Events are generated synchronously, so the application can continue running until an event occurs.

In current versions of Windows, WMI provides developers with methods to instrument an application, but that's only part of the job. What's needed is not only a way to instrument the application, but to coordinate and map those events into a set of actions that can potentially remedy the situation. The solution may be a local setting that can be fixed without contacting a server or peer. In this case, the client system would just fix the problem and continue working.

### **On-Demand Wizards**

An On-Demand wizard is like a personal servant that is commanded by the user to perform a particular task. It is a user-initiated action that requires an immediate response. A proactive monitoring component may be unaware that the task even needs to be performed, or the task might be something that is not part of the normal operation of the component. An example of this is getting connected to the Internet from inside an enterprise. As previously described, this can be a daunting task. Using DHCP, the system can retrieve the address of a local name server, the local gateway, and obtain an IP address. This assumes, however, that the network adapter is properly configured, that the correct drivers are loaded, the correct protocol software loaded, and the configuration set to DHCP. If these options are not correct, the system won't even be able to get an IP address. Because connectivity is not always possible, the drivers, settings, and protocols necessary for establishing connectivity must reside on the client machine.

### **Collaboration**

Collaboration assumes some type of connectivity, either to a server or another peer. In the case of a server, the client can contact the server for information about a particular application or device driver. For example, if a program is started but then quickly fails or consistently fails during a particular operation, the client system can contact the server to find the latest level of the offending program, and to see if any patches are available. Microsoft Update performs part of this service by contacting the Microsoft Update server and comparing the client software levels with the levels on the server. If a newer version is found, it is either downloaded and installed or downloaded for installation at a later time. The program could contact a peer system to inquire if the peer is aware of a fix for the problem, or if perhaps the peer system has the fix available for download.

It is possible that the problem could be one that the event monitor does not recognize, or that it has no fix for. In this case, the client system can contact the server to see if it has a resolution to the problem. The client system could also contact a neighboring system on a peer basis to find out if perhaps the same problem had been encountered and if so, how the problem was fixed on the peer. For example, if the user plugged their system into a network and found that they could not access the Internet outside a corporate firewall, the user's system could broadcast a help inquiry to neighboring systems asking "do you have Internet access?" If the answer is "yes", the client could then ask the peer for its network settings and configuration and use that information to update the client's own network settings.

### **Future Autonomic Services**

While we can do a fair job of making systems and software more autonomic using current technologies, we could do a much better job with the addition of some extra features. We will classify these features using the same categories discussed earlier.

- Break/Fix

- Proactive monitoring
- Device instrumentation
- Application instrumentation
- On-Demand Wizards
- Collaboration

### **Break/Fix**

Instead of requiring applications or drivers to monitor the health and state of the system, the operating system should take a more proactive role in fixing things that it has control over. Memory exceptions, illegal instructions, and other system failures should be handled by the operating system with no user intervention. The operating system should handle those faults and write them in a fault log for later viewing. Users should not be presented with meaningless dialogs or message boxes that the user can't do anything about. The level at which these errors are reported should be configurable, and allow the user to decide when they want to be notified.

Break/Fix behaviors should be pluggable strategies. This would allow a system to be optimized for use in a battery-operated environment, or perhaps optimized for use in a highly secure environment. Strategies could be customized for a particular environment or enterprise without requiring changes to the engine that implements those strategies.

Some strategies should be autonomic in that the strategy should evolve over time based on the use of the system. The user's habits and workload would be monitored and used as input to the strategy engine. The strategy engine could then perform a dynamic strategy update based on the way the system is used. Power consumption is an area where usage habits could result in a power savings by turning off components that are not likely to be used.

### **Proactive Monitoring**

Future proactive management should include a class of non-critical events that are sent to subscribers to those events. Applications should have the ability to

register for these events by class and use them to dynamically modify the behavior of the application. These modifications might include the ability to dynamically change the application's user interface as resources come and go, or to modify the look and feel of the application based on the type of user. The application may also need to set certain system parameters on a per-session basis, and have those parameters in effect only for that session. Instead of parsing the event log to determine what requires attention, the system should generate events asynchronously and allow the application to take action or just ignore the event. For example, the application might request to be notified when the disk drive reaches 60 percent of its capacity, or if network bandwidth falls below 1 Mb/s. Each application should have the ability to set its own threshold at which the event will be triggered as each application has its own unique requirements.

The ability of an application to modify its behavior based on a series of events or pluggable strategies does not currently exist in today's operating systems. Adding these types of features would make the system autonomic, and provide a platform for autonomic applications to run. However, it is important that these features be implemented in frameworks and run time components so as not to complicate the design of autonomic software and applications.

For example, an application might vary the rate at which it sends serial data based on some external factors. The programmer should be able to write an application that sends the serial data without regard to the rate by simply calling the method or function that sends the serial data. The code that implements the API should be autonomic, and could change the rate of communications based on a set of events or a particular strategy. Encapsulating the autonomic functions in the data transport would allow the application to be written without requiring the program to learn new technologies and interfaces.

### **Device Instrumentation**

Future device instrumentation should include performance metrics for each device. For example, it should be possible for applications to retrieve the average transfer rate, the ratio of seek time to read time, and the average read latency of a disk drive. The ability to instrument these devices should be configurable as this type of instrumentation will add overhead to system operation. This instrumentation should not be something that the programmer needs to explicitly invoke; rather it should be built into the current application run time and easily available to the application. It should not require the user to learn complex new technologies such as COM, but should provide a common set of frameworks that expose the information in a way that seamless to the application developer.

Programmers writing software in Visual Basic, for example, should not have to learn COM, but should have the information exported in a way that makes the information easily available to a VB application. Likewise, developers who write in C or C++ should also not be required to learn new technologies, but should be provided with a seamless set of functions and methods to provide access to the instrumentation subsystem. For example, a programmer writing an application in the C language should not have to learn the C++ language, COM programming, and OLE data types just to get access to the instrumentation subsystem.

### **Application instrumentation**

Applications should have the ability to query data about their own performance so they can adjust their own parameters or operating mode dynamically. When an application is launched by the operating system's program loader, the loader should have an option to create an instrumentation object which is linked with the application for the life of the application. Using this object pointer, the application should be able to easily access its own instrumentation information with simple method calls. The programmer should not have to initialize or use complicated subsystems to get at this information. Having the information handy and easily accessible will encourage developers to use the data to make their applications autonomic. For example, the Windows operating system is currently instrumented with the Windows Management Interface, or WMI. While rich in features, WMI requires a fairly in-depth knowledge of Common Object Model, or COM, as well as a good understanding of COM or OLE data types.

### **On-Demand Wizards**

As operating systems become more complex, the need for on-demand wizards will increase. Users will elect to bypass menus and property settings in favor of quick shortcuts for certain operations. For casual users and frequent travelers, the on-demand wizard offers an easy way to perform complicated tasks with little or no knowledge of the underlying technology. An example would be getting connected to the internet from a new or unfamiliar location.

Large enterprises use various types of hardware and software to provide security for their internal networks including firewalls, Virtual Private Networking (VPN) connections, and various types of encryption. Users inside the firewall need to access the internet, but this can be a potential security risk so access is granted to the Internet by some type of proxy server. When the user attempts to view a Web page, the HTTP request is sent to the proxy server, which in turn performs the request and sends the data back to the requesting system. The name of the socks server often

varies by location, so a user traveling from one location to another might not be able to connect to the Internet from inside the firewall without changing the name of the socks server in the user's configuration. A simple wizard or button labeled "Connect Me Now!" should be implemented to automatically connect a user's system to the Internet without requiring the user to know the name of the local socks server or have knowledge of how to change that parameter in the client system.

When a Windows XP user double-clicks on a folder that contains pictures, they are presented with options about how to display the contents of the folder. They can select if they want the contents of that folder displayed in a traditional file view, or as thumbnails. In the thumbnail view, the user can browse the thumbnails, then view, edit or print them. There is also an option to save the picture or pictures to disk. However, most users just transfer their pictures from a camera and save them to an album, not a file. Yet the XP wizard has no option to add the photo or photos to the user's photo album.

The other thing that users like to do is to send pictures to their friends and relatives. A wizard should allow the user to easily send a picture or pictures to someone without knowing what type of file it is or where it is located. For example, when the user checks the properties of the picture, one the options should be to send it. The wizard then would pop up a list of recipients from the user's address book and allow the user to enter a short note and click the "send" button. If the system does not have an active network connection, system should queue it up to be sent later when a connection exists.

## Collaboration

Within a large enterprise, a great deal of specialized knowledge exists but is often not shared because no one knows specifically where to look for it, or perhaps isn't even aware it exists. In a large company, it is not uncommon for a project team to be unaware of a similar project underway in the same company. Once together, however, they can discuss common problems and solutions, and perhaps share some technology and knowledge with each other.

Like the project team, systems within an enterprise also have specialized information that can be shared with peers to help solve problems or provide assistance in location resources. For example, suppose a client could access the corporate intranet but not the Internet because it did not have the name or IP address of the local SOCKS server. It could ask one or more of its peers if they have Internet access and if so, what they are using for their connectivity parameters. The system could then configure itself and get connected with no operator intervention required.

Another example might be the sharing of bookmarks among peers. Bookmarks usually contain a great deal of site-specific information, such as URLs for computer repair, telephone support, shipping, receiving, and purchasing. These bookmarks can be categorized, aggregated and shared among willing peers.

## Conclusions

Writing autonomic software requires the programmer to take a close look at the functional components of their code from a non-technical perspective. It's easy for us to detect errors and report them, leaving the resolution up to the user. We do this because it's what we understand, and because we also understand how to resolve the problem. It's also easier to write code this way because it requires the least amount of thought. Writing autonomic software requires that we not only understand the problem, but that we understand enough about the solution that

we can correct it ourselves without requiring the user to fix it.

The operating system should provide a set of functions to make developing autonomic software easier. There should be standard access to system instrumentation information, and a set of tools and frameworks to provide autonomic behavior without forcing programmers to learn new technologies, languages, and architectures. Even without these tools, we can begin writing autonomic applications right now. We can even go back and modify current applications to be more autonomic, just by looking at them from a functional point of view.

Look at every place you request operator input to see if that input could be filled in automatically. Try and reduce the amount of operator interaction. Pay careful attention to menus, dialogs, and fields to make sure they really make sense. Have the program evaluated by several users that have never seen it in operation. Have them install it, configure it, and use it without a manual. Examine carefully every error message to see if it makes sense, if it is really necessary, or if it could be replaced with software that could fix the problem for the user. If an error message is required, make sure it makes sense, and that it's written in terms that the user can understand. Be kind, as users are often intimidated by errors that make them feel that they've somehow done something wrong.

Be proactive. For example, if your software saves large amounts of data to disk, check

the disk space when the program is started, before the user tries to save several hours of work and can't because there's not enough space left. If your software uses the serial port, check to see that the port is available before they try to send or receive data and get a timeout error. If your software will be issuing a remote procedure call to another server, make sure that connection exists before making the call and hanging the system.

Writing autonomic software requires extra effort and thought, but users will find it a more pleasurable experience to use. You don't have to wait for the operating system to provide all of these features. With a little extra work, you can begin incorporating autonomic features in your code today.

**Steve Mastrianni** IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: [stevemas@us.ibm.com](mailto:stevemas@us.ibm.com)). Dr. Mastrianni is a senior software engineer currently writing autonomic software. He joined IBM Research after running a software development and consulting company for 10 years. He has authored two books, several papers, and over 70 articles, and holds a Ph.D. in computer science. He has filed over 60 patents with six issued, and he prefers writing software instead of talking about it.