University of Geneva

Departement of Computer Science

Management-Enhanced Software Components

Diploma Work

Matthias Wiesmann 1996-1997

Responsible Professor: J. Harms

Responsible Assistant: V. Baggiolini

I would like to thank the following persons for their help and support for this work:

Sylvie Roux , Isabelle & Anne Roissard de Bellet, for proofreading and checking that what I wrote made sense.

Vito Baggiolini for his help, counseling and ideas.

Patrick Rohr for the interesting discussions.

My Family for their support.

Table of Content

Introduction

1) Context

The aim of this project was to create a framework for complicit managed systems. A managed system is a one that is overlooked by a Manager that takes preventive or corrective action in case of trouble. This Manager can be an autonomous program or a human tool.

Today, more and more programs have troubles: buggy software, changing, non uniform or unreliable hardware. In most actual systems, the Manager is a human being, but with the growing complexity of software and the vast amount of installed computers, human Managers are a scarce resource, so most corrective action has to wait for the next bug-fix. Also such unstable software is most unsuitable for unsupervised operations (servers, automated systems).

There are two approaches to add the Manager to the system: the non-complicit and the complicit approach.

1a – Non-Complicit Management

A non-complicit approach would be to build a Manager that would handle the system's trouble without any inside help. The components that build the software do not offer any cooperation to the management tasks.

1b – Complicit Management

The idea of **complicit** management is to build the software with the concept of management in mind. Each piece of the managed software cooperates with the Manager.

2) Premises

2a – Application Field

Making a stable application is not always possible, nor always wanted. In this paper we will make the following assumptions about the application that we want to manage.

- The application is complex.
- The application does no time critical calculations.
- The application uses or communicates with many external elements: other applications, servers, libraries, networks resources. All GUI (Graphic User Interface) applications fall in this category.
- The application is written in a modern code, it is structured in small components and manipulates structured data.
- The application will be built with complicit management in mind, and will not restrict access to its internal state (security problems are beyond the scope of this paper).

2b – General Hypothesis

There are many kinds of problems that can make an application unstable or unable to function properly. The causes of those problems can be many: hardware failures, wrong conception, inadequate context, complexity, insufficient resources, improper use etc.

In this paper, we will concentrate mostly on small problems. The first reason for that is that small bugs are numerous and can render a complex application useless. There is no point in building a complex system able to counter multiple lock problems if it crashes because of a memory leak. The second reason is that most big problems manifest themselves as small problems.

There is another way to view this. All programs are built on increasing abstraction levels. So every problem is a simple problem at the right abstraction level. The trick is to try to solve the problem at the right level.

2c – Complicit Programming

The other important assumption in this paper is that it assumes complicit programming. This means the code must include special management handling sequences. Even if it is possible to add management handling functions afterwards, it is far more efficient to construct the whole system with management in mind.

Cooperative Schemes

There are a many of cooperative schemes around. Cooperative multitasking is a typical example, instead of having the OS interrupt one process in favor of another, each process voluntarily gives up the control to the next process.

The main advantage over preemptive multitasking is that the system is simple and light. A process that is idle simply releases the processor. This scheme also prevents a process from being interrupted in critical places or in system calls.

This system works well if all process comply to the rules. But if only one process refuses to release the CPU and the whole system is locked.

The whole architecture presented in this paper works on a cooperative basis: different components cooperate through their Manager to correct problems as they occur. The overall stability of a system will depend of the behavior of all component active in problematic situations.

This means, when building a managed system, that a special care must be taken to comply to the coding interfaces, but also to the management *spirit*.. Technically, a managed component can be completely bugged, trash regularly its data, do nothing on management calls and still be considered as a managed component.

Management is not a way of avoiding proper programming…

3) Terminology

In the rest of this text I use the following terminology to distinguish the different error types.

3a – Problem

A generic state defining an unwanted and unexpected situation that can endanger the program's execution. An unsolved problem is an error.

3b – Error

A particular problem that prevents a process from executing further. A **Critical Error** is one that endangers the whole system.

3c – Failure

A special kind of error meaning that a process tried to offer a service but was unable to and announced it.

3d – Warning

A problem that was an error but could be corrected and thus the process was able to resume its operation. The condition causing the error was not corrected and so other problems may arise.

3e – Notice

A problem that was an error but could be corrected. The problem was not serious and no further harm should arise from it.

3f – Handler

A code block that handles unusual conditions, typically problems.

4) Convention Used in this Text

4a – Typographical Conventions

`The Courrier Font is used for:`

- path names and other machine specific names
- computing and string examples
- code examples
- computer messages
- package and libraries names
- function and methods names
- object names
- language statements

The Times Roman font is used for:

- references
- footnotes
- Figure information

Management Basics

1) Errors

One of the primary aspects of management are errors. Without errors, no need for management. In an ideal world of programming, without bugs, hardware failures or wrong human manipulations management do not exist.

To build a complete management system, we need therefore to define and understand the nature and structure of errors.

1a – What is an Error?

The basic definition of an error would be something like: an unexpected event or condition. Something the end user wished he'd never see. The problem lies in the *"unexpected"* word. What should be expected and what should not. Ideally the programmer should foresee everything that could possibly happen in his code. This would be perhaps be possible in a stable environment with very small code, but certainly not in the very complex and ever changing computer world we face today. So the definition should be changed to something like an event or condition that could not be *reasonably* be expected.

Dealing with such errors could therefore seem absurd, as it implies programming the unexpected. In fact most of the time, there are no spontaneous errors. Most bugs are more easily explained by *Murphy's law* than by the hand of God…

1b – Error or Communication

According to our definition, an error would never occur during the normal course of events. Is it really true? No it isn't. Often errors happen but not as an illegal state, but merely as a message, an indication of a possible event.

Take for instance this piece of program. The program checks if a given file exists, if it does, it reads its value, if not, then it uses the default values. Most codes would not check if the file exists but try directly to open it. If this opening operation fails, it uses the default value. In this case an error state (trying to open a file that does not exist), is used as a message to reflect a perfectly normal state (that the file does not exist, perhaps because the program is launched for the first time) and does not reflect a real error state.

This attitude, doing things without checking first and patching the eventual error, is very common in programming, because it is fast and simple. Virtual memory and API (Application Program Interface) implementation[1] have been built in such ways. This behavior is certainly acceptable in low abstraction operating system code, but not in high level programs.

There are in fact *true* and *false* errors that coexist in a same system/model with no easy way to distinguish them. The error communication channel is in fact used

[1] The Classical Macintosh Toolbox is implemented on the 68000 Unimplemented Instruction Trap.

to transmit information between different pieces of code, for instance an API and the application.

Correction and analysis are therefore rather complicated: correcting *true* errors would make the program more stable, but correcting *false* errors would scramble the program's way of communicating.

Clearly, proper programming should avoid using error states for normal operations and no healthy program should rely on error signals to perform well. Using error signals only for errors case is a first criterion for a manageable program.

1c – Defining Errors

Even in modern Operating Systems, errors are generally simply defined as numbers; with any luck, some text message is associated to this error number. Some system try to structure those numbers a little bit, allocating some errors ranges to given Managers or subsystems. Some systems, like the Java virtual machine, declare errors in a more structured object hierarchy.

Sadly, all those systems fail to transmit what gives an errors its meaning: **the context.** The context is what gives an error its meaning. Trying to treat errors simply by their identity is nearly impossible. How many human users where unable to decide what to do in case of a disk error: `abort/fail/retry`? Most current error dispatching schemes, including exceptions rely on non-local error treatment, meaning the context is lost.

Including context information inside the error itself or staying in the context is the important requisite for a managed system.

1d – Finding Errors

The main problem when dealing with errors is to find where they *are*, and not where they *seem* to be. An single source can cause many effects at different positions and times in the program. One thing that makes bugs hard to find is the great distance between source and effect. So if a bug has survived the debugging process, it is probably not obvious.

So the question is, what is an effect and what is a cause? Finding the formal link between cause and effect would require complex methods and could well fail because causes are also effects: take the case of an overloaded network line, because there is much traffic, the line is overloaded, but because the line is overloaded, some transmissions fail and must be repeated, generating more traffic thus overloading the line more. What is the cause? The overloaded line or too much traffic? Both elements are cause *and* effect.

If the system is unable to determine if an error is a cause or an effect, it will not be able to report it correctly. Correcting such an error could not be possible, or could even be dangerous: if two corrections are applied to the same problem, unpredictable interference may appear. For instance, if a server process crashes, many clients may notice it. The fact that they cannot access the server is an *effect* (the *cause* being that the server crashed). This means that the reconstruction of the server must be coordinated, or else each client will rebuild its server.

1e – Error Localization

Preconditions, Postconditions…

Those who are acquainted with formal system may recognize the Interface Errors, they are simply violated preconditions. Assertions and postconditions typically yield Internal Errors.

When more than one person is involved in a problem, it is always the other's fault. One way of solving this problem is with a contract that specifies clearly who does what. We can see all the component interactions as request for services: one part of the system wants another do to something. There is therefore a client (who wants something) and a server (who can give something).

Interface Errors

Sometimes the client wants to do something impossible. Asking for a new car in a restaurant will not work. The problem is probably not in the restaurant, but in your request. This is a interface error, a request for something impossible or inappropriate. Asking the printer to play a sound file is a typical interface error.

In a case of an interface error, the request (the contract) has been violated by the client (he is expected to require possible services), so this error is his responsibility.

Internal Errors

Sometimes the server accepts a request for a service, but fails to do its job. The reasons can be many, but only the server is to blame. No other component is responsible of the problem. This is a internal error.

Sub-Request Errors

Sometimes, the servers need the service of other components to do the job. A file system component may require the service of a network component when using remotes disks. What happens when the network component fails? The file system component will be unable to complete its request, but its not its fault, the network component is to blame.

This is a sub-request error: when a component fails to do its job because another component failed.

2) Management Strategies

There are three approaches to make a program work seemingly without problems: prevention, protection and correction. All three approaches are legitimate and should be used together. Those three approaches are mostly used during the building of the program.

Prevention or Protection?

In fact, Prevention and Protection are very complementary. In fact most preventive measure can been seen as complicit protection: many preventive measure include some protection code. This protective code is either explicitly included by the programmer, or implicitly by the compiler.

Sometimes it is necessary to correct a program after it has been built, this is called patching. Patching is not a corrective measure, it is a way of modifying existing code. A patch in itself can be preventive (more stable code), protective (a data protection scheme) or corrective (a corrective code sequence).

2a – Prevention

A great deal has been done to prevent the coding of bugs and errors. Most prevention is done in the program conception phase. Some of the conceptual preventive measures are enforced by programming languages.

Program Conception

The first bugs appeared with the first programs. Since, many techniques have been proposed to avoid the coding of errors. Many of these techniques involve the idea of "good" code. Comments, structured code blocks and data, assertion checking, structured programming with increasing abstraction levels are the classical criteria for "proper" code. Such code is indeed less bug prone.

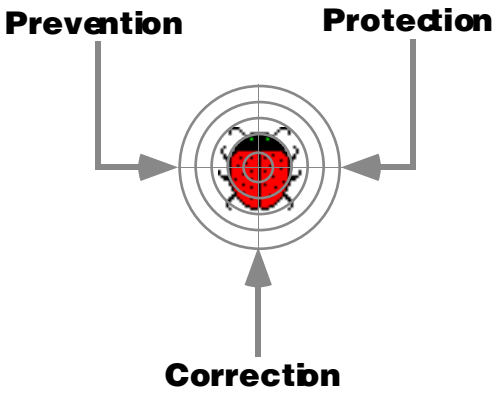

Figure 1 - the three approaches

Another idea was to be more precise in the conception preceding the coding. Carefully built plans insure a clear view of the working of a program and avoid conceptual flaws. Clear specifications are a typical example of conception tools. Many specification languages and methods have been proposed and are regularly used. The problem is that in the fast evolving computer world, the original specifications are rapidly outgrown by new developments and applications.

Code Reuse

Another recent trend in programming has be to try to increase the reuse of tested code instead of rebuilding each application from scrap. On the conceptual side, patterns are tested solutions for classical programs. Both code libraries and pattern systems are increasingly used. This means the reused code has been tested and debugged more thoroughly and so should be less bugged.

The Size Problem

Intuitively, the more we code, the more bugs we produce. This means, the number of bugs in a program is linearly proportional to the amount of code the programs contains. This could be expressed by the following hypothesis.

$$b = \lambda B$$

Where b is the number of bugs, and, B the size of the code. The λ factor depends of the programmer and the programming environment (the bugs per byte ratio).

If proper conception and coding technique may reduce the number of errors inside the code (the λ factor), there will always be bugs.

The fact that programs get bigger and bigger exponentially, means that we should improve our programming technique en environment exponentially to keep the number of bugs even.

Program Rejuvenation

Another preventive feature is software rejuvenation. That is, quitting the application at idle time and launching it again immediately in a clean state [1]. By doing this, the system *forgets* all possible data corruption that could have occurred and thus is more stable. Rejuvenation should typically be done at idle time, when the application is not doing any work. For this scheme to work the application must be able to resume at a point stored in a log file.

Programming Language

Many languages offer features to build more stable code. Many languages encourage the use of certain code structures and forbid others. This preventive feature is strongly associated with the language definition. Data typing, block code structures, abstract references and exception handling are considered typical features.

The other way of making stable code is by adding implicit checking instructions along the compiled code. This means the compiled program will check himself at runtime. Range and type checking code are typical examples.

Some time the responsibility of runtime checking is transferred to host environment where the program will run. Instead of having the program restraint itself from doing illegal things, the host checks if the client does legal operations. This is a protective approach.

2b – Protection

The protective approach tries to protect some part of the system from problems occurring in another part of the system. The aim of a protective approach is to make sure that the part that crashes doesn't take the whole system with it. Usually, protection is implemented at the OS (Operating System) Level. The ideal OS offers some safe services (typically memory and files).

Memory Protection

A typical example is the memory protection scheme that is used in many operating systems. The principle is simple, each process has its own memory space, and cannot access other processes memory space. When an access is done outside the memory space, the process is killed.

The problem is that this memory space is usually defined by a 32 bit pointer. The more the processes occupies its memory space, the less likely it will be to cause a memory protection exception, and the more likely it will corrupt its own data space.

One problem with this approach is that in the event of a problem, the faulty element is destroyed, its state is lost and other elements depending on it may be unable to function. The other problem is that the process is protected against other processes, but not against itself. This means a process can completely trash its data, without the system complaining.

A more advanced approach would be to let the program do some "last ditch" measures to save its state and data[2]; somehow having a word processor crash is not dramatic, it is loosing the document that is…

With the advent of user-friendly interfaces, the error message have gone from a black screen with hexadecimal gibberish to a color dialogue with sound and shadowed icons announcing that a program has tried to read memory position at `$000FFF0` and therefore was terminated. If having the band playing while the ship sinks certainly improves the user-experience, most people would prefer having more lifeboats…

2c – Correction

This approach is less implemented, perhaps because it implies that both previous approaches are not sufficient to make stable programs, but probably because it requires more work. Before killing a component when a problem occurs, some corrections system should be activated to try to save it.

How do you a correct a Bus Error?

Somehow, many crashes seem to be assembly level problems. How can we correct such low level errors? Most of the time we can't.

In fact, an address error (bus error, segmentation fault, general protection fault etc.) should not be taken as the error itself, but as an effect (clearly a pointer was wrong). A component causing such a error can probably not be corrected, but the rest of the system should be saved.

With the advent of languages without pointers (like Java) this sort of error should be replaced by more explicit (higher level) errors which will be more manageable.

A corrective approach would seem a first glance very hard to implement, given the complexity of most computer systems. But if you observe an experimented user, you see that many small problems can

[2] Unix system typically make a "`core`" file containing a complete memory image of the crashed program.

be fixed by very simple manipulations. By implementing small heuristics that contain only small corrections schemes, the overall stability will increase.

Alternatively such a scheme should permit to emulate services in case of insufficient/deficient hardware, a problem that often arises with the fast evolving computer world.

Here are some recovery schemes that are or could be implemented:

- In case of network problem, fall back to a more primitive network system[3].
- In case of insufficient memory / processes, kill inactive graphical programs that have no open window[4].
- In low memory situations, use slower, less memory hungry algorithm.
- If a file cannot be found at a given place, search the file system for a similar file.
- if a file is in the wrong format, launch a translating utility and open the translated file[5].
- Automatic correction of some parameters, especially when those parameters are error prone, like editable text fields[6].

2d – Patching

What is Patching?

Patching is a technique that consist in inserting corrective code to hide a defective piece of hardware or software. Patching is not so much an approach to make bug free software than a corrective measure. Nonetheless, it is such a common practice to patch deficient hardware or software that it cannot be neglected.

The Trap Idea

One concept that has enabled most patching is the trap system. When an unforeseen event occurs, a trap is raised. This trap is then treated like an interrupt, a request for special service.

Some low level environment let the programmer redirect some traps to handle errors by himself.

Patches can be preventive, protective or corrective. A Patch could replace a dirty piece of code by a more clean one (preventive approach), or enhance the memory protection scheme (protective approach), or add specific recovery code (corrective approach).

Hardware patching

Hardware and firmware are probably the most often patched pieces of the computer, from the CPU[7] to ROMs. The reason is simple, by nature hardware is rather costly to change. This means hardware is more often built with patching in mind than software.

---

[3] Macintosh Computers fall back to Localtalk networking when other Network fails.
[4] When a low memory situation occurs, the Mac OS asks the user if if can quits inactive applications in.
[5] The Macintosh Easy Open and some disk-compression system use this scheme.
[6] The Netscape Naviagator is able to correct some uncomplete URLs.
[7] Some version of the Intel Pentium Processor needed a software patch to correct a bug in the floating point pipeline.

With the increasing size of both operating system and application programs, correcting bugs and errors is getting harder and harder. This means that the need for a way of inserting corrective code without installing a new version of the application / OS is getting stronger and stronger.

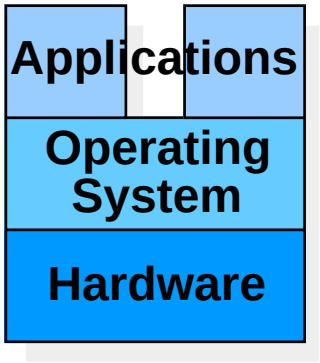

Figure 2 - Abstraction Levels

Hardware patching is typically done at the OS level. The goal of the OS is to give an united abstract view of specific hardware. The OS hides hardware particularities, and so can also hide errors.

Software patching ?

This can be done because the OS resides *above* the hardware (see Figure 2), the application code only sees a very simplified version of the hardware. This way, the hardware is patched from above. So, logically, to implement corrective code it should be done from above. So for the problems inside the OS, this would mean inserting corrective code into each application which is not very practical.

At the Application Level, the only thing above the application is the user, which means that the end user does the patching code (usually fiddling and swearing). Today in fact, the user is "a living patch" for the application he is using, because he has to work around the problem of the application.

In clear, the idea of patching the application and the OS from above is not good. Patching them from beneath is not a good idea either: it means adapting the hardware and for the OS and the OS for the applications. When this has been done, it was solely for compatibility issues and it was never efficient or stable.

So it appears that OS and application must be patched at their own level. This means their must be a way of inserting corrective into an existing code sequence without recompiling it. This possibility has existed for long in OS: device drivers are pieces of code that handle a particular piece of hardware and are added to the OS as necessary.

In applications, the same idea has appeared with plug-ins, pieces of code that perform special operations for the application. For this idea to work, the application (or the OS) must be built with those plug-ins in mind. If this is done well, the program could support very advanced customization[8].

[8] Adobe Photoshop 2.5 supported a plug-in that permitted it to work on a different processor.

3) Management Elements & Mechanisms

Now that we have manageable components, we need the final piece: the Manager. The Manager is the piece that handles problems. As in the human world, an ideal Manager should offer guidance and help in case of trouble but at the same time be as unobtrusive as possible.

The Manager is also responsible of user-interface in case of problems. The Manager should also initiate preventive measure when the CPU has some free cycles.

3a – Managers Structure

The Manager system can be an autonomous program or a tool for human control. In both cases, the Manager system will try to recover problematic situations. Either the human operator or the manager's algorithm will choose the right tool to recover the problem.

A management system should be invisible, and the application should be able to work without it. All Managers should share the same interface. This way a Manager could be substituted to another at any time.

Managers should themselves be organized in a flexible way. Many components should be able to share a Manager (for instance a client and a server could have a common Manager, that would handle the correction of problem at the interface between them). Managers should be able to communicate and share a common log.

Most important of all, Managers should help to correct problems, and should therefore be very stable. One easy option for this, is to make them manageable, and like everything else, to structure them into components.

3b – Management vs. Debugging

Similarities

The Dylan Language

In the Dylan language, if no handler is specified, the exception is directly transmitted to the debugger. There must always be a debugger, even if all it does is a core dump.

One interesting aspect of Manager systems is their similarity to debugging systems. Both debugger and Manager hook in some points of a component and intercept error messages. Here are some features that are on the wishing list in both Manager and debugger:

- Logging of a component's normal and exceptional events.
- Access to a component's general state.
- Access to a component's resource usage.
- Interception of error signals.
- Possibility to order a rollback or a retry for some operations.
- Possibility to enable and disable a low resource consumption mode.

Both the Manager and the debugger depend on some complicit[9] code to be able to work correctly. In the traditional programming ways, the debugging code is removed for the final working version. Interestingly, a recent trend is to include the debugger scheme lower and lower into the system: into the API[10], the OS[11] or even the language specifications[12].

Differences

Still Manager and debugger are different components with different tasks. The debugger is typically a complex tool meant for program conception. The Manager is an auxiliary program that helps existing programs to work.

The debugger has to have complete access to the internal states of the whole software, even private data. The Manager should only have access to certain data. In fact, the privileges and accesses a Managers has are a subset of those of a debugger.

Unification

By unifying the debugging and management hooks it should be possible to replace the debugger by a Manager once the development phase is finished, without changing the actual code.

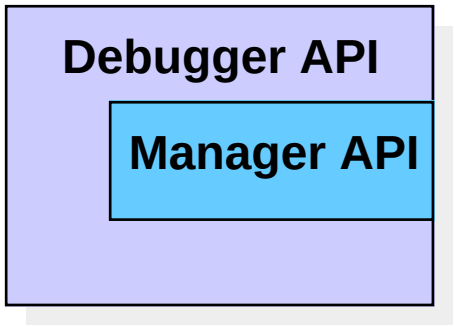

Figure 3 - Debugger & Manager API.

The management API is a subset of the Debugging API (see Figure 3). Building such a unified debugger/Manager would require a very low level conception and is beyond the scope of this paper.

3c – User Interface

One tempting idea in case of trouble is ask the user what to do. This is often a very bad idea. Firstly, it makes no sense if there is no user, like with servers or delegated tasks. Secondly, with the advent of popular computing and advanced interfaces, the number of users who know what's going on has drastically dropped. In fact we should consider three sorts of users, the normal user, no user, and the advanced user.

Normal User

This user has perhaps some understanding of what the program does, but certainly none of the underlying structure. This user will certainly not understand unclear messages or technical gibberish (see Figure 4). The more unclear the message will be, the less likely it is to be read. Messages should be clear and concise (see Figure 5).

---

[9] Most debugging tools require a special compilation mode in order to work.
[10] The Macintosh Toolbox includes some debugger commands.
[11] The Windows Operating System exists in a debugging version.
[12] The Java API offers interfaces to both the Runtime and the Compile Environments. A special Breakpoint exists within the Java Virtual Machine.

Warning:
Name: Paste
Class: XmPushButton Gadget
2054-xxx Illegal mnemonic character. Could not convert X KEYSYM to a keycode.

Figure 4 - non user friendly message

If confronted with an error dialog box, the normal user reaction will often be:

- Panic
- Not read the text, click on OK.

So most of the time, throwing a dialog box to ask what the program should do makes no sense. Giving a complete problem description is of no use either: most users do not really read the error message and, if they do, it is only to transmit it to an advanced user.

Feedback

Not asking the user what to do in case of problem doesn't mean that he should no be informed. Most users have a certain experience of the time their computer takes to do certain tasks and the noises it makes. So, if a lengthy recovery is taking time, the user will notice the delay and worry.

Therefore, the recovery system interface should give the impression that the situation is under control, and something is done to correct the problem.

This means explaining shortly what is going on and show a status bar with the progression. On no account should the system give the impression it is frozen, or else the user could panic and do dangerous things (like resetting the computer).

In presence of a normal user, the system should try to recover by itself and keep a precise log for the advanced users. Somehow, it is statistically a safe bet to assume that the user is a normal user.

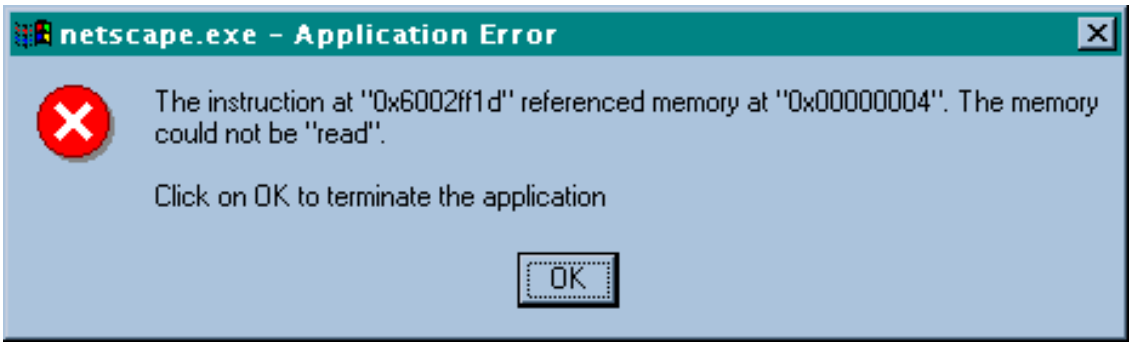

Figure 5- user-friendly message.

No User

This case happens more and more with servers and the advent of delegated task systems. In fact handling no user is like handling a normal user, but without the hassle of interface feedback.

Advanced User

The advanced user understands the structure of the system and how it works. Advanced User access should require a password or access through an discreet control (hidden button, special keyboard shortcut).

The one thing the super-user doesn't have is time. So the recovery system should do all the information gathering work and present a synthetic view of the situation. The advanced user interface should permit access to the log of the recoveries the Manager did on normal user or no user mode.

3d – Free Cycle Management

The workload of a given system always varies depending of what the users does. Personal Workstation tend to have lots of idle processing time when the user thinks or is not using the computer. A great part of the processing power is used by the screen saver to do impressive animation. During those idle period, the system could do some preventive work. Some operating system use this time to do some disk cleanup, like file defragmentation. During this idle time, the Manager should try to do some preventive work:

- **Intensive checking**: verify that all structures are in a perfect state.
- **Structure optimization**: many systems use structures that tend to become cluttered or fragmented with time: tables, caches etc. Cleaning up those structures ensures better performance.
- **Information Gathering**: the Manager can gather information to help future recoveries: check the quality of network connections, build hash-tables containing all critical files, etc.
- **Software rejuvenation**: by killing and relaunching server processes during idle time, the Manager can improve the stability of the given server, because the server restarts in a clean state [1].

3e – Recovery Objects

The Manager needs to know how to recover actual problems. Building the knowledge directly into the manger makes no sense. This would imply that all possible problems and their recoveries are known at the time the Manager is built. Even if it were possible to build such an "omniscient" Manager, it would be huge.

In fact, we do not want the Manager to know how to manipulate components that are not attached to it. This means a Manager should not have access to recoveries concerning components he has no access to.

The recovery knowledge must be stored in small objects that are loaded as needed. This way, means of recovering specific problems can be added as needed, without having to change the Manager.

Recovery Source

The recovery knowledge must come from somewhere. Ideally, every piece of software could contribute to the knowledge of the Manager.

- **the program**, typically recovery code concerning the field of application of the program.
- **the component**, typically specific recovery code concerning the component itself, or pier components that interact with it.
- **the Manager**, typically, the Manager could know simple recovery sequences (retry, fallback).

- **the user,** typically patches that correct some defect in the program could be specified by the user.
- **the World.** With the advent of the Internet and networking code (Java, ActiveX), it could be possible to share recovery code through the network. This would, of course, raise security and complexity concerns, and is beyond the scope of this paper.

Recovery Context

One important thing about recoveries is that they can be specific to given situation, to a given context. This means a recovery could be OS or machine specific.

Native Recoveries

In the case of emulated virtual code, like in Java, this means some recovery can and probably must be native (implemented in assembly code rather than in byte code) .

This could be very useful for some host related problems (like memory shortage).

For instance, if a disk write operation fails on an Unix system, one possible recovery is to write the data in the `/tmp` directory. On a Macintosh system, the `Temporary Files` directory must be requested to the Operating System. Implementing this recovery must typically be done in a platform dependent fashion.

In this case, the recovery exists in many contexts, so both platform dependent code should be hidden beyond a generic interface. Some basic recoveries, like a retry operation, can work on any platform and can be implemented in a generic way.

Because each environment will accept different recoveries, each recovery should, with the help of the Manager, decide if it can be applied in a given situation.

Recoverable Recoveries

What happens when recovery code fails? The obvious answer is to manage this failure and tell the Manager who will recover from this failure. This simple idea has many implications.

- Recoveries should be built into managed components
- Managers should be able to detect recovery loops (a recovery that fails that is recovered by itself and fails and is recovered by itself etc.).
- The Manager and the component should be reentrant: they should be able to handle several recoveries at the same time.

3f – Special Managers

If having a Manager is a good thing, having the adequate Manager is better. A Manager is in fact a small database containing recoveries and matching them with incoming trouble. Different situations may require different database structures and different matching schemes. This means different situation will require different Managers.

An application on a network component might want a very simple Manager that very few resources. A time critical application may need a fast Manager that can find a recovery in a given time. A banking application might prefer a Manager with a strong emphasis on security and stability. An autonomous application

might need a very smart Manager that could adapt itself, perhaps using Artificial Intelligence algorithms.

In any case, our system should permit each case and offer a standard interface so that each Manager could substitute to another. This way an application developed for one Manager will work perfectly with another.

3g – Checking Services

One important concept of management is checking. Many checking mechanism exist, from the assert macro in the C language to advanced debugging tools. It is clear that the checking mechanism should be linked to the management system. This way, if a check fails, the Managers can dispatch the problem.

To simplify the writing of managed programs, a checking service should be offered by the management system. This system should let the program specify legal and illegal states. If those conditions are violated, the Manager would be notified and could launch recovery code sequences.

Checking Component

One way of doing this is by defining a special checking component that does the checking work. This component can be changed and upgraded as needed. This checking component should be able to perform all sorts of assertion and precondition checking.

One of the most important aspect of the Checking Component will concern references and types. The checking component should be able to verify that a given element is really what is expected. It will also responsible of the casting operation (that is the transformation form one type to another).

The checking component (Check-MEC) interacts closely with components and Managers. It permits to detect and handle all kind of problems and errors at any level: interface, internal or sub-request.

Manageable Application

1) General Architecture

Now that we have analyzed the problem, let's find what we will manage. If we want to manage a system, we want it to be manageable.

Logically, the smaller and the simpler the system, the more manageable it will be. Structuring both the component and the data it handles should ensure increased stability.

We will also need a way of checking the data the component manipulates. This means there must be a certain redundancy in the data.

1a – Components

The first step to complicit management is to break the system into small working units that perform different simple tasks. Each unit typically has its private data and code blocks, that are not visible from outside the component.

Client & Server

Each time two components interact, they will either be a client (i.e. requiring services from another component) or a server (that is, providing a service). They will only be server or client for this relation. A server may in turn be a client for another server.

Each components will in fact always be both, they all offer a service and require some other in order to do their job. The user-interface components are servers for the human users, and all components are client of the Operating System, the processor or the virtual machine.

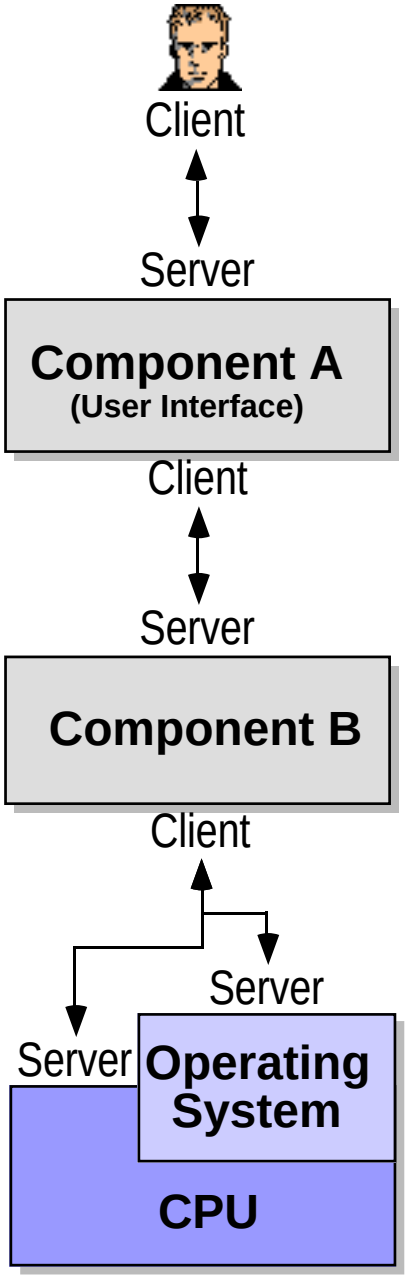

Figure 6 - Client & Server

On Figure 6, Component A is a client of Component B. B is A's server. In turn B is a client of the Operating System and the CPU. Because Component A offers an user-interface, it is also a server for the human user.

Data Encapsulation

One way of being sure that components do not step on each other's feet is to define clearly who has access to what. Each component should clearly have its private storing area. Access to this private data should as much be done through

controlled points. This way, possible sources of corruption will be limited. In system that do not use pointers, private data cannot be corrupted by other components.

Checkpoints & Rollback...

One classical mean of building a stable application is to regularly make checkpoints. That is, to save the state of the system at a given moment. When trouble occurs, the application can always rollback to this stable point. That is, restart form the last checkpoint.

The idea is to segment the execution in small stretches, when a problem occurred in one stretch, the system can fall back to the previous stretch (see Figure 7).

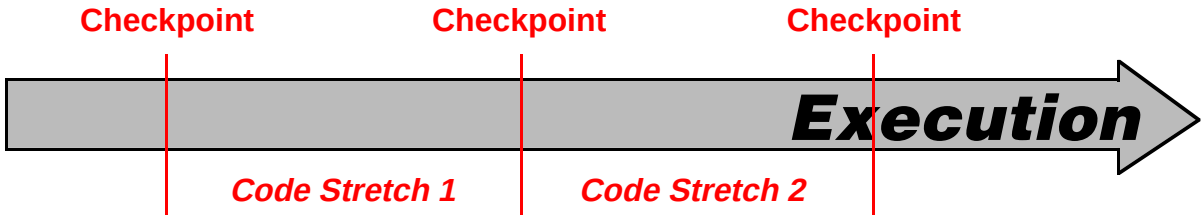

Figure 7 - Linear Code Stretch

With the advent of structured code, where executions passes from one component to another, our stretch is a little bumpy (see Figure 8). The executions constantly jumps in and out of components. The idea here is to decide that at one level of abstraction, each stretch is summarized as one service call. Therefore, the stretch goes from the function call entry-point to the return-point.

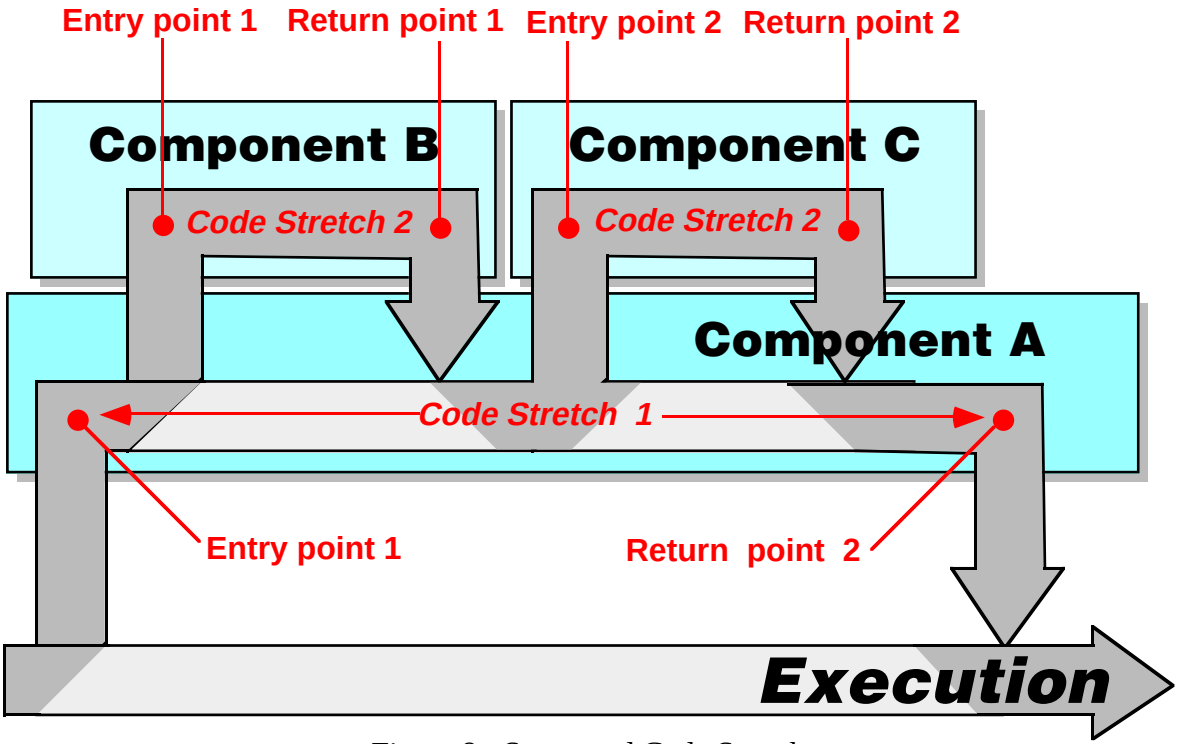

Figure 8 - Structured Code Stretch

So our checkpoint and rollback system can be seen as a service call that can fail. We only need to ensure that when a service fails, it rollbacks to the entry-point. This means that, in case of failure, a component should restore all data to the entry-point state.

Figure 8 shows structured rollback structure. The code execution first enters a service of Component A at entry point 1. The code between entry point 1 and return point 1 is stretch 1. If something goes wrong in between, we will fall back to the entry-point 1.

To offer its service, component A calls component B and C. First, the execution enters component B. Between the moment we enter Component B and the moment we leave it, we are in code stretch 2 (we are still in stretch 1). If something goes wrong, we will fall back to the nearest entry point: entry point 2.

When component B as finished its service, component A calls component C. If something happens between those two service calls (between return point 2 and entry point 3), the execution falls back to entry point 1. If an errors occurs inside component C, the execution falls back to entry point 3.

1b – Data

To manage problems we need a way of detecting them. One typical trouble is corrupted data. To be able to check data, we need to know what this data is supposed to be, that is to know its type. The other thing we need is a way of checking that the data is effectively what it should be.

Components are data too!

Code is also a special kind of data. If some languages like Java do not give access to the code in memory, direct memory access is still common.

In this case code should also be checked by the system for possible corruption. If the programmer can access the code in memory, bugs can also.

Another threat to code structure are viruses that modify the code to include themselves. By checking the code structure, we can protect code elements against viral infection.

Type

Type is foremost information for data integrity checking. How can we tell if something is what it is supposed to be, if we do not know what kind of thing it is supposed to be. It is impossible to decide if a text is correct if we do not know what language it is written in. The more precise the type information, the more accurate our checking will be.

Types in languages can be seen as a label that identifies what's in a container, this way, we won't mistake salt for sugar while cooking.

Units

If you ever filled out a form, you know that there are certain entries that can only contain specific data. You cannot put 90 Kg into the height field. The Kg is a weight unit, and a height is expected. Units are human way of specifying types. Unit information is very important, as they are not uniform, even if degree and radians specify the same thing (an angle), some conversion work must be done between them.

Using unit typed data will help both prevent type mismatch error (using one type instead of another) and unit conversion errors (counting expenses in pounds instead of dollars).

Run-Time Types

Most language can check the type of data while the program is compiled. This is called compile-time type checking. Often we want to transform data from one type

to another. For instance, elements in a list are of a general type, like pointers, and we want to transform them back into a precise type. This transformation is called a **cast**. The compiler cannot check that the data inside the list is really of the type we expect. In many language (like C) the compiler expects that the programmer knows what he does when he performs a cast. In those languages, the type information is lost at execution-time, so the program has no way of checking if the cast makes sense or not.

Runtime checking is an infrastructure that performs type checking while the programs runs. This is possible because the type information is kept for run-time. The cast operation is checked and can fail if the type transformation is impossible. This insure the programs does no absurd things, like trying to send the data form a picture into the sound-card. Run-Time checking exists as an extension to C++ and is included in Java. It is clear that such a feature will be very convenient to check types.

Typed Files and Pipes

So if we used typed data, we will always know what kind of data we manipulate? In fact, no, if most languages support types, there are places where data is not typed: Files and communications links.

Two File Types ?

In fact, many systems know two types of files. Some Unix libraries as well as the FTP protocol distinguishes between text and binary data.

The buffered library has a switch in the read mode that permits to specify if the file must be opened as a text file or a binary file. In practice the switch does nothing.

The switch in the FTP protocol permits to transfer text data between different systems.

Sadly this scheme has not been extended to transform platform dependent graphical or sound data.

When data is saved on disk, the information about its type is not stored with it. Most OS rely on a loose suffix rule to distinguish different data types (`.txt` means text, *usually*), and no checking is done. This means the data can be very strongly typed while in memory, and loosely typed on disk. Files are certainly one of the weak points where type information is lost.

This means there is no actual way of knowing what is inside a file. Most systems rely on the name suffix (in the Amiga OS, it was a prefix) or on some identification key stored at the beginning of the file.

OS with typed Files

Some OS like the Mac OS assign a type information to files. Files can also have resources, that is typed information linked to the file. This means one file can actually contain many version of the data in different forms.

The BeOS is even more advanced, the file system can be seen as a database system. Type, creation and modification dates, but also author and notes are simply entries in the database.

The same thing applies to most communications channels, like pipes. The Java language defines a way of storing typed information into a data stream (file or link). The HTTP (Hyper Text Transfer Protocol) defines a type encoding scheme that transfers typing information along with the actual data. The sad thing is that most of the time, the servers relies on the suffix of the requested file to decide of its type.

Strongly typed files and network links are very important to maintain the coherence of a system. As with most situations implying many components, the actual level of coherence is given by the weakest component. By using strongly

typed files and pipes, the system will not loose the information while storing or transmitting the data.

Tagged Data

We all know that `4-2` does not means the same thing that `2-4`, the order of operands has an important meaning in the subtraction operation. The problem is when we build code routines that take parameters, there often is no canonical order.

Let's assume we are programming a routine that copies the content of a window into another. Let's call this routine `copywindow`(a, b). Which parameter is the source and which is the destination? Do we copy from a into b, or into b from a. As both parameters are windows, we cannot use the type information to check if we are doing the right thing.

In some languages[13] you can use tags instead of the order to tell what parameter does what. We can write `copywindow(src=a, dest=b)` or `copywindow(dest=b, src=a)`. `Src` and `dest` are tags that tell what is what.

Structure

Normally, a component won't manipulate isolated bits of data, but linked information. One way of checking the information integrity can be done by checking if one part of the information matches the other. If our system handle persons, with their height and weight, and contains a person measuring 2 meters and weighting 40 Kg, we know something is wrong. Why? 2 meters is a legal height for human beings, and 40 Kg is legal weight, but both information do not match.

In fact, most data has **invariants**. Invariants are properties that are always true even while data changes. If those invariants are not respected, the data is corrupt. Even if a picture is scaled, mixed or clipped, the number of pixel will always be equal to its height multiplied by its width.

Most data manipulated in computer systems is structured in some way or another. By implementing small verification schemes that can detect absurd or corrupt data, the system can detect trouble more reliably.

Redundancy

Another way to check if things are OK is to be able to compare them to redundant information: a copy. Such practice is current at the hardware level, in storage or communication schemes (parity and CRC are typical examples). By storing redundant data into data structure, it will be possible to detect corrupt or incoherent data.

Hardware Redundancy

Redundancy has been around for year in computer's hardware, insuring data integrity in memory, transmission and disk storage.

Parity, CRC, Checksums and RAID are typical examples of redundancy schemes that are commonly used.

Let's take for instance a personal record. It contains a name, a surname, a birth date and a social security number. In Switzerland, the second number of the social security number is the birth year. The first number is built in a more complex manner with the family name, and the third with the birth date. By storing both the birth date and the social security number, we can check if both dates match. If they do not, the data is corrupt.

13 The Dylan Language permits tagged parameter passing. This scheme is also used to pass parameters to Java applets.

A lower level example: in the case of text strings, we could store the string's length (like strings in Pascal) and a special character at the end (like in C strings). This way, we can check that both end information (length and end marker) correspond.

This redundancy could also be used to speed-up operations. When we want to know the string length, it is already calculated. While parsing strings, we can simply check for the end marker. For each operation involving strings, we can choose the most efficient end-marking system.

Byzantine Errors

Most idea developed here assume that errors, when they occur, will transform syntactically correct data into syntactically incorrect data. The possibility exists that an error transforms the syntactically correct data into semantically incorrect data that still is syntactically correct.

For instance a bogus memory write could transform the surname "Anne" (a existing surname) into "Anna" (which also exists). This kind of errors cannot be detected by methods using the data structure (even a human being could not detect this). Redundancy is not 100% safe from Byzantine errors either, a bug could modify both copies of the data.

We can therefore not completely exclude such errors, but simply stress the fact that they are very improbable and are therefore less likely to happen than visible errors. Given a infinite time, a monkey typing randomly with a word processor could write the complete script of Hamlet, but before that, the word-processor will crash…

Reclaiming

The Java `transient` keyword

The Java language defines a keyword to mark objects that are "reclaimable" and thus can be erased in low memory condition, or during file swapping.

Until now, most ideas included adding data to obtain some redundancy. In low memory resources conditions this won't help at all. In fact, building redundant structures is the worst thing to avoid "out of memory" errors. The fact that most systems now have lots of memory available, and will have even more in the future, will not protect us from "out of memory" errors. Memory usage grows as fast as memory availability.

When memory is low we would like to throw away everything that is not important. This means managed code should try to pack or to store the data in order to use less resources (memory, network bandwidth etc.). Here are some possible strategies:

- Destroy all data that has been loaded from disk or the network and only keep a reference to the representation on disk or in the network (URL).
- Destroy all redundant information.
- Compress data.

Data should therefore be organized with some redundancy, but also in a way that part of it could be sacrificed in a dire need situation.

2) Components Requirements

Thou shallst accept other's mistakes…

There are three basic things a manageable component should do in order to become a MEC (Management-Enhanced Component).

- It should communicate with its Manager, this will be done through a management interface.
- It should do its job in the best possible way, adapting the service to the situation.
- It should be fault tolerant and behave in coherent way in case of failure. The failure can happen inside the component , or in a peer component.

2a – Management Interface

The component communicates with its Manager using its management interface. This means the component should know a Manager. The Manager will itself have a Manager interface to communicate to. With this standard structure (see Figure 9), system design and maintenance will be facilitated.

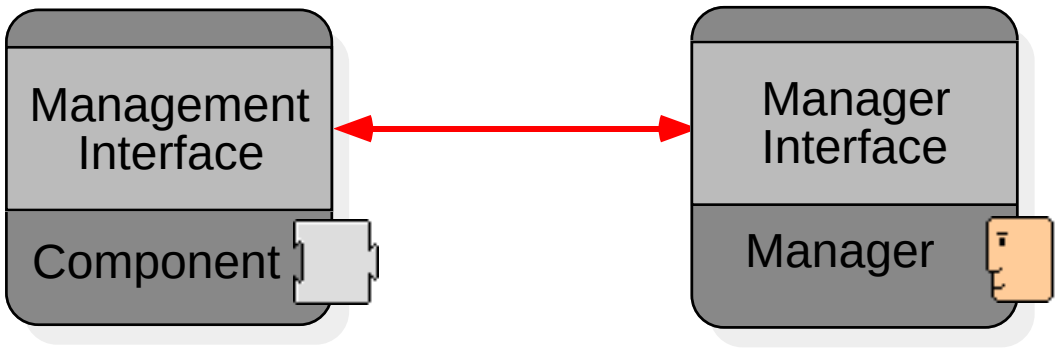

Figure 9 - Standard Interfaces

The management interface should be as lightweight as possible in order to minimize the management overhead. There is no point in building a managed system that is unable to do anything. Emulating bureaucracy is certainly not the point of this work. Ideally the management interface could be removed or added in a existing component without affecting it.

Management Information

The first step to being managed is telling the Manager what's going on. This means notifying the Manager in case of trouble but also giving the Manager information about what is going on. An uninformed Manager is useless. A component should always be able to answer some questions from the Manager. Here is a list of elements the component should provide to the Manager.

Version

We would like our system to evolve. This means new versions of the component may appear afterwards. The Manager must be able to distinguish different versions of a MEC. Different versions of a given MEC may not encounter some problem, or be unable to fulfill some requests.

Availability

Availability is the measure of the component's capacity to handle a request for service. Availability is affected by the current load of the component. For instance a file system component may only have a limited number of open files.

Availability can also be affected by the availability of sub-components. A compression component could well be unavailable because the file system component it uses is unavailable.

Management Level

Not all MECs will be managed to the same point, nor will they offer the same safety level. A MEC should always offer precise information about its management and recovery levels.

Recoveries

Specific components could well know about specific recovery schemes. If a MEC knows about some recoveries it should say so to the Manager.

State

Some components have to retain some information between services, that is they retain a state. Others do not and are therefore stateless. This information is very important for the Manager because it affects the way the component works. Each component should tell its Manager if it is stateless or not (see *State Preservation* page 29).

2b – Service in all Circumstances...

The component should always work, not only in ideal situations. This means the component has to be able to offer its service in problematic situations. The component should be able to handle partially damaged data, the component should also be able to work in a downgraded mode in case of resource shortage.

Working with Erroneous Data

The `NaO`

One interesting feature of mathematical systems is the existence of the `NaN` (Not a Number), a special symbol representing the result of illegal or failed operations (divide by zero, overflow). In many systems the `null` pointer or reference plays the same role except in one way: all mathematical operator accept the `NaN` object and do coherent computing with it (`NaN +2 = NaN`), most systems and API crash if given a `null` pointer…

Ideally a component should accept to work with a symbol representing a illegal object/state. I call this special object `NaO` (Not an Object). This `NaO` symbol represents an object in an indeterminate state.

So does this `NaO` thing represent a completely unknown object we know nothing about? Wrong, we have one important information: the type. This will be very important, because we cannot access information about the object itself, we can access the type's information.

All operations implying NaOs must be coherent. For instance a NaO of a Picture can be rotated and will still be a NaO of a Picture. Mixing a valid image with the NaO gives a NaO. Most information about a NaO object will themselves be NaO, for instance it may be impossible do determine the number of color used by a NaO, the answer will therefore also be a NaO of a color array.

Working with a NaO

Let's imagine we are working with a graphic meta-file format[14]. This graphic format can contain either a line, a polygon, a Bezier curve, a text element, a bitmap or a collection of these objects (see Figure 10). Let's imagine we have loaded a rather complex file, containing numerous graphical elements, and one element, a bitmap, is somehow garbled.

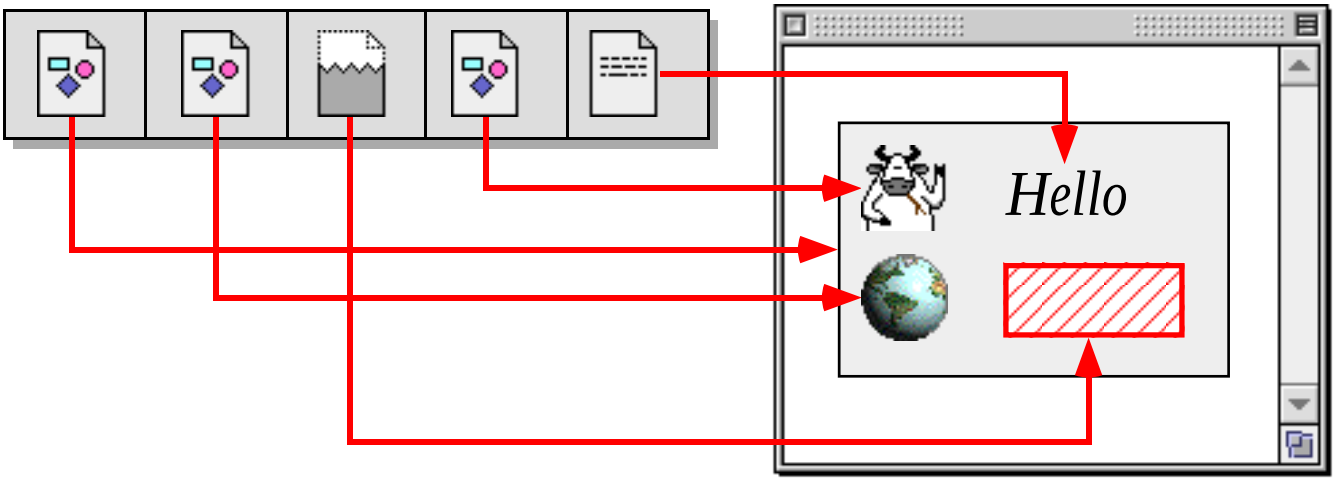

Figure 10 - Meta File Example with Corrupt Data Element.

Perhaps the file was corrupted during the data transfer, or the compression scheme is not supported by the application. Anyhow, the application knows this data is a bitmap, but a unusable one. Instead of complaining, the loading/transfer component should insert a NaO object in place of the corrupt bitmap.

What happens now? We have a collection of graphical objects, one of which is corrupt. If asked to display the NaO, the component should do nothing, perhaps write a warning message at the bottom of the display. If the component receives an instruction to rotate the collection, it will rotate each element, including the NaO (a rotated NaO is still a NaO). If asked to compute the area of the collection, the component will answer NaO (it cannot calculate the area of a unknown object).

Now let's imagine a program wants to extract only specific information from the collection, like all lines, this can be done without having to worry about the NaO because it is a bitmap (if it were a sub-collection, we would have to). We can therefore perform a service with corrupt data, as long as the data isn't directly needed for the service.

[14] Meta-files are literally files that regroup other files, typical examples are the CGM (Compugraphic Graphic Metafile) and WMF (Windows Meta-File).

Downgraded Service

In many cases a bad service is better than no service at all. This is particularly true in the case of human interface systems (display, sound, printing) where having a low resolution information is much better than no information at all.

In the interlaced GIF file format (GIF89a), the image's lines are not stored in a sequential order, but low resolution first. This way, the image can be show in a low resolution mode while loading. The TIFF file format can contain a preview image at the beginning of the file. With such storing techniques, if the transfer is only done partially, or if memory or other resources (like decompressions systems) are insufficient, it is possible to work with a low resolution version.

If a rendering system cannot do shading, it should draw flat colors, if it cannot represent colors, it should work in gray scale, if this is not supported either, it should fall back to dithered black & white, or even a vector display, but *the show must go on…*

Every advanced service should be able to use a more primitive service to fall back on in case of trouble. As an analogy, services should try to perform as much work as possible. If asked to read a file that is corrupted, a file service should return NAOs (see above) for the corrupted zone and work for the rest. In both cases, the service should report that only a partial or minimal service was performed.

2c – Fault Tolerance

In a real situation any component can and will fail. This means any component must be able to deal with a crash. The crash can be located inside the component or happen in a peer component. A MEC must be able to accept those crashes.

Accepting Errors from other Components

One main problem with many systems is that they work on a very optimistic basis: they admit that all the services that they request from other components will work. If they fail, the programs fails and stops more or less gracefully. With the advent of complex (and buggy) systems, and unreliable resources (typically networks) such approach is certainly problematic.

As most components are built this way, a simple error can cause a sequence of failures, each component failing because its sub-components failed because their own sub-component failed. In the end, a small problem (such as a missing file) can stop a very complex application.

This means that even a perfect code (if such code existed) could still be affected by errors because it depends on others elements (OS, API , Servers). Dealing with its own possible errors is only half of the problem, all other components the component interacts with could fail.

On the other side, internally treating all possible errors of all sub-services called leads to a *"Trust no One"* situation. What is the point of a service if it cannot be trusted? Like in the human world, there is no point in delegating work if it takes more time to check and correct it than it takes to do it yourself. By doing too much correction and checking, programs are less portable and less efficient.

A managed component should simply accept the failure of other components and dispatch them to the Manager. This way the component won't be loaded with recovery code but still remain stable in case of trouble.

Accepting Errors from itself

Errors do not only occur in other components. Each component must be prepared to handle errors occurring inside itself. For this reason the component must try to save its state. In a component crash, the component must also save all important data.

State Preservation

A stateless component is one that does not need to remember things from one service to another. A component offering an image compression service does not need to know what it did previously to compress an image, all it needs is the image. Such a service is basically stateless.

On the other hand, a service offering a windowing service must remember what windows it opened and where they are on the screen. Such a service has an internal state. In this case a list of windows with their positions.

Stateless Example: NFS

Sun's Network File Service (NFS) is a typical example of a stateless service. NFS is a network file system, it permits to mount remote disk through the network.

One important aspect of such a service is robustness. To avoid state conservation problems, the NFS system is stateless: no information is retained inside the server between transactions.

NFS does not keep track of open and closed files, each time a read or write operation is required, the file is opened and then closed. This way, if a client crashes, no corruption occurs.

What happens when the windowing component crashes and is unable to restore its information? If, when restarted, it is asked to draw a picture inside window 4, what will it do, as it does not remember where window 4 is?

The best way to avoid this problem is to encapsulate this information inside the object the client manipulates and to store a copy inside the server. This way, if the server crashes, when the client asks to draw inside the window at a given position, the server will be able to know that there is a window at this place.

Graceful Death

If something can go wrong, it will go wrong. So a component should be built to do some "last ditch" measures in case of trouble. Typically a "dying" component should clean up data structures and release resources it locked. Critical data should be saved.

This means that a starting component should check for other component's "last words" and try to resume the service at the point where the system crashed. This feature is already found in database systems and some applications[15] but should be generalized and brought to the component level.

[15] Word for Windows is able to rebuild a unsaved document by inspecting temporary files.

MEC Structure

1) Introduction

Now that we see what we want, here is the abstract structure of the MEC (Management-Enhanced Components) System. This system is a management pattern that could be built on or in any existing service. In this part, the general structure of a MEC system will be described.

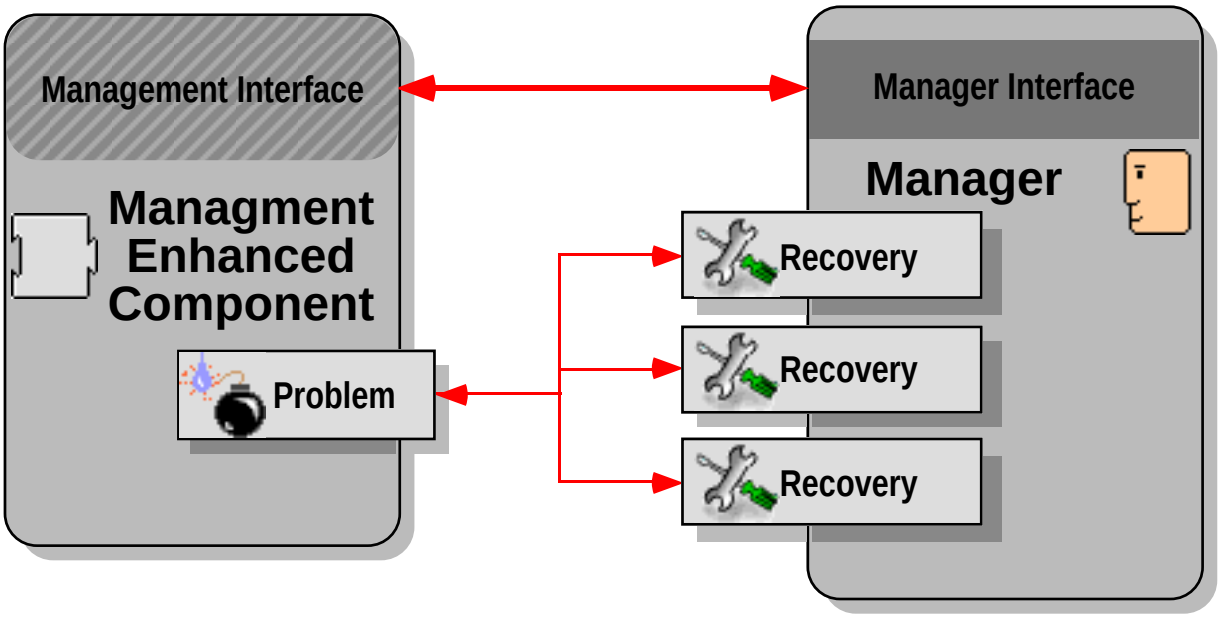

Figure 11 - General MEC System Structure

The system is built around five elements: the Safe-Object (MESO), the Management Enhanced Component (MEC), the Manager, the Recovery and the Problem.

The Safe-Object represents data whose type is known and that can be checked. The MEC is the building block of the system, each application is built by assembling such components. Each MEC communicates with the Manager through a Management Interface.

Figure 11 shows the general MEC structure: the Manager handles Problems. To correct problematic situations, it uses Recoveries. Recoveries are objects that contain corrective code. The Problem object represents a problem, each problem is identified by different information (localization, gravity, type).

2) Management-Enhanced Safe Object

2a – Concept

A Management-Enhanced Safe Object (MESO) is the basic data block of a MEC system. Safe objects should be used as much as possible in the whole application, even if no management is implemented. Using safe objects allows to detect object corruption and illegal states.

A safe object should be able to perform a self-check. The object can use redundant information or heuristics about the structure of the data to perform this check.

If the object uses many resources, it should be able to discard all redundant data to work with less resources.

2b – Safe Object Services

A typical MESO should provide the following information:

- give its type
- give its status (i.e. corrupted or not).

3) Management-Enhanced Component

3a – Concept

A MEC (Management-Enhanced Component) is a component that handles complicit management requests. A MEC is managed. That is, it has an attached Manager that tries to recover and prevent problems. MECs use the service of other MECs, systems calls and the host system's API. As the MEC is also an object, it should be based on Safe Objects. This insures that the integrity of a component can be checked.

3b – ME Levels

Ideally, all components would be completely managed, but in reality it won't be the case. Mixed design will certainly occur, so the Manager may need to know how far a MEC is managed. One idea is to define Management Enhancement Levels, that is, a measure to what amount the component is *Management-Enhanced*.

ME Level 0

The MEC doesn't provide any management infrastructure. This is the default value for non MEC components.

ME Level 1

The MEC has an attached Manager to which it dispatches occurring problems. When such a problem cannot be recovered it is transmitted to the client.

ME Level 2

In addition to level 1, the MEC only manipulates safe data. The MEC can and does check the integrity of all the data it manipulates. This data is implemented as safe objects.

ME Level 3+

Level 3 and following are the same as level 2. ME-Levels greater that 2 mean all sub-component the MEC uses are also managed MECs. The actual ME-level is the worst ME-level of all components the MEC interacts with plus two.

For instance, if a MEC is fully managed (level 2) but only uses the service of a MEC with ME level 1, the MEC's actual level is 3…

3c – MEC Services

A typical MEC should do the following:

- Accept and dispatch an internal failures
- Check the data and service requests received from other components for corrupt data and illegal requests.

Implementation Ideas

Those services, called MEUF (Management-Enhanced Utilities & Functions) should be typically implemented in a basic MEC object, so new MECs could inherit most of those services (typically the dispatching functions).

- Accept orders from its Manager (self-test, reset).
- Try to offer its services even in problematic situations (insufficient resources, corrupt data, NaO parameters).

3d – What is a MEC ?

The question is, what element should be programmed as MECs?

Libraries

In classical procedural programming, related function are grouped into libraries. Libraries have different names in different languages: units in Pascal, packages in Ada etc. Those libraries can typically be implemented as one or many MEC depending on their size and the data they share.

Stateless Libraries

Libraries can often be stateless and therefore be easy to implement as MECs. A Library is stateless if it uses no static data.

Large / Complex Objects

Ideally all objects in a MEC system should be safe-objects. Large and complex objects would need more advanced management controls. The logical way to do this is to implement them as MECs.

In Object Oriented design, every object can have attached methods. That is each object can be a small server offering services on its inner data. Would it make sense to implement all objects as MECs. The answer is no: only large and complex objects contain a sufficient amount of data and structure to make management possible.

Software Components

Software components are, of course, the best candidate to become MECs. In most case adding the management interface to the software component would be sufficient.

If the component is large and offers a services that requires a state (see State Preservation page 29) inside the server component, one good idea will be to build two MECs. One containing all the services that don't require the state, and one containing the state. This way, the first component will be more easier to manage because it is stateless. The complex management code will be concentrated on the second MEC.

4) Manager

4a – Concept

A Manager is an entity that dispatches problems it gets from its client MEC. A Manager can be shared by many MECs at different levels. The Managers stores and handles recovery schemes. The Manager also logs and controls problems.

What a Manager should not do.

In the MEC design, here are the things a Manager should absolutely not do:

– Initiate management and corrective action without being requested to do so by the MEC.

– Make assumption on the way a service is offered by a MEC.

4b – Basic Manager

Basically the Manager should do the following tasks:

- Maintain a list of its client MECs
- Maintain a list of Recovery objects.
- Try to match incoming problems with a recovery object and apply it.

The last point is the one that could require the most work to be efficient. Searching through the recovery list to find the best recovery could imply advanced searching strategies. Figure 12 shows the general MEC structure.

Figure 13 show a typical Manager state transition. In the beginning, the MEC works, the Manager is inactive.

A problem occurs, and the MEC tries to dispatch it. To do this it contacts its Manager. If no Manager can be found, the MEC fails. If a Manager is found, the Problem is transmitted to the Manager.

The Manager is now active. The Managers searches for a recovery. It does so by asking each recovery to estimate itself in relation to the problem.

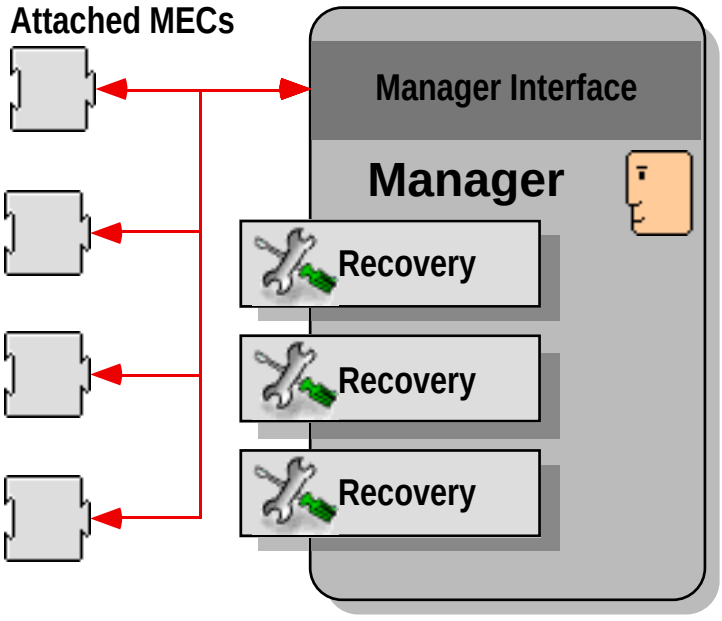

Figure 12 - Manager Structure

The Manager then chooses one of those recoveries. The Manager then tries to apply this recovery. Either it works, or it does not. If the recovery works, the problem is solved, the Managers logs the problem, the MEC works again and the Manager is inactive again.

If the recovery fails, the Manager searches for another recovery, and will try it. If no other recovery can be found, the Manager fails, the problem is logged and the MEC fails: the problem could not be corrected.

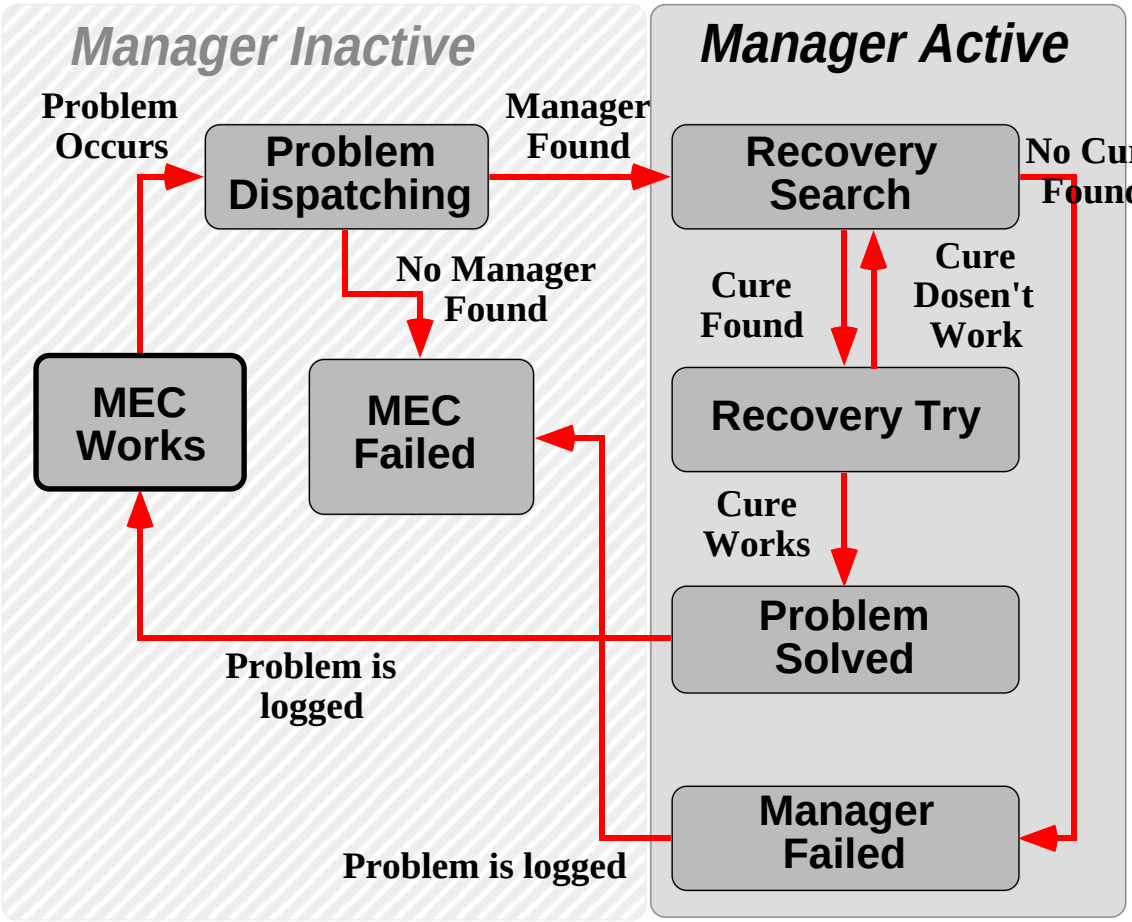

Figure 13 - Minimal Manager states

4c – Hierarchical Managers

One problem with this design is that the Manager can get quite fat with recoveries. Filling the memory with recoveries is not the point of management.

Loading the recoveries as needed could also lead to a problem. In case of a low memory situation, the Manager might be unable to load the recovery to reclaim memory.

The idea to circumvent this problem is to build a Manager hierarchy. Only the primary Manager is loaded. This primary Manager works like a generalist doctor.

This generalist Manager can dispatch classical problems (like the memory shortage). If more specialized expertise is required a more specialized Manager is loaded.

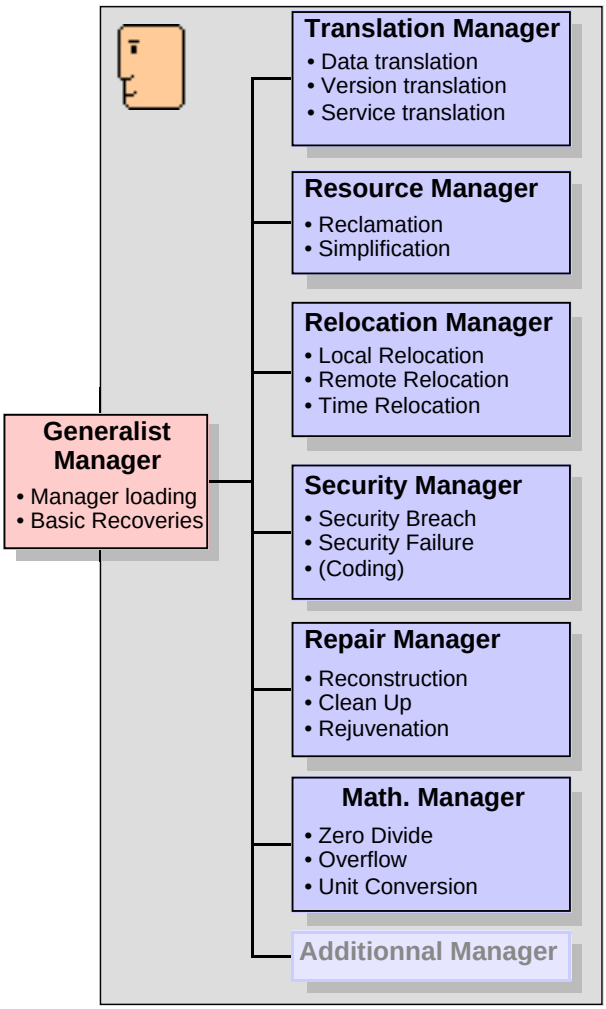

Figure 14 - Hierarchical Managers

Figure 14 shows the Hierarchical Manager structure: the generalist Manager is built like a basic Manager, but also keeps references to the specialized Managers, if it cannot correct a given problem, it loads the appropriate Manager and transmits the problem to the newly loaded Manager.

Each specialist Manager has the same Manager interface than the generalist Manager. To the outside world the Generalist Manager appears as a very smart Manager. The specialized Managers are invisible. Because all specialized Managers use the same management interface new specialized Managers can be added at nay time. If necessary, a specialist Manager can even replace the generalist Manager.

4d – Inter-Manager Communication

Our design allows many Managers to coexist. But in order to cooperate, they must communicate. Communication is very important in situations were a single source causes many problems.

Let's imagine a system with many components. They share a cache to boost network performance. The cache gets somehow corrupted. All components will soon detect this. They all will contact their Manager to correct this problem.

It is important that all implied Managers synchronize themselves so that only one of them does the recovery. In this case, it makes sense that the first Managers rebuilds the cache system.

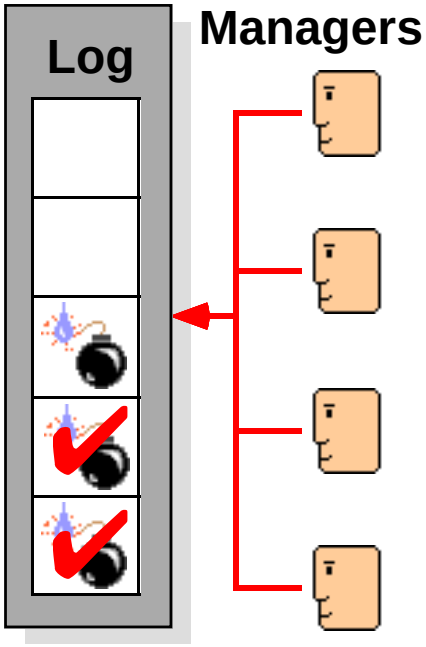

Figure 15 - Log

The central point of communication is the log (see Figure 15). This log contains a record of all problems that occurred and what recoveries were applied to them. This log object also serves as a synchronization point. Before correcting a problem that could have been already corrected by other Managers, the Manager will check the log for indication about related problems. The Manager will engage in a recovery process only if no other Manager is working on the issue.

4e – Meta-Management

Who watches the watchmen?

Introduction

Who manages the Managers? Managers and Recoveries, like all everything else, can fail. So they must be managed. By whom? it could be a super-Manager (the hierarchical idea) or a peer Manager. Managers could be linked by pairs, each managing the other.

To be managed, Managers should be built the same way as all components: accept management orders, perform self-test, etc. In short, they should be built as MECs. In our case the Manager was the first MEC built.

Self-Management

A simple idea for meta-management would be self-management. Because the Manager is built like a MEC, it can handle itself like a client and handle its own problems.

If this approach is simple, it is also very risky: if the Manager has a problem, it is probably unable to offer its management service, and therefore unable to handle its problem. An exterior Manager would be needed.

Hierarchical Meta-Management

One idea to handle the Meta-Management is to build a super-Manager that handles many Managers. This Manager component could in turn be managed by a super Manager component (see Figure 16).

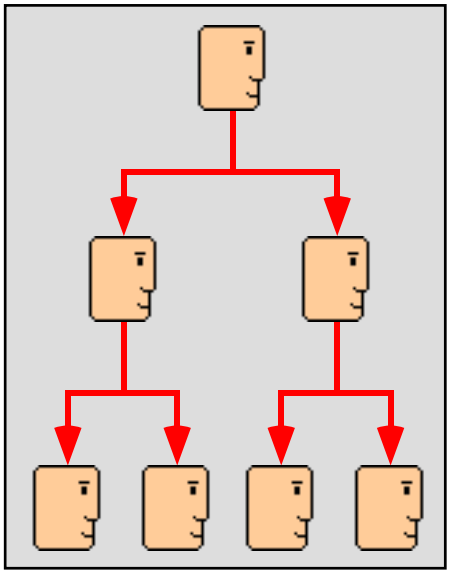

Figure 16 - Hierarchical Meta-Management

Like with any hierarchy, the weak point is at the top. At one point, there will be a supreme Manager that will not be managed.

This supreme-Manager could be self managed, but would still remain the weak point of the system.

Peer to Peer Meta-Management

Another approach to Meta-Management is the peer to peer system. In this architecture, Manager are grouped by pairs. Each member of a pair is responsible of dispatching the problems occurring in the other Manager (see Figure 17).

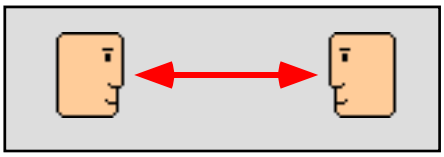

Figure 17 - Peer to Peer Meta-Management

This way, if a Manager fails, the peer Manager who is unaffected can recover the crashed Manager. Of course both Manager should be as independent as possible, so that an error affecting one will not affect the other.

5) Check MEC

The MEC system implies lots of checking. This means advanced checking services are necessary.

Type Checking

One checking that is done a lot in dynamic system is runtime type checking. This could typically be done by the Check-MEC.

This means the Check-MEC should also be responsible of dynamic cast operations.

One simple approach to checking services is to unite them in a special MEC. The checking calls can be linked to the Manager or to the MECs, but the checking would be performed in a special component. This component will not be unique. Specialized component will need specialized check-MECs.

Because this component does checking for other components, it will report problems in a slightly different way.

One Check-MEC ?

Having only one Check-MEC is unrealistic. This would imply that managed programs will only manipulate a limited set of data with a limited set of checking methods.

There will of course be many Check-MECs, but they will share the basic checking services. Most MECs will manipulate objects references, integer ranges and filenames…

Technically, all checking violations will happen inside the check-MEC, but in fact the problem is inside the client component. So the check-MEC will dispatch its violations as having happened inside the client component.

What is the point of having a central checking point? This means that all checking code can be optimized for a minimal overhead (remember we do not want to have a paranoid program that only does checking). In case of check violation, the Check-MEC can try to recover the problem directly without having to trouble the MEC.

Let's assume we have a small component that draws arcs, one of its parameter are the starting and ending angle, those angles are supposed to be in radians. The actual values received are 90 and 180, the component calls the Check-MEC to verify that both are valid radians angles (we assume the radians angles are all between -2π and 2π). They are not.

The Check-MEC then calls its Manager to dispatch this trouble. The Manager looks up in its recoveries and finds one that can correct the problem, a small piece of code that does degree to radian conversion. The problem is corrected and a warning is issued (clearly the client code has the wrong units).

6) Recovery

6a – Concept

The recovery is a small piece of knowledge. A recovery typically contains instructions and data necessary for correcting a given problem, or a specific class of problems. Recoveries can be seen as tools for the Manager. The Managers selects a recovery for each problem and uses this recovery to correct the problem.

The recoveries do the actual job of recovering the problem. Recoveries do no checking or deciding. The Managers decides what recovery will actually be used.

6b – Structure

Recoveries must be simple in order to be easily handled by the Manager. All recoveries share a very small and simple interface. A recovery can only do two things.

- Estimate its capacity to handle a given problem.
- Apply itself to the problem and return the effectiveness of the recovery.

This can be seen as the two things one can do with a screwdriver. One can look if the screwdriver fits the screw, and one can try to turn the screwdriver.

6c – Recovery Estimate

The typical estimate could be a simple value, a more flexible estimate could be built along the following axes:

- **Cost**, i.e. what amount of resources (time, memory, sockets etc.) the recovery will consume.
- **Reliability**, i.e. what are the chances a given recovery will actually work.
- **Quality**, i.e. what is the quality the recovery will yield.
  Will it give a degraded version of the problematic service, or a perfect emulation?

What criterion is the most important will depend of the situation. In a real time situation, cost will be the most important variable (the Manager may chose to try many cheap recoveries with a low reliability instead of one big recovery). In a user-interaction situation, quality could be very important. An unattended server situation will probably require a maximum reliability.

7) Problem

7a – Concept

The problem is an object defining a problem situation. This object contains all relevant information about the problem: context, gravity, localization, type, class etc.

The problem object is passive. It is simply a container that is transmitted along the management system.

There are two important things about problems, the first is the way they are dispatched, the second how exactly the problem is defined.

Problems & Recoveries

Recoveries are designed to correct problems. It would seem logical to class problems accordingly to the way we could correct them. This is a dangerous assumption. Even if it is true that often there will be a recovery for each problem class, it is not a rule. A recovery can work on many problems. Retrying a failed operation might work on many network problems, but not on a corrupt file problem.

The other flaw of this approach is that even if recoveries and problems are often related, they don't appear at the same moment and the same point of the design. Most of the time, the recoveries appear quite a time after the problem…

Because problems are, by nature, unpredictable, we must be very cautious in making assumptions about them.

7b – Problem Context

Each problem has a context. The context is the state of the system when the problem occurred. To be effective, a recovery must have access to this context. Typically in a service the context includes:

- Service request parameters.
- Sub-services return codes and exceptions.
- Global variables and other global information.

This means that each problem must be specially tailored to contain those information.

7c – Dispatching Procedure

There are two approaches to problem dispatching, the optimistic, and the pessimistic. The first assume the problem will be corrected and thus is built with return in mind. It is typically built on subroutines calls. The pessimistic approach assume there is no coming back from the problem, and thus implements the error dispatching function with a `goto` or an exception.

Somehow this isn't very satisfying, and a intermediary solution would be welcome, a solution where we could chose to come back. Some systems implement resumable exceptions; this would be very welcome to implement the MEC system (see Java Wishing List page 65).

7d – Problem Classification

Introduction

One idea was to define a problem according to different criteria (see Figure 18), three axis and one object hierarchy. All three axes join at the **No Error** point.

3 Dimensions: Gravity - Localisation - Type

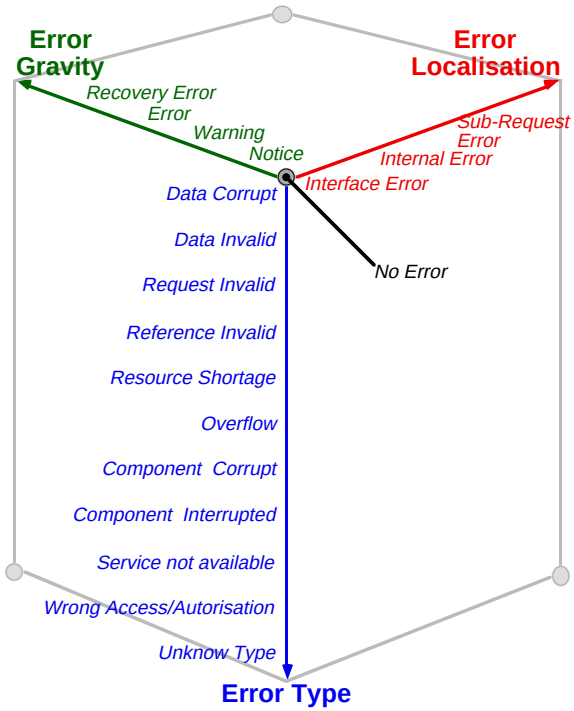

Figure 18 - Error Space

Problem Gravity

This axis defines the severity of the problem. Basically it fits in one of the following categories (listed with increasing gravity).

No Error	All is well…
Notice	a problem occurred but was perfectly corrected.
Warning	a problem occurred, was corrected but should not have occurred, it could hide another greater problem. Execution has resumed but an action should be taken to correct the cause of the problem
Error	a problem occurred and was not yet corrected.
Uncorrected Error	a problem occurred and could not be corrected
Recovery Error	a problem occurred and another error occurred during the correction process. The original error is lost…
Critical Error	A error occurred. The error is sufficiently severe to endanger the whole MEC system. No recovery should be attempted, all components should try to save their data…

Of course, Recovery Errors and Critical Errors should be avoided, but this case cannot be excluded.

Problem Localization

This axis defines the localization of the problem, i.e. the place where the problem occurred. Error localization is very important to indicate where associated trouble could be and which component should take corrective action.

This localization information is relative to a component. The localization information is very important, because it can serve as an indication of the amount of possible damage. In a system without pointers, the damage is limited to the name-space of the crashed component. A Component can only harm data it has access to.

Sub Request Problem	A problem has occurred while the component requested a service form another component. There is no problem in the component itself and the trouble should be contained in the called component.
Internal Problem	The component has reached an illegal state or crashed. The problem should be contained inside the component.
Interface Problem	The component has received a malformed request for a service, or one that is illegal given the context. The component should reject all modification associated by this request and revert to a legal state and therefore not be affected by the problem.
No Problem	There is no Problem.

Problem Type

This axis defines the error type. It is mainly a list of abstract errors. A precise recovery scheme will probably not use this information, it will try to find the precise nature of the problem in the problem hierarchy. A high-level recovery scheme could use this data to decide if it can be applied (a retry on a Resource Shortage makes sense, it does *not* on a Data Corrupt event).

Data Corrupt	An element of data is corrupt
Resource Shortage	Shortage of needed resource. Element of this resource could be recovered from elsewhere
Data Invalid	Data is of the wrong type.
Service not available	Component needed to access another component for a request. Call failed or component is not reachable.
Wrong Access/Authorization	A component needed to access a resource or a component, but has not the right authorization to do so.
Overflow	Component has reached its maximum in a way or another. This maximum cannot be raised.
Reference Invalid	Reference to Data (pointer, handle or filename) is invalid. The data cannot be reached.
Invalid Request	Component was asked to do something illegal, impossible or absurd.
Component Corrupt	The component reached an illegal state. Perhaps its internal structures are corrupt.
Component Interrupted	The component was interrupted or stopped in a way or another, perhaps because it was blocked or waiting on a impossible condition.
Unknown Type	A problem occurred, and the system does not know what kind of problem it is. *This should never happen.*

Problem Class

This information defines the exact class of the problem. This hierarchy reflects the organization of the errors by subject. It will typically follow the exception structure of the host language/framework/operating system.

Problem Class Classification

One tempting idea would be to classify problems by the way they could be recovered. This is a dangerous idea because it assumes the following things:

- That a given problem can only have one class of recoveries.
- That all recoveries for a problem are known when the problem is defined.

This of course not the case. For this reason is wiser to follow the native problem structure.

Because the Problem objects serves as a container for the problem's context, this hierarchy should be built with care (if it does not yet exist). Related problems should share context information (file problems should share a file pointer). This way recovery code could work on a generic problem class…

8) Interaction

In this part, we will explain how those elements link together and how they work.

8a – Interaction Diagram

Here is a interaction diagram showing a typical interaction between MEC, Problem, Manager and Recovery Elements.

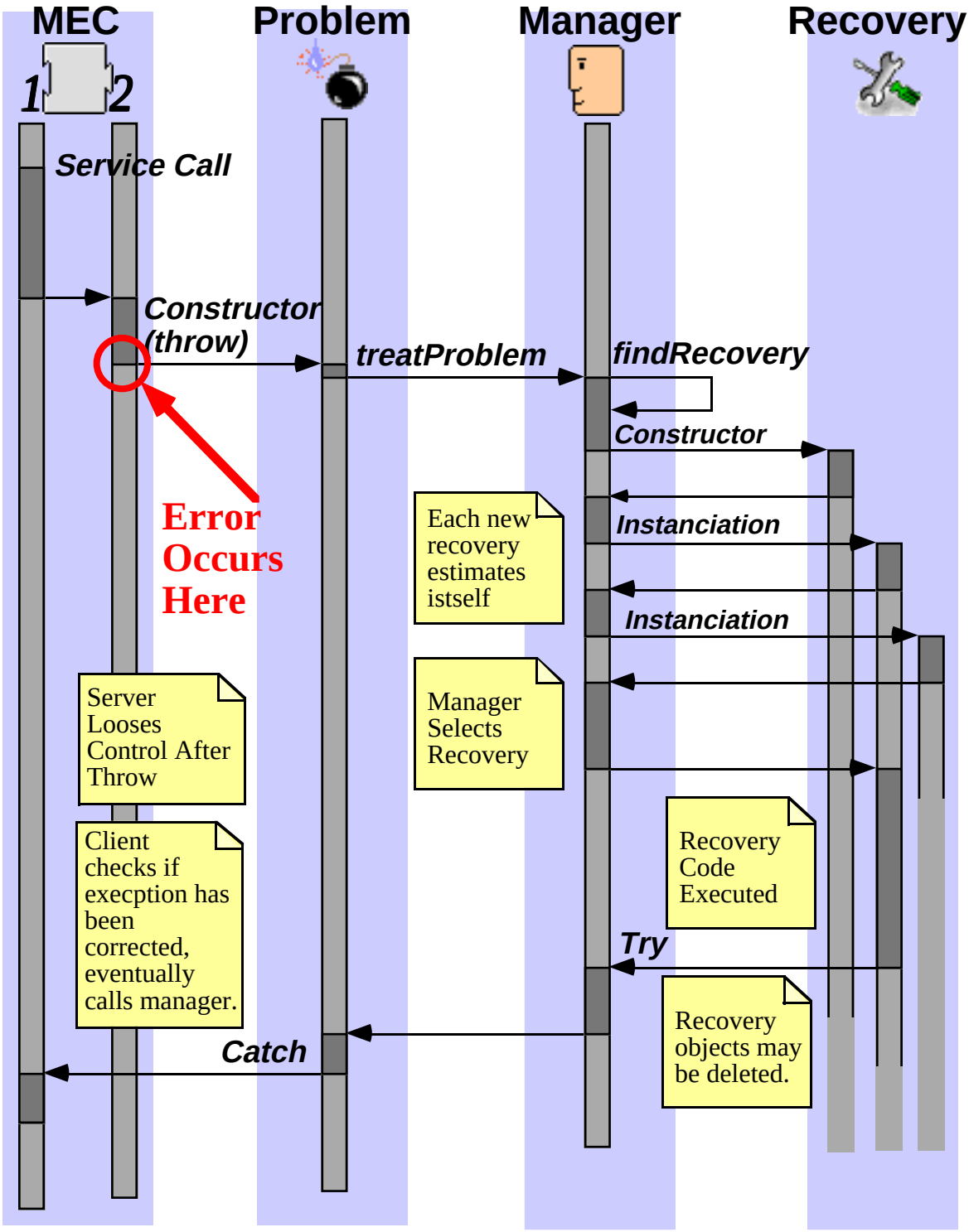

Figure 19 - Interaction between MEC, Problem, Manager and Recovery

Figure 19 is an example of the interaction between MEC, Problem Manager and Recoveries.

MEC 1 calls MEC 2. Something goes wrong: an exception occurs in MEC 2. The exception is thrown. The exception constructor tries to dispatch the problem as it is built. To do this, it calls the Manager. The Manager tries to find an appropriate recovery. In this example, the Manager does not store the recoveries, but builds them as needed. The Manager therefore builds its recovery collection. Each recovery is asked to estimate itself according to the problem.

The Manager then selects a recovery and applies it to the problem. If the recovery code works, the problem is solved, and the Manager returns from the dispatching routine. The problem constructor returns and the problem type is changed to "notice" or "warning". The client then catches the exception. The client needs to check the type of the exception, if it is a "notice" or a "warning", the problem has been solved and the execution can go on normally.

What happens if the recovery code does not work depends on the Manager and the recovery. If the recovery should have worked and failed because of a sub-service failure, it can in its turn call its own Manager to dispatch this sub-problem. If the recovery does not work, the Manager can try to find a new recovery.

8b – Error Transmission

Figure 20 shows how a problem is transmitted between many MECs, and how its type changes in the process. In this diagram, we do not take the Managers into account. The diagram only shows how errors are transmitted along the caller chain.

MEC A calls component B for a service. MEC B calls component C and passes it some data. Component C checks the data and detects that this data is corrupt. It refuses the service request and returns a "corrupt data interface error".

Because of this error, component B checks the data and discovers that the data is really corrupt. This is an illegal state for component B, who gives up and returns a "corrupt data internal error".

MEC A cannot continue because component B failed, it will also return, but the error is this time a "sub-request failure": component B failed to offer its service.

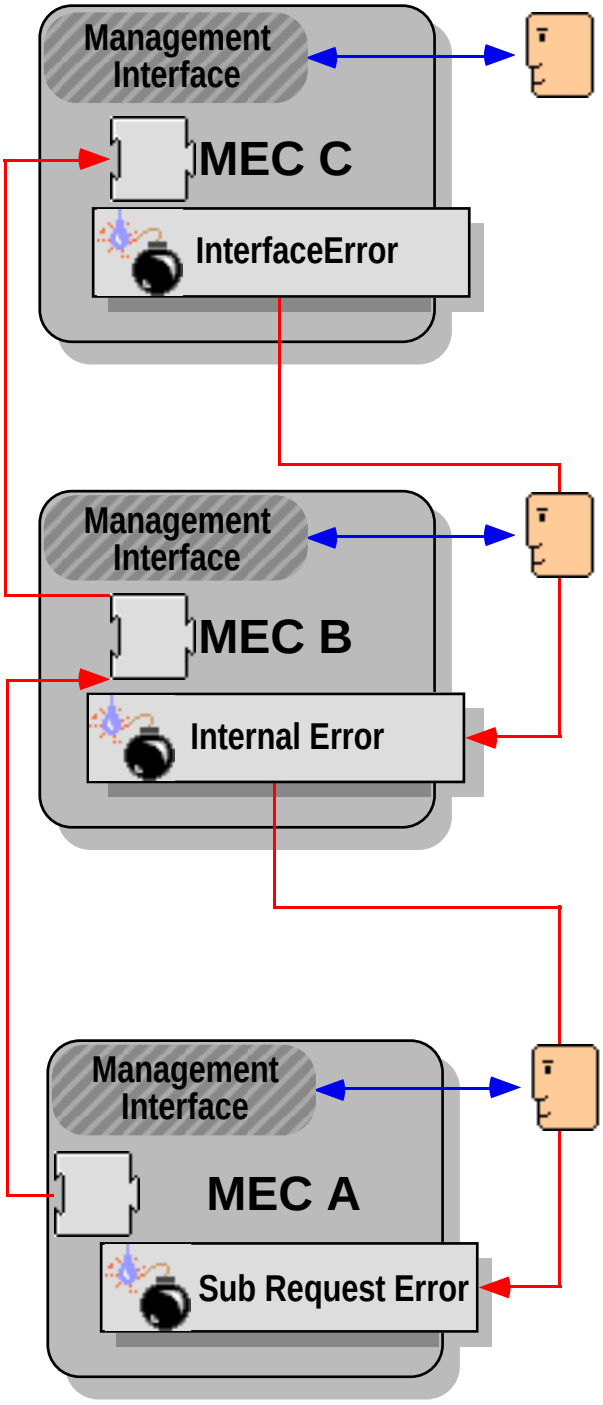

Figure 20 - Component Interaction & Error Transmission

MEC Implementation

The MEC system was implemented twice: a C++ version was done first, it served to try out and refine the MEC concepts. It was built with no concern about integrating it in an existing system. All elements were built in the most straightforward way.

The Java MEC was built on a more flexible basis: to become a MEC system, you only have to implement certain interfaces. So existing code can be transformed into MEC code more easily. Most structures are also lighter and simpler.

1) Language Requisite

In theory, the MEC scheme could be implemented in any language. In practice building such a system in Fortran 66 is probably a bad idea. Here are some requisite that, in my sense, are necessary to build a viable MEC implementation.

1a – Types & Structures

Most of today's programming languages support types and complex data structures. Both types and structure are necessary to detect and dispatch problems.

1b – Object Oriented

Object Oriented code is especially useful for objects with attached code like the recovery object. Objects hierarchies and inheritance are also necessary to build a safe object from whom all objects can inherit safety and checking methods.

And the Winner is...

This will come as no surprise, building a MEC system requires an advanced language.

C++

Certain version have Exceptions and Run-Time Type Identification (RTTI), with those extensions, C++ is a good choice.

Pascal

As for C++, some version of Pascal are Object Oriented and have an Exception mechanism.

Ada

With its strong emphasis on stability, its exception handling and strong types, Ada is a good candidate, the newest version also features Objects.

Dylan

This language offers all required facilities: dynamic types, object oriented, recoverable exceptions, etc. Alas, Dylan is rather complicated and very rarely used.

Java

Probably the best choice, the absence of pointers diminishes memory access problems. The runtime type checking is automatic.

1c – Runtime Type Information

In an Object Oriented environment, this can be easily obtained with a `Get_Object_Type` methods, that returns an object identificator. The runtime type checking must then be done explicitly. Of course if runtime type identification is available in the language, the code will be clearer.

1d – Exceptions

An explicit channel to carry error information is necessary to properly dispatch problems as they arise. The exception mechanism is very practical to build a MEC system, as the Management function can be embedded in the exception catching mechanism. One feature that could be very useful are resumable exceptions, that is, exceptions you can come back from.

2) C++ MEC

2a – Introduction

To determine if the whole MEC scheme could be implemented in a functional way, a first simple C++ prototype was built.

The whole MEC architecture (Manager, MEC, Recoveries, Problems & Safe Object) was built to test the whole system.

Do we need Java?

One interesting aspect was that once that all the MEC infrastructure was in place, the resulting environment was very similar to Java. From type checking to safe pointers, many Java features were available with the performance of compiled code. In such conditions, one can build very stable applications.

The problem lies in the *can* word. In C++ programmers can write proper code. In Java it is very difficult to do anything else. Many sources of ugly code have simply been eliminated.

We wouldn't need Java if we could breed a new sort of programmers, in the meantime, Java is a good thing…

Implementation

The whole system was built with no regard for existing systems. To build a MEC system on this C++ infrastructure, you have to build the new system on top of the MEC system. This mean transforming existing code into MEC code is rather difficult.

To test the management system, a simple stream component was built on top of the buffered stream library (`fopen, fscan, fclose`) . The implementation was done in ANSI C++ with Exceptions.

During the design, the integrity check and the fact that all object inherited from a safe object made checking and debugging much faster and easy. This testing implementation was very useful to determine if the architecture was usable and what structure could and needed to be simplified.

Test Application

The test component is a management enhanced version of the buffered input/output library. The test application simply tried to open a non-existing file and then to read it its content even after the end of the file. The Manager has two recovery functions to correct those problems. The first recovery function tries to find a file by changing the suffix. The second recovery function simply returns a empty string if the program tries to read after the end of the file (the original library returns the last valid string).

What was left out…

The original specification included many additional concepts/objects that were left out during the implementation phase. Most were abandoned because the concept were not refined enough.

Locking

One idea was the possibility to lock an object. A locked object cannot be changed until it is unlocked. This would prevent components from changing objects that are not supposed to be changed. This access control makes sense in language were every object is transmitted by reference (C and descendant languages).

Locking inside the language

The locking idea is technically a good idea, but should be inside the language, not the management package. Languages like C++ or Java do not check the code for read/write permissions on objects. This limitation should be corrected inside the language…

NaO Objects

Literally Not an Object Objects. A Boolean value inside each object permitted to decide if a object was valid or not. The original plans implied rebuilding all existing types with a NaO object. This meant handling NaO integers and even NaO Booleans. The NaO scheme was not implemented in the C++ prototype, because the NaO concept was not clear enough at the time.

NaO implementations

There are two ways to implement the NaO concept. The first idea is to decide that any object can be a NaO. Any valid object is transformed into a NaO by changing a flag inside the object.

The second approach is to use an unique symbol to represent the NaO (like the null pointer) and to define operations on this special symbol.

General Structure

The implementation was rather straightforward from the first specification (see appendix B). Methods were grouped into *suites*, that is, logical groups of functions and procedures. This idea is very similar to the concept called *Interfaces* in Java.

One idea was that each *suite* would be an object and that the final object would inherit from each object defining one of the suites. This design was to complex and posed many encapsulation problems and was thus abandoned.

In a Manager there are two king of problems, MEC problems that need to be recovered and internal problems (i.e. when the Managers fails). Both kind of problems are represented as exceptions, to distinguish them, they have different types. The Management System's exceptions are integers, the problems of the client components are descendants of the problem object. This made the whole design simpler, but meant that no meta-management was possible.

The Safe Object (MESO) , the MEC, the Manager, the Problem and the Recovery were each implemented in a separate class. Two additional objects were specified: the cure object and the estimate.

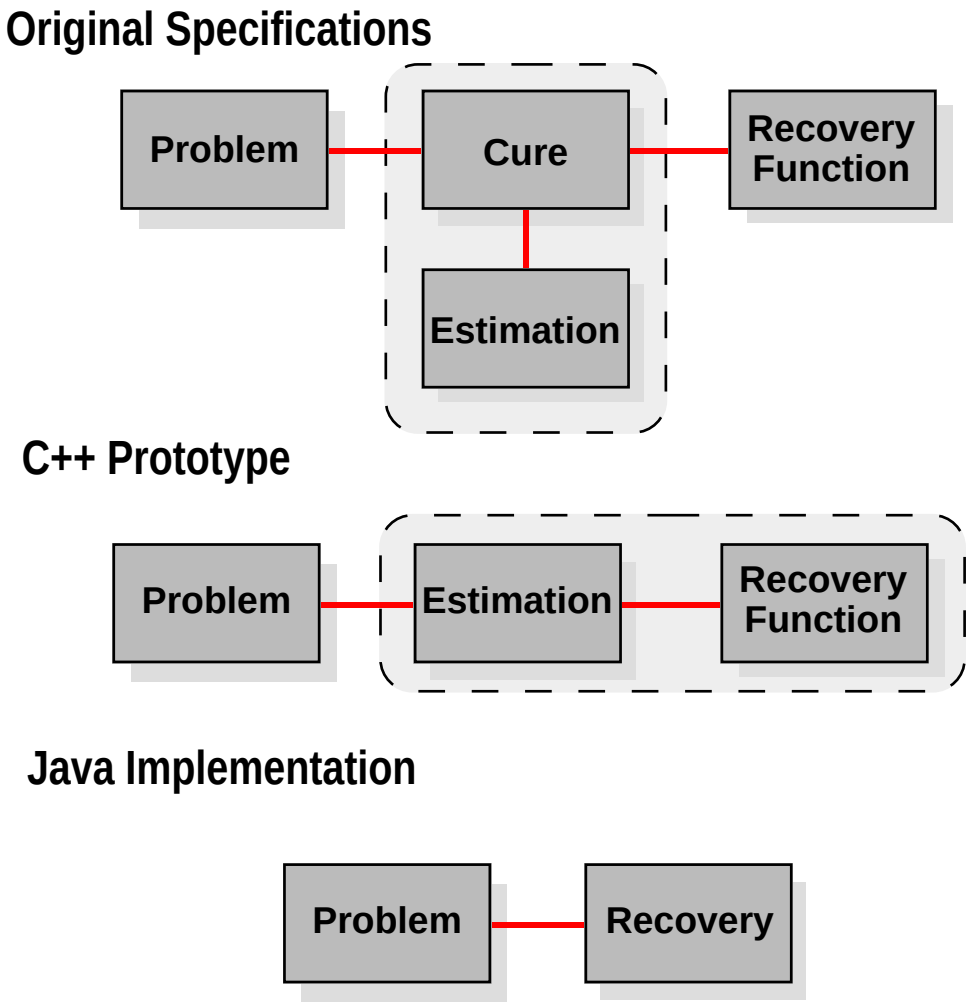

Figure 21 - Object Evolution

Figure 21 shows the evolution of the object Structure. In the original specifications the cure object was simply a link between problem and recovery. The Estimation class contained the estimation of a recovery for a given problem (so an Estimation was linked to a Cure, because the Cure was a link between problem and recovery). The Cure object was abandoned in the C++ design, all the functionality was transferred into the Estimation. The Estimation object was removed in the Java Implementation and all functionality was transferred into the Recovery object.

Most type checking tasks were not delegated to a check-MEC, but inherited from the basic MEC object. This made coding much simpler but gave a mixed design: the original MEC object had to contain all the checking code for all types.

2b – Implementation

MESO Implementation

Self Test

The safe object was built with a copy of the implicit `this` pointer in private storage (it was called `that`). This means that any pointer on a Safe Object could be checked by comparing the pointer with the copy. The system could check that:

```
pointer -> that = pointer
```

If this condition is not verified, the pointer is corrupt. This scheme is very useful, because it gives the system (and the programmer) a way of checking every pointer with a very simple routine. It helped debugging in every stage of the system.

Type

Each sub-class of the MESO overloaded a `get_object_type` method, and returned an unique type number. This system provided a simple type checking mechanism for pointers. A much more effective type system is implemented in the RTTI C++ extension. This is a standard C++ proposition. It was not used in the C++ prototype.

MEC Implementation

The MEC Object inherits directly from the MEC Object. It contains a private pointer to the Manager, and special methods to communicate with it. It offers a dispatching suite, and some checking code. This code is called MEUF (Management Enhanced Utilities & Functions).

The dispatching code tends to take a lot of place and was simplified in the Java version. One idea was to insert the problem dispatching code inside the problem object constructor. This way, the problem could be corrected before the `catch` statement. This idea was used in the Java version.

The checking services had to be defined once and for all in the basic MEC object. This limitation lead to the concept of a specialized MEC, the Check MEC.

Manager Implementation

The Manager inherits directly from the MEC object. One idea was to overload its MEUF functions with management functions. That is a client MEC's own dispatching methods called the same methods on the Manager as a client.

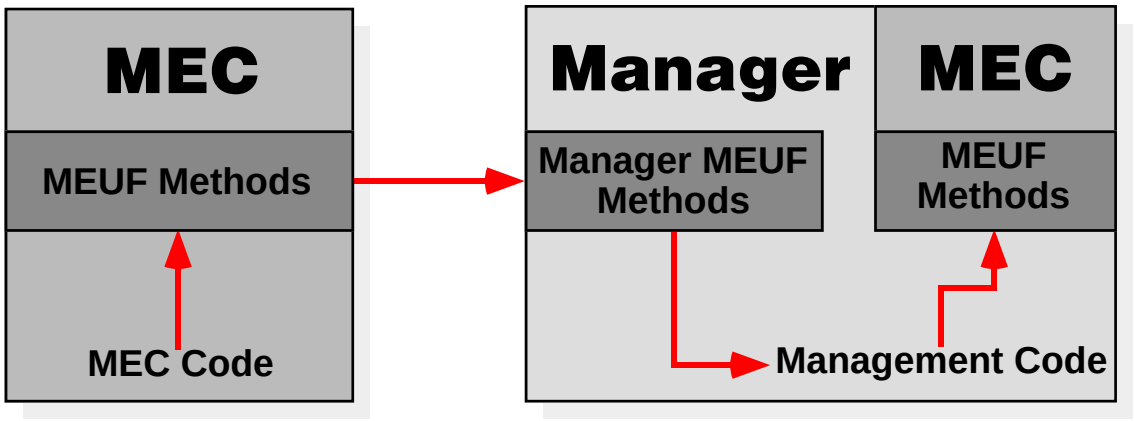

Figure 22 - MEUF Utilities Linking

Figure 22 shows the MEUF structure. Each MEC inherits the MEUF methods from the original MEC Object. Each of these methods checks if there is an attached Manager and the calls the corresponding method in the Manager.

Because each Manager is also a MEC object it inherits all MEUF methods. But all these methods are overloaded to actually offer the service. If a problem occurs, the Manager calls its MEUF methods, but not the ones it overloaded, but the ones it inherited from the MEC object.

Problem Implementation

Because of structure problems, the MEC internal errors are handled using an alternative exception system. Those exceptions are of type integer.

The problem structure follows two distinct criteria:

- the vectors in the error space.
- the object hierarchy, linking errors that touch the same application domain.

The problem object serves as a container for the problem context. This also means that all relevant data must be included in the constructor method. This requires a careful building of the constructor. Anyway, because custom constructors cannot be inherited in C++, each new particular problem will require a particular constructor.

Error Recovery Function

The Estimate class offers a mean of communication between the Manager and the recovery function objets, but should also serve as a memory container between evaluation and problem solving time. In the current state of the system, the Estimate class provides one integer selector for communicating between the estimate and the problem solving phase. It is clear that a more complex mean of communicating is necessary.

In the Java implementation, the recovery function and the estimate object were merged into one Recovery object. This object contains the recovery code, the estimate and private data that

2c – Error Dispatching

Figure 23 shows a sample of C++ dispatching code. The checking is done inside the MEC. When an assertion is violated, an exception is thrown and caught in the main block of the MEC. The catch block then calls a dispatching method (`ME_dispatch_problem`), this method checks if a Manager is present. If this is the case, the corresponding method (`ME_dispatch_problem`) is called on the Manager. The Manager then handles the problem.

The MEUF utilities also contains some specialized methods, that do precondition checking. Those methods throw exceptions that are caught in the main block.

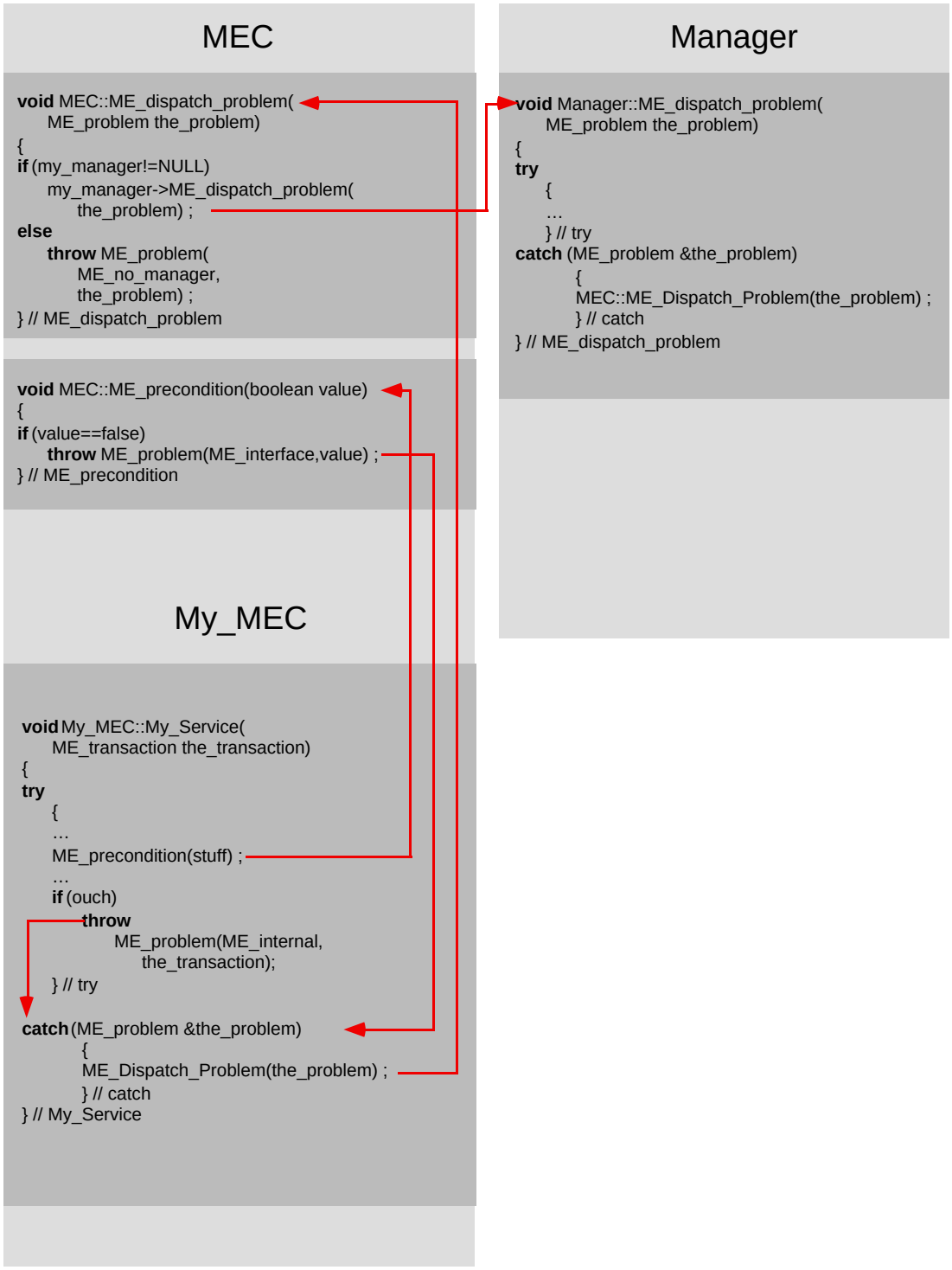

Figure 23 - C++ Prototype Structure, Exception Dispatching

3) Java MEC

3a – Introduction

Java MEC Overview

The Java MEC was built with the experience learnt on the C++ prototype. The object structure was simplified: the C++ prototype used to many different objects. The Recovery function, the Estimate object and the Cure object were collapsed into one object: the Recovery (see Figure 21).

Java Interfaces

The Java language define a special kind of object: interfaces. Interfaces are abstract object that only contain methods. An interface can be seen as a collection of methods.

This concept was very similar to the "suite" idea that was used in the C++ design. The whole Java MEC design was built around those interface in mind.

Each object in the Java-MEC system is an interface, this means Managers, Problems, MECs etc. are only accessed through standard function calls.

The dispatching procedure was also greatly simplified. Instead of doing all the dispatching in a main block, the dispatching is done in the problem class constructor. This means the Manager is called and tries to do the dispatching *between* the `throw` and the `catch` clauses.

Java Networking

Java offers an interesting access to network resources as well as local files. The most straightforward mean to access networked files is the URL Object. An URL (Uniform Resource Locator) is an address permitting access to any resource on the Internet. Java implements such an address as a object. This object has different methods to access the content pointed by this address.

Because the URL is the address and not the retrieving mechanism, I prefer to refer to the mechanism as the URL-loader: the functionality that does the actual work of fetching resources in the network.

The actual complexity of loading resource through different networks is handled by the java.net package. The URL scheme also permits access to resources on the local file system (see Figure 24)

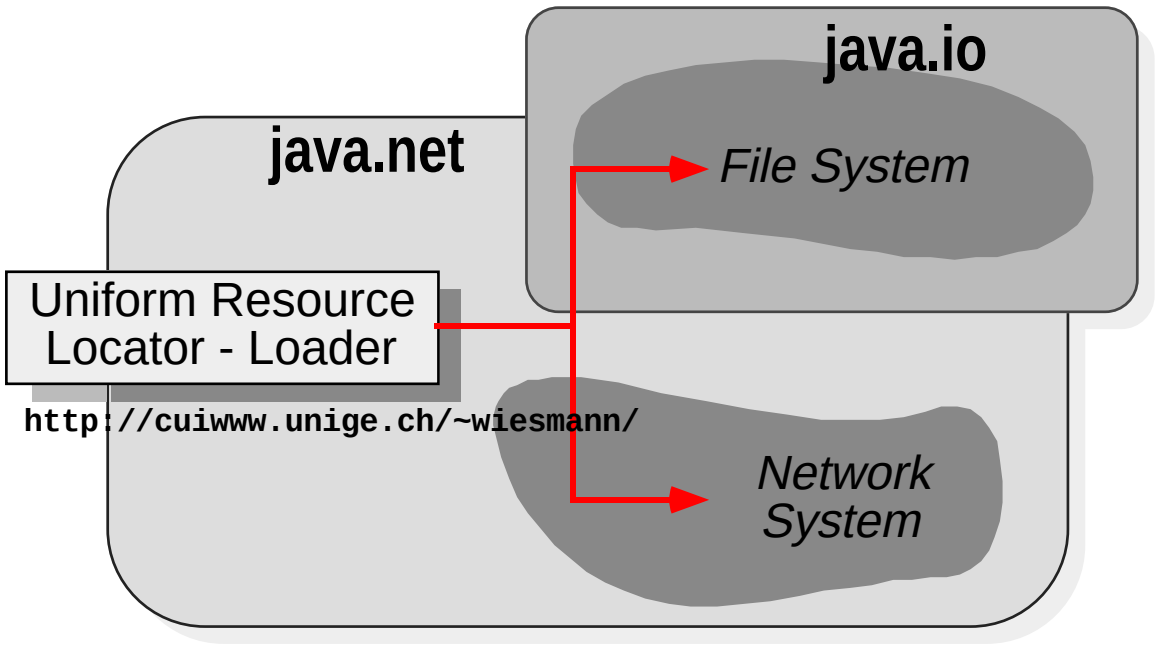

Figure 24 - URL - Loader Structure

Because this system is very flexible, it can also encounter problems of many kinds. Because of this we decided to build a Management-Enhanced version of the URL-loader.

URL Usage

URL are used in many ways, but the most common is inside World Wide Web pages. The underlying data format, HTML (Hyper-Text Meta-Language) can be edited both by human using simple text editors, complex page composing applications, or automated tools. Because of this, Web pages and the URLs inside often contain errors or inadequacies. So building a fault tolerant URL loader is very appropriate.

Another reason for building a Management-Enhanced URL loader is that the World Wide Web is changing very fast. New protocols and data formats are proposed regularly, all of whom are accessed by URLs. By building corrective code to handle new formats and protocols, one can avoid rewriting complete applications to supports new standards.

3b – Test Application

Management-Enhanced URL - Loader

For this implementation, we decided to build a management enhanced version of the URL object (URL-loader). The MEC object offers the same basic functionality than the original URL object. In addition, this MEC URL loader has a simple Manager attached and can recover from its errors.

Test Application

The test application tried to open a malformed URL pointing on a non-existing HTML file and tried to read its content. To correct those problems, the Manager had a recovery that tried to extract a filename from an URL. A second recovery tried to find a corresponding text file and translated it into a HTML file.

By including the whole correction system inside the problem constructor, an occurring problem is dispatched between the `throw` and the `catch` clause. This means that in an application using a Management-Enhanced URL, when a exception is caught, it may already be corrected.

This can be checked simply by calling a method (`isCorrected`). If the problem is corrected execution may resume. If a function-call has been interrupted and the return parameter was lost, this parameter is also stored inside the exception.

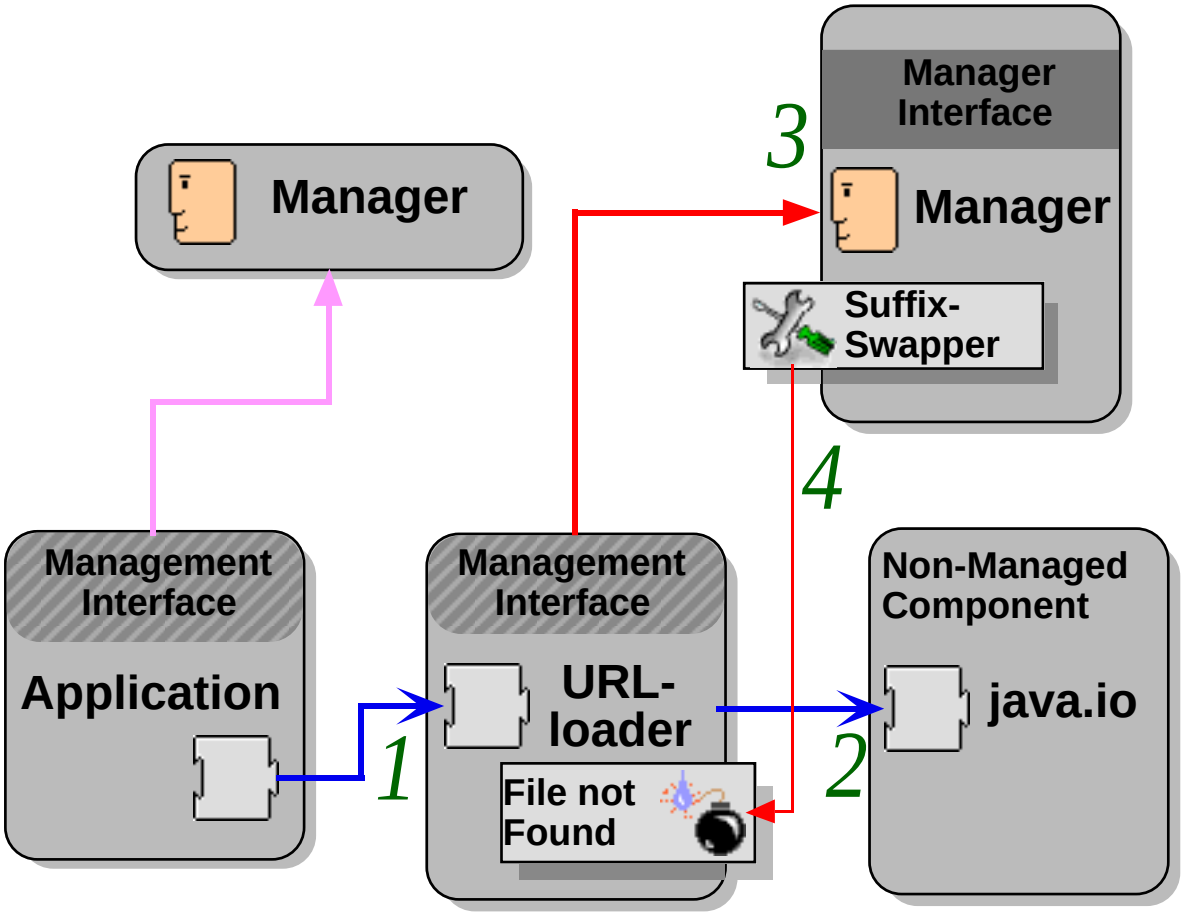

Figure 25 - URL Correction Test Application

Figure 25 shows a typical problem dispatching. The application tries to open a valid URL that points on no file (`file://foo.html`). The URL-loader calls the `java.io` component to handle the file operation. `Java.io` is not managed. An error occurs: the file does not exist.

The "File not Found" problem is created in the URL-loader component. The problem is dispatched to the URL-loader's Manager. The Managers inspects the problem and asks to all its recoveries to estimate themselves according to the it. The manger then selects the suffix swapper recovery. This recovery tries to swap the suffix (hence its name) with a compatible suffix. In this case the file (`file://foo.htm`) is opened and execution can resume.

Basic Manager

The Manager used by the MEC URL loader is a very simple Manager. This Manager is built around two vectors, the first containing all client MECs, the second containing all recoveries linked to the Manager. When a problem occurs, the Manager scan its recovery vector and asks each recovery to estimate itself

accordingly to the problem. The Manager selects the recovery with the highest reliability and applies it to the problem. If the recovery fails, the Manager does not try to recover the problem further. All Manager functionality are accessible through the Manager interface.

Expanding this simple design raises several issues.

- A Manager trying other recoveries when one fails risks doing cycles. To avoid this, the Manager must be able to detect cycles.
- A Manager operating in a multi-threaded application or doing meta-management must be fully reentrant. One way of achieving this is by cloning the whole Manager with each management request. A more economical approach would be to clone only selected recovery objects.

URL Loader recoveries

The first recovery object is one that tries to find a path pointing to a local file inside a garbled URL. This permits to correct problems occurring when an user enters an valid UNIX or DOS pathname instead of a valid URL.

The second recovery object is called a suffix swapper. Most files designated inside URLs have a suffix specifying their type (`.C` for C source files, `.html` for HTML files). Those suffix are not a fixed standard, rather a common usage, this means that many different suffixes may designate the same file type. Theoretically those suffix can have an arbitrary length, in upper or lower case, but certain operating system impose limitations on suffixes. MS-DOS for instance only accepts three character uppercase suffixes.

This means that an HTML file can have the `.html` suffix on a Unix Machine and the `.HTM` suffix on a DOS machine. The suffix swapper contains lists of compatibles suffixes and tries to swap them. For instance the reference `"file://exemple.html"` could by replaced by `"file://exemple.htm"`.

The third recovery object tries to translate a text file into the HTML format if the requested HTML document does not exist. One interesting thing was that the translating recovery was built upon a simpler recovery, the suffix swapper. The reason was that the suffix swapper already "knew" what format had what suffixes.

3c – Packages

The Java MEC is structured in packages. This way objects can be precisely identified.

Frame & Problem Package

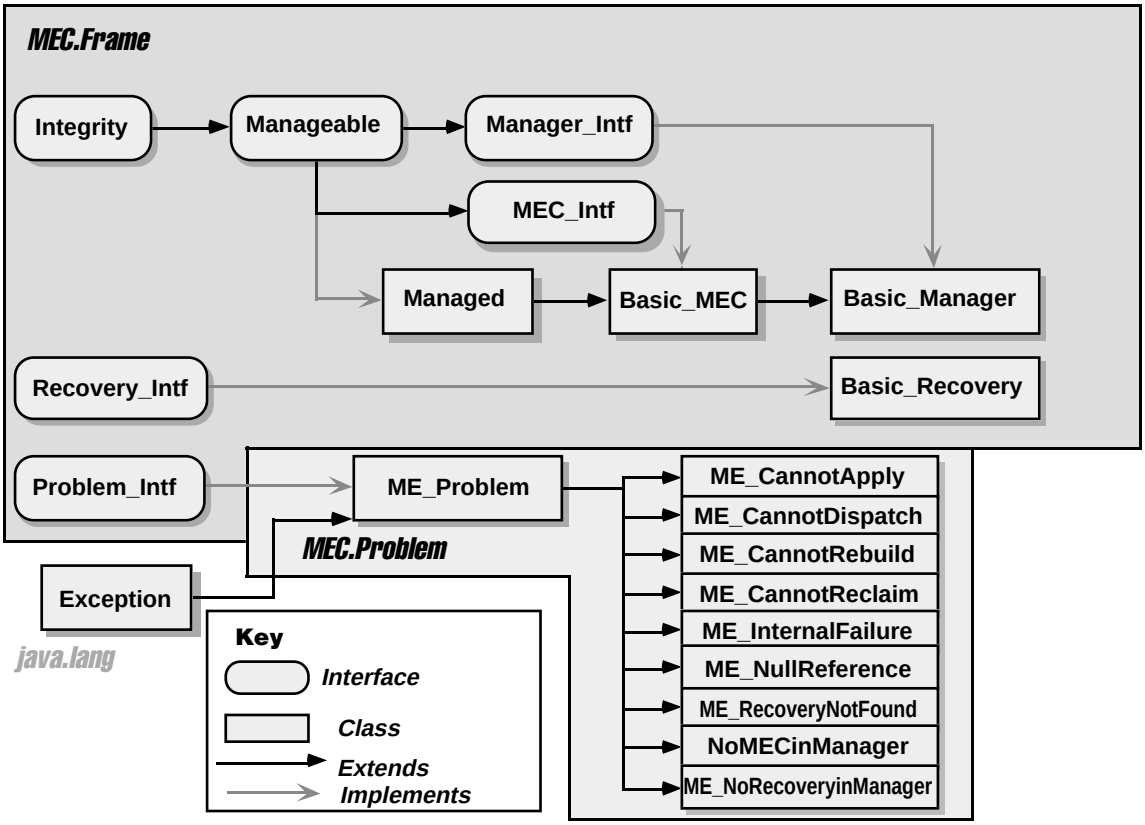

Figure 26- MEC.Frame Hierarchy

The general MEC infrastructure is built into two packages: `MEC.Frame` and `MEC.Problem` (see Figure 26). The first contains the general MEC frame, the second contains all Problems that could occur inside the Frame. The Frame package offers a basic Manager that is very simple but fully functional.

The Frame Package also offers a basic MEC and a basic Problem object that can be derived upon to build a MEC and its related problems. The Frame Package only uses `Java.Lang` and `Java.Util` objects, it should therefore be usable in most Java implementations.

MyNet Package

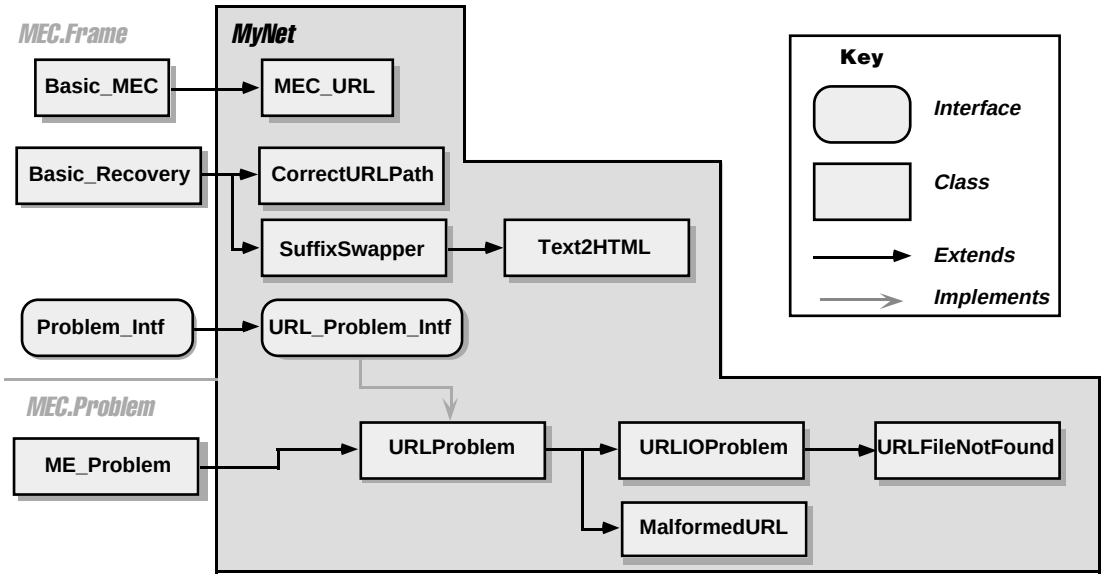

Figure 27 - MyNet Package Hierarchy

A third package, MyNet implements a Management-Enhanced URL Object (see Figure 27) Along this object are a few recoveries and the related problems. This package uses the MEC.Frame and the MEC.Problem packages.

3d – Interfaces

The whole MEC system is built around interfaces. All MEC methods manipulate interface rather than objects. Each MEC element is implemented as one or more interfaces. Any object can therefore be included into a MEC system simply by implementing the required interfaces.

Because all object communicate through their interfaces, they can be swapped easily. This also permits and simplifies code reuse.

Standard MEC Interfaces

`Integrity` `(Integrity)`

This interface implements all of the safe-object service that are not included in the basic Java Object. This means the integrity checking services.

`Manageable` `(Manageable)`

This interface extends the Integrity interface. It sole interest is to simply the design of the MEC: instead of implementing the whole management interface in one piece, the manageable interface offers a part of the management interface.

A manageable object is one that is linked to a Manager. A manageable object can register to a Manager and keep a reference to this Manager. This interface is not much use by itself, but serves for the Managed Object and the Manager Interface.

`MEC Interface` `(MEC_Intf)`

This interface extends the Manageable Interface. This interface specifies all methods an object must implement to be a MEC. This includes giving information to the Manager, and transmitting known recoveries.

Manager Interface (Manager_Intf)

This interface extends the Manageable Interface. This interface specifies all methods an object must implement to be a Manager. This includes registering MECs and dispatching trouble.

Problem Interface (Problem_Intf)

This interface contains all methods a problem must define. This implies mostly giving information about the problem: localization, problem type, etc. The MEC system only requires that a problem object implements the Problem Interface. But in order to work inside a `throw` statement, they must be descendants of the `java.lang.throwable` object.

Recovery Interface (Recovery_Intf)

This interface defines all methods a recovery must define. This implies building itself around a problem, giving a estimate of the recovery conditions and do the actual recovery.

Interface Advantages

One interesting idea arising from this Interface-based design is the possibility for objects to perform more than one service. MECs could be Managers, MEC could be Recoveries.

This possibility would be very useful in case of simplified design where a complete MEC design would be to heavy. Those possibilities have not been used in the current implementation.

Recovery & MEC

One idea would be to build recoveries into components. Those components would of course be managed. This way, they could be linked to a Manager and could recover their problems. Those components would implement both the recovery and the MEC interface.

Another interesting option offered by this structure is that a MEC object can also declare itself as its own recovery. For instance, a network connection object could implement the recovery interface as so be able to handle its own trouble. This is very important for code handling unstable resources (like a network connection), this code might want to have a direct access to trouble when it happens. By implementing the recovery interface, such an object might do privileged error handling without interfering with the general MEC structure.

MEC & Manager

One interesting prospect of this structure is the possibility to build auto-managed components. By implementing the Manager interface, an object could manage itself. That is use himself as a Manager by default. A more complex Manager could by inserted at any time. This could prove rather useful in situations were a very simple Manager would be required.

Recovery & Manager

By implementing the recovery interface, a Manager may be used by another Manager. This is particularly simple way of implementing a hierarchical Manager structure (see Hierarchical Managers page 36). Each specialized Manager is declared as one powerful recovery.

3e – Objects

The MEC system offers some public objects: most of them are not functional as such but serve as a basis to build managed systems. By inheriting most methods, it is easier to build a MEC, a Recovery etc.

Basic MEC (Basic_MEC)

This object implements the MEC interface but does nothing. In is a "dummy MEC". To be used, most information method must be overloaded.

Basic Recovery (Basic_Recovery)

This object implements the Recovery Interface. This recovery does nothing, it is a "dummy Recovery". if called it always estimates it cannot recover the problem, if asked to apply itself, it throws a `cannot_apply` exception.

Management-Enhanced Problem (ME_Problem)

This object is built upon the java.exception Object. It implements the Problem Interface. All exceptions thrown by the MEC system are sub-classes of the ME_Problem.

Basic Manager (Basic_Manager)

This object is a fully functional Manager. Its recovery scheme is very simple: it scans its know recoveries for the one with the best Reliability and applies it. It also support all methods required in the Manager interface: MECs and Recoveries can register and unregister to the Manager.

3f – MyNet Package

General Structure

This package implements a Management-Enhanced URL Object. This object offers the same basic services as the `java.net.URL` object. Each URL is considered to be its own MEC. By default, all instance of this MEC URL share a common Manager that is built when the first URL is built. The package also defines some specific errors and recoveries.

Problem & Recoveries

Here are some URL related problems and their recoveries.

Problem	Explanation	Error Type	Recovery	Does What
`Malformed URL`	The URL is malformed	Invalid Reference	`CorrectURLPath`	Searches the URL for a file path
`URLFileNotFound`	The file is not found	SubRequestProblem	`SuffixSwapper`	tries to find a compatible file with a different suffix
`URLFileNotFound`	The file is not found	SubRequestProblem	`Text2Html`	tries to translate a text file into a html file.

The problems object were built upon the ME_Problem object, the recoveries were built upon the `Basic_Recovery` object.

3g – Java MEC Conclusions

The Java MEC implementation is perfectly functional. The Manager used is very basic in design and could be upgraded in many ways. An advanced implementation could include a reentrant version and multi-threaded recovery searching. One elegant idea would be to store recoveries inside the Manager as class identifiers instead of the object instances. The Recoveries would only be instanciated as needed.

An implementation with many communicating Managers would require the implementation of a log object. This would also be very useful in case of a fully reentrant Manager. Although the specifications for the log object were done (see appendix A), it was not built.

Component structured Recoveries would also help meta-management. This means the recovery objects would also implement the MEC interface and be managed. Meta-management would also require cycle detecting schemes to avoid recovering recoveries recovering recoveries…

Conclusions

1) General Conclusion

The general MEC was built successfully twice. Both implementations are very simple but yet fully functional. The C++ prototype improved the overall stability of any program by checking pointers to safe objects. The C++ prototype was used to implement a management enhanced stream library. The Java MEC was used to build a Managed URL Object. Both were able to perform simple recoveries on given problems.

The problems that both system corrected were trivially simple, yet such trivial problems can completely cripple even complex applications. How many applications cannot start because they lack a file? Simply by correcting those problems, the overall stability and responsiveness of a system will increase.

2) Design Conclusions

2a - Low Level & large Scale Implementation

By implementing the MEC system deeper in the virtual machine/operating system, we will be able to build very robust components. A large scale implementation, with all services implemented as MECs, would permit to build solid, fault tolerant applications.

2b - Recovery Structure

Instead of building complex recovery systems, small recovery elements should be used. A smart Manager will be able to chain many small recoveries to build a solution for complex situations.

By implementing the recovery code, like everything else, into MECs, we will ensure that those recoveries are able to work even in crisis situations with lacking resources.

2c - Component Reuse

By using the standard management component and interfaces, all elements can be exchange freely. This means that all management elements can be reused. Because Management-Enhanced Components can interact with non-managed components, MECs can be reused in non managed application and still retain their management functionality.

All basic MEC components can be used to build more advanced management systems. For instance to build a specialized Manager, one can inherit most features from the Basic Manager.

2d - Conclusion

As conclusion, the MEC system is both functional and useful. A further step would be a full implementation. This would include fully managed services, multiple Managers, meta-management, recoverable recoveries and dynamic Manager loading.

3) MEC Java Wishing List

Here a some wishes that would make a full scale MEC system more simple to implement or more efficient. Most ideas could apply to any language, they are simply explained with Java in mind.

3a – Low Level MEC Implementation

By implementing the basics of the MEC system directly into the Java virtual machine, we could extend this already safe language, and make it completely managed.

Low Level Safe Object

The whole MEC scheme would work best if some hooks could be implemented directly in the Java machine. The original Java object implements already a lot of methods (hash-table, cloning, string conversion) so adding integrity functions directly in it should not pose such a problem. Such a low-level implementation would simply the generalized use of Safe-Object.

Read-Only References

Even if bogus memory writes are theoretically impossible in Java, objects may get corrupted by bugged components. This is particularly true because Java uses the same parameter passing mode than C. All objects are passed by reference and so it is not possible to declare that a method may not transform the object referred. A special keyword to forbid write access to the object would enhance safety.

MEC Runtime Exceptions

We can perfectly rewrite a MEC version of all Java API objects, and make them throw MEC-problem objects. The one thing that cannot be overloaded is the runtime package and the runtime exceptions. This is most annoying because runtime exceptions can occur anywhere. They contain many problems that occur often and could be corrected (`null` reference problems, invalid cast operations etc.).

3b – Java Extensions

Null Methods

One idea to implement the `NaO` object is to use the `null` reference. The problem lies in the `null` reference runtime exception. Like all runtime exceptions, it is hard to catch as there is no way of knowing when it will happen.

The idea was to give the `null` reference some special methods. One basic idea would be to implement all basic Object methods like `getClass` and `toString`. This way, MEC could manipulate `null` references like `NaO` Objects.

Context Object

Java is built on the idea that everything is an object. One thing that cannot be seen as an object is a method. This may sound absurd, but each call to a method can be seen as the instanciation of an "method" object. This object can be multiply instanciated in case of re-entrance or multi-threading. It contains public data (its parameters) and protected data (the local variables). Each time a method is overloaded a descendant object is built.

A function object even exists in memory, simply it is allocated on the stack rather than the heap, it is built each time its constructor is called, and destroyed when control reaches the end of its code or a return statement is detected.

Ok, but what's the point of such an object? When throwing an exception, we want to get a copy of this context. So if could be accessed directly through a special keyword (`now`?), we could explicitly copy the content into the exception we are throwing.

Range Checking

One feature found in Pascal is the possibility to define a specific range for numbers. This is particularly useful for indexes and other state information we know must stay in a given range. For instance we know the day in the year cannot exceed 366, still without checking we could find ourselves in 369, which is impossible.

If performance can explain the absence of this feature in the `integer` and `float` types, it should at least be implemented in some way in the `java.lang.Number` container object.

Resumable Exceptions

When an exception is raised the control goes to the `throw` statement and then to the `catch`. This means that the context (all local variables) is lost because the `catch` statement is not in the same code-block that the `throw`. Not only are we outside this context, but it is destroyed.

Do Resumable Exceptions Exist?

The Dylan language permits to way to handle an exception: by **recovery** or by **exit**. In the first case the control remains in the unit, in the second, control is transferred to an external unit.

The same feature exists in the Win32 API, except that the exception type is limited to unsigned integers.

This means that the context must be saved (in our case, copied into the problem object by the problem constructor), but also that execution cannot resume where is stopped (because this point is in the context and is destroyed). This means that we cannot return to the point where the problem occurred when the problem is corrected because this point exists no more.

A resumable exception is one you can get back from. This means that the original context is saved in memory. Once the problem has been corrected in the exception handler, execution can resume at the point it stopped. Resumable exceptions would greatly simplify the design of the MEC system.

One idea to implement resumable exceptions in Java would be to create a new thread with each exception. The old thread retains the context and is blocked. The new thread goes into the exception handling block. If the exception can be

corrected, the original thread is un-blocked and the new one killed, if the problem cannot be cured, the original thread with the context is killed.

In fact both exceptions could coexist in some way, for instance in case of a Warning: the code continues to execute normally, but a Warning Exception is raised and sent to the program code, so that it can react.

Bibliography

1) Articles

[1] Software Rejuvenation: Analysis, Module and Applications

Yennun Huang, Chandra Kintala, Nick Kolettis and N. Dudley Fulton
AT&T Bell Laboratories
Proceedings of 25th International Symposium on Fault Tolerant Computing.

[2] My Hairiest Bug War Stories

Marc Eisenstadt
Communications of the ACM April 1997 - Volume 40, Number 4

[3] Implementing Efficient Fault Containment for Multiprocessors

Mendel Rossenblum, John Chapin, Dan Teodosio, Scott Devine, Tirthankar Lahiri & Anoop Gupta.
Communications of the ACM September 1996, Volume 39, Number 9

[4] An Essential Design Pattern for Fault-Tolerant Distributed State Sharing

Nayeen Islam & Murthy Devarakonda
Communication of the ACM October 1996, Volume 39, Number 10

[5] Vax 6000 Error Handling: A Pragmatic Approach

Brian Porter
Digital Technical Journal, Summer 1992, Volume 4, Number 3.

[6] Verification of the First Fault-Tolerant VAX System

William F. Burckert, Carlos Alonso & James M. Melvin
Digital Technical Journal, Winter 1991, Volume 3, Number 1.

[7] Checking the Integrity of Trees

Johnatan D. Bright, Gregory F. Sullivan & Gerald M. Masson
Proceedings of the 25th International Symposium on Fault Tolerant Computing.

[8] Mechanisms for Detecting and Handling of Timing Errors

David B. Stewart & Pradeep K. Khsosla
Communications of the ACM, January 1997, Vol. 40, Number 1.

2) Books

2a – Technical Reference

[9] Inside Macintosh: Operating System Utilities

Chapter 2 - System Error Handler
Apple Computer
Addison-Wesley ISBN 0-201-62270-X

[10] Inside Macintosh: Quickdraw GX Environment and Utilities

Chapter 3 - Errors, Warnings, and Notices
Apple Computer
Addison-Wesley

[11] Java Virtual Machine Specification

Sun Microsystem Computer Corporation
Java Virtual Machine Specification PDF File

2b – Language Reference

[12] C++ Primer

Appendix B. Exception Handling
Stanley B. Lippman
Addisson-Wesley ISBN 0-201-54848-8

[13] Dylan Reference Manual

Chapter 7, Conditions - Exception Handling
Andrew Shalit
Addison-Wesley ISBN 0-201-44211-6

[14] Java Language Specification

Chapter 11 - Exceptions
Sun Microsystem Computer Corporation

[15] Ada avec le Sourire

Chapitre 9 - Les Exceptions
J.-M. Bergé , L.O Donzelle, V. Olive, & J. Rouillard
Collection Informatique - Presse Polytechniques Romandes

3) Technical Notes

[16] Bus Error Handlers

Wayne Meretsky, Rich Collyer, Colleen K Delgadillo.
Technote DV 04 - Apple Computers

[17] Visually Basic: Error Handling for Multi-user Database Applications

Robert Eineigl
Information Technology Group - Microsoft Corporation

[18] Exception Handling in Delphi

Ray Knopka
Raize Software Solutions Inc.

[19] Exception Handling in C

Ben Eng
Ben Eng (engb@ican.net)

[20] Exception and Error Handling

Marshall Cline <cline@parashift.com>
C++ FAQs lite

Glossary

API	Application Programmer Interface
CPU	Central Processing Unit
CRC	Cycle Redundant Code
GIF	Graphic Interchange Format
GUI	Graphic User Interface
ME	Management-Enhanced
MEC	Management-Enhanced Component
MESO	Management-Enhanced Safe Object
MEUF	Management-Enhanced Utilities & Functions
NaN	Not a Number
NaO	Not an Object
NFS	Network File System
OO	Object Oriented
OS	Operating System
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
ROM	Read Only Memory
SO	Safe Object
TIFF	Tagged Information File Format
TMFA	Too Many Acronyms
URL	Uniform Resource Locator
WIMP	Windows, Icons, Menus & Pointers
WWW	World Wide Web

Appendixes

This part contains specifications and other appendix documents.

A) Java MEC Specifications

This document contains the final specifications for a implementation of a MEC system in the Java Language. The Java implementation does completely comply to these specifications, but some functionality were not build in.

Integrity Interface

`integrity`

use:	this service tells if the object's integrity is intact or not
returns:	a Boolean value
access:	any component may access this service

`canRebuild`

use:	this service tells if the object's integrity can be restored
returns:	a Boolean value
access:	any component may access this service

`canReclaim`

use:	this service tells if the object can reclaim some resources, so the system can work despite low resource conditions
returns:	a Boolean value
access:	any component may access this service

`Rebuild`

use:	this service tells an object to rebuild itself
returns:	-
access:	only Manager and recovery code should access this function

`Reclaim`

use:	this service tells an object to reclaim resources
returns:	-
access:	only Manager and recovery code should access this function

`isNaO`

use:	this service tells if an object is a NaO (Not an Object)
returns:	-
access:	any component may access this service

Manageable Interface

getManager
use: this service returns the object's Manager
returns: a Manager interface reference
access: only the object should access this service

registerManager
use: this service sets the object's Manager
parameters: a Manager interface reference
returns: -
access: only the object should access this service

unregisterManager
use: this service returns the object's Manager
returns: a Manager interface reference
access: only the object should access this service

dispatchProblem
use: this service tells the object to dispatch a problem through its Manager.
parameters: a problem interface reference
returns: -
access: only the object should access this service

MEC Interface

getAvailability
use: this service tells the MEC to give its availability.
parameters: -
returns: an integer representing the availability
access: any component may access this service.

getMELevel
use: this service tells the MEC to give its availability.
parameters: -
returns: an integer representing the ME-Level
access: any component may access this service.

getVersion
use: this service tells the version number of the MEC.
parameters: -
returns: a version number
access: any component may access this service.

isStateless
use: this service tells if a MEC is stateless. As stateless MEC does not retain information between services.
parameters: -
returns: a Boolean
access: any component may access this service.

hasClassManager
use: this service tells if a MEC has a class Manager. A class Manager is a default Manager attached to the object's class.
parameters: -
returns: a Boolean
access: any component may access this service.

BuildClassManager
use: this service tells a MEC to try to build its class Manager.
parameters: -
returns: -

access: any component may access this service.

loadRecoveries

use: this service tells a MEC to load the recoveries it knows into a Manager.
parameters: a Manager interface reference
returns: -
access: any component may access this service.

getClassManager

use: this service returns the object's class Manager
parameters: -
returns: a Manager interface reference
access: any component may access this service.

doExceptionRecovery

use: this service tells a MEC that a exception could not be recovered.
parameters: -
returns: -
access: only the Manager should access this service
note: when receiving this signal, a MEC should try to rebuild its internal data structures.

doGracefullDeath

use: this service tells a MEC that it will be terminated.
parameters: -
returns: -
access: only the Manager should access this service
note: when receiving this signal, a MEC should try to save its internal data structures. Because of the situation, the MEC should not try anything complex or involving other components.

Manager Interface

registerMEC

use: this service tell the Manager to register a MEC.
parameters: a MEC interface reference
returns: -
access: any component may access this service.

registerRecovery

use: this service tell the Manager to register a recovery.
parameters: a recovery interface reference
returns: -
access: any component may access this service.

unregisterMEC

use: this service tell the Manager to unregister a MEC.
parameters: a MEC interface reference
returns: -
access: any component may access this service.

unregisterRecovery

use: this service tell the Manager to unregister a recovery.
parameters: a recovery interface reference
returns: -
access: any component may access this service.

findRecovery

use: this service tell the Manager to find a recovery to a problem.
parameters: a problem interface reference
returns: a recovery interface reference
access: any component may access this service.
note: This service is used by the treatProblem service.

treatProblem

use: this service tell the Manager to treat a problem.
parameters: a problem interface reference
returns: -
access: any component may access this service.
note: This service normally uses the findRecovery service.

Recovery Interface

`getProblem`

use: this service returns the problem associated with the recovery.
parameters: -
returns: a problem interface reference
access: any component may access this service.
note: this service should only be called after a call to the `build` method.

`getQuality`

use: this service returns the quality associated with the recovery
parameters: -
returns: an integer between `RecoveryMaxQuality` and `RecoveryMinQuality`.
access: any component may access this service.
note: this service should only be called after a call to the `build` method.

`getReliability`

use: this service returns the quality associated with the recovery
parameters: -
returns: an integer between `RecoveryMaxReliability` and `RecoveryMinReliability`.
access: any component may access this service.
note: this service should only be called after a call to the `build` method.

`getCost`

use: this service returns the quality associated with the recovery
parameters: -
returns: an integer between `RecoveryMaxCost` and `RecoveryMinCost`.
access: any component may access this service.
note: this service should only be called after a call to the `build` method.

`applied`

use: this service tells if a given recovery was applied.
parameters: -
returns: a Boolean value
access: any component may access this service.

`canWork`

use this service tells if a given recovery can work in the current situation
parameters: -
returns: a Boolean value
access: any component may access this service.

`build`

use: this service tells a recovery to build itself around a problem
parameters: a problem interface reference, and a Boolean specifying if the recovery can be interactive.
returns: -
access: only the Manager should access this service

`apply`

use: this service tells a recovery to apply itself to its linked problem
parameters: -
returns: -
access: only the Manager should access this service
note: should only be called after a call to the `build` method and only if the `applied` service returns false.

Problem Interface

`getGravity`

use: this service returns the gravity of the problem

parameters: -

returns: an integer representing the gravity of the problem.
It can be one the following: `ProblemNone, ProblemNotice, ProblemWarning, ProblemError, ProblemCriticalError, ProblemRecoveryError`

access: any component may access this service.

`getLocalization`

use: this service returns the Localization of the problem

parameters: -

returns: an integer representing the Localization of the problem.
It can be of the following: `NoProblem, InterfaceProblem , InternalProblem, SubRequestProblem, RecoveryProblem`

access: any component may access this service.

`getType`

use: this service returns the type of the problem

parameters: -

returns: an integer representing the type of the problem.
It can be of the following: `InvalidRequest , ServiceNotAvailable, ReferenceInvalid, ResourceShortage , ServiceFailure, InvalidReference, InvalidData, UnknownError .`

access: any component may access this service.

`getExplanation`

use: this service returns a text explanation of the problem

parameters: -

returns: a string reference

access: any component may access this service.

`getServer`

use: this service returns the server where the problem occurred

parameters: -

returns: a MEC interface reference

access: any component may access this service.

`dispatch`

use: this service tells the problem to dispatch itself

parameters: -

returns: -

access: any component may access this service.

note: this method searches for a Manager it the server MEC and transmit the problem to it.

`isCorrect`

use: this service tells if a problem has been corrected

parameters: -

returns: a Boolean value

access: any component may access this service.

note: a problem is corrected if is of type `ProblemNone, ProblemNotice, ProblemWarning`.

`setWarning`

use: this service tells a problem to set itself to `ProblemWarning` type.

parameters: -

returns: -

access: only the Manager and recovery code should access this service.

goodObject

use:	this service returns the object the service failed to return
parameters:	-
returns:	a reference to a generic object
access:	any component may access this service.
note:	this method is used to patch the failure in a method, it contains the data the service should have returned.

Log Interface

This interface defines the log object that is used to synchronize multiple Managers. This interface has not been implemented yet. This interface would be built on an enumeration interface (`java.util.Enumeration`), this way, all elements could be accessed in a standard way.

`logProblem`

use:	this service logs a problem
parameters:	problem interface reference
returns:	-
access:	only Managers may access this service

`lockProblem`

use:	when accessing this service, a Managers tries to establish a lock on a problem. This service returns true if the Manager is in control of the problem, if it returns false, another Manager is treating the problem.
parameters:	a problem interface reference
returns:	a Boolean value
access:	only Managers may access this service

`unlockProblem`

use:	this service lets a Manager release the lock on a problem.
parameters:	a problem interface reference
returns:	-
access:	only Managers may access this service
note:	this service can only be subsequent to a sucessfull call to `lockProblem`.

B) C++ MEC Object Access

This appendix contains the original draft MEC specification that was used to build the MEC C++ Prototype. Many aspects of these specifications were dropped during the design phase. Those specifications were completely updated and simplified for the Java implementation.

Management-Enhanced Safe Object (MESO)

The Management-Enhanced Safe Object is the basic element of the ME - system, all object within the system derive their functionalities from it. All services are grouped in suites, they are 4 basic suites: the Information Suite, the High-Level Suite, The Low Level Suite and the programming Suite.

The Information Suite can be accessed from anywhere, it only gives information about the object. The High Level suite gives the Manager system basic and general control over the object. The low level gives the Manager system access to specific or delicate control. The functionnalities in the High Level and information Suite must be implemented, the low level suite can do nothing. The Debugging suite is designed for the programmer and debugging purpose. No functional component may access its functionalities.

Information Suite

`Get_Info`

use:	this service returns information about the object in human readable form.
returns:	the information(text)
access:	any component may access this service

notes: The information should only be used to build human readable information. This service should not be used by any component to make assumption about an object.

Get_Version

use: this service returns the version number of the object.

returns: the version number of the object.

access: any component may access this service

note: This service should always be used before calling to check if the object effectively supports a given service.

Get_Object_Type

use: this service returns the type of the object.

returns: the object type of the object

access: any component may access this service

use: This service should always be used before calling to check if the object is of a given type.

Is_NAO

use: this service checks if the object is a NaO

returns: a Boolean value.

access: any service may access this service.

Is_Locked

use: this service checks if the object is locked.

returns: a Boolean value.

access: any service may access this service.

NaO Objects

Each MESO sub class should define an instance of the MESO object, references to this object should be returned each time call fails and should return an MESO reference (in the case of pointers, it should come instead of the NULL pointer).

Each operator that works on MESO should accept to work with NAOs an try to do a maximum of work.

Locks

Locks serve to lock an object in a given state, i.e. the object may not be changed until it is unlocked. Lock should have the same use as const in C++ code…

High Level Suite

ME_Integrity_Check

use: this service performs an integrity check on the object and gives the result.

returns: returns the integrity state of the object, it may be of the following states:

ME_integrity_failed	The integrity check failed, the state of the object is unknown but should be considered damaged, the check service is also damaged.
ME_integrity_lost	The object is damaged and cannot be rebuild, no service or data should be used.
ME_integrity_damaged	The object is damaged but could be rebuild, no service or data should be used before calling successfully the ME_Rebuild service.
ME_integrity_partial	The object may be damaged, proceed with caution.
ME_integrity_rebuild	The object has been damaged but has been rebuild. The data should be OK.
ME_integrity_OK	The object is in good state, it has never been damaged.

access: The Manager Service or an encompassing object may access this service.

note: This service should used by the Manager each time he suspects an object may be damaged, or when there is some idle time left.

ME_Rebuild

use: this service tries to rebuild an object, it returns the result.

returns: returns the integrity state of the object, it may be of the following states:

ME_rebuild_failed	The rebuild function could not repair the object. The object should be discarded and regenerated from another source.

ME_rebuild_partial	The rebuild function has repaired the object. It cannot certify that the object is in perfect shape.
ME_rebuild_done	The object has been rebuild and is certified to be functional.
ME_rebuild_stupid	The ME_Rebuild service has been called but the object integrity was ME_integrity_rebuild or ME_integrity_OK, so nothing was done.

access: The Manager Service or an encompassing object may access this service.

Low Level Suite

ME_Can_Reclaim

use: this service tells if an object can reclaim part of its resources, that is if it can run with less resources (memory, files, threads).

returns: an number giving the maximum level of reclaim it can reach. 0 means no reclaiming can be done. 5 means five levels can be done, the level one means the least reclamation.

access: The Manager Service or an encompassing object may access this service.

ME_Has_Reclaimed

use: this service tells to what extent is has reclaimed its resources.

returns: an number giving the actual level of reclaim it can reach. 0 means no reclaiming has been done.

access: The Manager Service or an encompassing object may access this service.

ME_Reclaim

use: this service tells an object to reclaim resources in order to use less of them. The number given has parameter gives the amount of reclamation to be done. If the new level of reclaim is higher than the actual level, the object may use more resources accordingly.

returns: the reclaim level effectively achieved, or one of the following status

ME_reclaimed_corrupt	The reclamation process failed, the object is now corrupt.
ME_reclaimed_failed	No reclamation could be done.
ME_reclaimed_impossible	The ME_Reclaim service has been called with a level the object cannot reach (higher than ME_Can_Reclaim)

access: The Manager Service or an encompassing object may access this service.

ME_give_NAO

use: this service returns a reference to a NaO (Not An Object).

returns: a reference to a NaO instance of the class.

access: The Manager service, or an encompassing object.

note: there should be only one instance of the NaO object for each class.

Debugging Suite

ME_Dump_Debug

use: this service tells an object to dumps its content and state in a human readable form.

returns: information in a human readable form

access: This service should only be used to debug a program, other object may only call this service if they are encompassing this object and executing the ME_Dump_Debug service themselves.

MESO Suite

This Suite contains some basic services for the manipulation of objects.

Clone

use: this service makes a clone of an object.

returns: a reference to the clone of the object.

access: any component or object.

Lock

use: this service locks the object. A locked object may not be changed as long as the unlock service is not used.

returns: a Booleans telling if the

access: any component or object.

Unlock

use: this service unlocks the object. A locked object may not be changed as long as the unlock service is not used.

access: any component or object.

MEC– Boolean

As the whole MEC system is meant to work even if part of it fails, it must accept a certain amount of uncertainty. To this mean the Boolean notion is redefined to accept 3 states: true, false and unknown.

One idea would be to implement a ME_Boolean, that could either be true, false or a Na0.

Management-Enhanced Collection Object (MECO)

The Collection object serves as a container for a serie of safe object (MESO).

Collection Suite

Number

use: this service gives the number of objects in a collection

returns: the object number

access: This service can be accessed by any component or service.

Index

use: this service gives an object according to the index number

params: index number

returns: a reference to a MESO

access: This service can be accessed by any component or service.

Next

use: this service gives an object following an object

params: a MESO reference

returns: next MESO reference

access: This service can be accessed by any component or service.

Prev

use: this service gives an object preceding an object

params: a MESO reference

returns: next MESO reference

access: This service can be accessed by any component or service.

Add

use: this adds an object to a collection

params: a MESO reference

access: This service can be accessed by any component or service.

Remove

use: this removes an object from a collection

params: a MESO reference

access: This service can be accessed by any component or service.

Problem Object

The Problem object is a passive object. That is, it is created when a problem occurs and is transmitted to the relevant Mangers in order to find a cure. For this reason there are only services in the information suite.

The problem object may well serve as an transaction container, this way, each time a component initiates a transaction, it builds a problem object.

Information Suite

`Gravity`

use: this service returns the gravity of the problem

returns: the gravity of the problem it can be of the following:

`ME_recovery_error`	An error occurred during the recovery process and could not itself recovered.
`ME_error`	An error occurred. It has not been yet corrected.
`ME_warning`	An error occurred. It has be recovered. Attention should be payed to whatever caused the error. It may cause other errors, or hide a system problem.
`ME_notice`	An error occurred. It has be recovered. The error should not have other consequences.
`ME_no_error`	All is well.

access: any component may access this service.

`Localization`

use: this service returns the Localization of the problem

returns: the Localization of the problem it can be of the following:

`ME_interface_error`	An error occurred during the analysis of the parameters. The serie request was invalid and rejected. The calling service may be corrupted.
`ME_internal_error`	An error occurred inside a service. The service is in an illegal state and may be corrupted.
`ME_sub_request_error`	An error occurred in an sub-service.
`ME_no_error`	All is well.

access: any component may access this service.

`Type`

use: this service returns the type of the problem

returns: the type of the problem it can be of the following:

`ME_data_corrupt`	A element of data is corrupt.
`ME_data_invalid`	A element of data contains invalid values.
`ME_request_invalid`	A request to a service is invalid. The request is impossible or makes no sense.
`ME_reference_invalid`	A reference to an object (handle, filename, pointer etc…) is invalid. The reference points to no object and cannot be resolved.
`ME_resource_shortage`	There is a shortage of a given resource (disk space, memory, processes, ports etc…). This resource is needed.
`ME_overflow`	A service or a given structure has reached its technnical maximum.
`ME_component_corrupt`	A component is corrupt, it cannot perform a given service.
`ME_component_interrupted`	A component has been interrupted and could not perform a given service.
`ME_service_not_available`	A service is not available, the service may be corrupted, interrupted, not loaded or not implemented.
`ME_wrong_access`	A service or a resource is being accessed but the calling service has not the right to do so.
`ME_unknown_type`	An problem occurred and the system doesn't what type it is. The error type could be unknown or lost in the recovery process.
`ME_no_error`	All is well.

access: any component may access this service.

`Explanation`

use: this service returns an explanation of the problem in human readable form

returns: the explanation (text)

access: any component may access this service.
note: other component should not use this service to analyze a problem. It should only be used to generate reports for the human operator…

Client
use: this services gives a reference of the client involved in the problem.
returns: the reference of the client.
access: any component may access this service.
note: non-Manager components should only use this service to execute Information Suite Services.

Server
use: this services gives a reference of the server involved in the problem.
returns: the reference of the server. The CPU and the OS are considered as servers.
access: any component may access this service.
note: non-Manager components should only use this service to execute Information Suite Services.

Cure
use: this services gives a reference of the cure involved in the problem.
returns: the reference of the cure.
access: any component may access this service.
note: non-Manager components should only use this service to execute Information Suite Services.

Time
use: this services gives a time stamp of the moment the problem occurred.
returns: the time of the problem.
access: any component may access this service.
note: non-Manager components should only use this service to execute Information Suite Services.

Recovery Function

A recovery Function, is a sequence of code that could solve certain problems. A recovery could solve different problems. A problem could be solved by different recovery functions.

Information Suite

`Function_Context`

use: this services gives the context the recovery function can work in. This can include, the actual processor architecture or the OS.

returns: the set of context the function can execute on.

access: any component may access this service.

note: non-Manager components should only use this service to execute Information Suite Services.

`ME_Function_Manager`

use: this services gives a reference of the Manager the recovery function is attached to.

returns: the reference of the Manager.

access: any component may access this service.

note: non-Manager components should only use this service to execute Information Suite Services.

High Level Suite

`ME_Estimate_Quality`

use: this services tries estimate the quality of a cure.

param.: the problem

returns: a number indicating the level of quality estimated

access: Manager components only.

note: this service is typically used by Managers components to build a cure.

`ME_Estimate_Reliability`

use: this services tries estimate the reliability of a cure.

param.: the problem

returns: a number indicating the level of reliability estimated

access: Manager components only.

note: this service is typically used by Managers components to build a cure.

`ME_Estimate_Cost`

use: this services tries estimate the cost of a cure.

param.: the problem

returns: a number indicating the level of cost estimated

access: Manager components only.

note: this service is typically used by Managers components to build a cure.

`ME_Try_Recovery_Function`

use: this services tries to solve a problem with the function.

param.: the problem.

returns: a number indicating the level of success encountered.

access: Manager components and cure object only.

note: this service is typically used by cure object when executing the Try Cure service.

Cure Object

A Cure object is built by combining a problem with a recovery function. It is typically build by the Manager.

Information Suite

ME_Function

use: this services gives a reference of the recovery function the recovery function is attached to.

returns: the reference of the Manager.

access: any component may access this service.

note: non-Manager components should only use this service to execute Information Suite Services.

ME_Quality

use: this services gives the estimated quality of the cure.

returns: the quality estimate.

access: any component may access this service.

note: non-Manager components should only use this service to execute Information Suite Services.

ME_Reliability

use: this services gives the estimated reliability of the cure.

returns: the reliability estimate.

access: any component may access this service.

note: non-Manager components should only use this service to execute Information Suite Services.

ME_Cost

use: this services gives the estimated cost of the cure.

returns: the cost estimate.

access: any component may access this service.

note: non-Manager components should only use this service to execute Information Suite Services.

High Level Suite

ME_Try_Cure

use: this services tries to apply the cure.

returns: an estimate of the success.

access: Managers components only.

note: this services uses the ME_Try_Recovery_Function service.

Management-Enhanced Component Object (MEC)

The MEC Object is the basic building block of the ME system. All subsequent server components are derived from it.

Information Suite

`ME_Availability`

use: this services estimates of the general availbility of the component. This means the load of service the component could accept.

returns: an estimate of the availability.

access: any component may access this service.

note: non-Manager components should only use this service to execute Information Suite Services.

`ME_Level`

use: this service gives the ME-level the component operates.

returns: the ME-level

access: any component may access this service.

note: non-Manager components should only use this service to execute Information Suite Services.

`ME_Manager`

use: this service gives the MEC's Manager

returns: a reference to the Manager.

access: any component may access this service.

note: non-Manager components should only use this service to execute Information Suite Services.

`ME_Stateless`

use: this service tells if the Managers is stateless or not.

returns: a Boolean

access: any component may access this service.

note: non-Manager components should only use this service to execute Information Suite Services.

Low-Level Suite

`ME_reject`

use: this message the service to reject a given transaction.

param: a transaction *[to be defined]* reference.

access: The Manager Service or an encompassing object may access this service.

`ME_retry`

use: this message the service to reject a given transaction.

param: a transaction *[to be defined]* reference.

access: The Manager Service or an encompassing object may access this service.

`ME_enable`

use: this message enables the Management of the component.

access: Manager components and recovery functions only.

`ME_disable`

use: this message disables the Management of the component.

access: Manager components and recovery functions only.

`ME_set_Manager`

use: this service attaches a MEC to a Manager.

param.: reference of the Manager

access: MEC and Manager Access Interface.

MEUF Suite

This suite serves to dispatch and to treat local problems, it is meant for internal use by the MEC only, most signals and messages will be transmitted to the Manager's MEUF suite.

ME_assert
use: checks a given condition, if the condition fails, it generates a Internal Error Problem object and tries to dispatch it to the Manager.
params: the assertion
returns: -
access: MEC

ME_precondition
use: checks a given condition, if the condition fails, it generates a Interface Error Problem object and tries to dispatch it to the Manager.
params: the precondition
returns: -
access: MEC

ME_Dispatch_Problem
use: dispatch a problem by finding a cure and trying it.
params: the problem
returns: number indicating the rate of success
access: MEC

ME_Register_MEC
use: attaches a MEC to a Manager.
params: the MEC
access: MEC

ME_Unregister_MEC
use: attaches a MEC to a Manager.
params: the MEC
access: MEC

ME_Register_Recovery_Function
use: attaches a recovery function to a Manager.
params: the recovery function
access: MEC

Assertions & Preconditions

There should be not one but many assertion and precondition services, each data types should have its preconditions and assertion services. This way error detection could use data specific schemes.

Each descendant of the MEC class should implement its own extension to the ME_assert and ME_precondition service, so as to treat all data type it offers or uses…

Manager Object

The Manager object servers as model for all derived Managers components. It derives from the MEC service. Thus a Manager is itself a MEC, it can also be management, perhaps by itself or a super-Manager…

High-Level Suite

ME_Managers

use: this services returns a reference to the Manager Collection
returns: a MECO
access: any component may access this service.
note: Manager and Manager Access Interface.

ME_Recovery_Functions

use: this services returns a reference to the Recovery Function Collection
returns: a MECO
access: any component may access this service.
note: Manager and Manager Access Interface.

ME_Best_Cure

use: this services finds the best cure available to the Manager. This cure may be attached directly to the Manager or a sub-Manager.
params: the main criteria (cost, reliability or quality), the problem.
returns: the best cure
access: Manager and Manager Access Interface.

Low Level Suite

ME_load

use: this services make the Manager load itself completely. The Manager may load and activate its attached recovery functions.
access: Manager and Manager Access Interface.

ME_unload

use: this services make the Manager unload itself completely. The Manager may unload and deactivate its attached recovery functions.
access: Manager and Manager Access Interface.

MEUF Suite

This suite is the manager's counterpart to the MEC's MEUF suite. Contrary to the MEC's MEUF services, Managers MEUF services can be accessed from outside, that is from a MEC

C) Source Code

The source code for both the C++ prototype and the Java implementation is available at the following URL:

`http://cuiwww.unige.ch/~wiesmann/MEC/`