

Enfinity: The Good, the Bad and the Ugly

The Benefits and Limitations of Enfinity. Experiences with xWine.

Roland Turner

Gregor Klinke

Enfinity: The Good, the Bad and the Ugly: The Benefits and Limitations of Enfinity. Experiences with xWine.

by Roland Turner and Gregor Klinke

This paper summarises a number of experiences made with Intershop's Enfinity platform and shows some of the problems (generic and specific) to be encountered working with this platform.

THIS DOCUMENT IS A DRAFT

THIS DOCUMENT IS NOT INTENDED FOR DISTRIBUTION OUTSIDE OF INTEGRA

Table of Contents

| | |
|---|----|
| Enfinity: The Good, the Bad and the Ugly..... | |
| Object of discussion..... | 5 |
| 1.The Good: What Enfinity is Built to do..... | |
| Provides a straight-forward web storefront for selling fixed-price, durable, non- customised goods offered by a single vendor directly to consumers..... | 1 |
| Provides high scalability for the readonly-catalog case..... | 1 |
| Provides a complete, working store..... | 1 |
| Builds on some Industry Standards..... | 1 |
| 2.The Bad: Limiting Choices in Enfinity's Design..... | |
| Split into two separate servers..... | 2 |
| Unmotivated use of different standards..... | 2 |
| Unjustified claims to be standard conformant..... | 3 |
| “Use a bicycle as a moonrocket”..... | 3 |
| Basic architecture of customizable entity beans..... | 3 |
| Flowcharts do not foster reusability..... | 4 |
| Precompiling template language to JSP introduces a performance hit and discourages maintainability..... | 5 |
| Proprietary template language leads to vendor lockin..... | 6 |
| 3.The Ugly: Serious Implementation Problems..... | |
| Documentation..... | 7 |
| Enfinity is not a framework..... | 7 |
| Template architecture flaws..... | 7 |
| Transaction flaws..... | 7 |
| Instability..... | 8 |
| Oracle and Enfinity timezone integration..... | 8 |
| Integration of own EJBS..... | 8 |
| Missing exception handling..... | 9 |
| Development tools..... | 9 |
| Debugging interfaces are maimed..... | 9 |
| Visual Pipeline implementation is flawed..... | 10 |
| 4.Conclusion..... | |

Enfinity: The Good, the Bad and the Ugly

This document discusses our impressions of Enfinity developed while using it on large scale projects over the last year, notably xWine.

(With apologies to Sergio Leone and Clint Eastwood.)

Object of discussion

The object of this discussion is the Intershop's flagship product, Enfinity. Most of the discussion relates to Enfinity version 2, but some of the problems described are intrinsic to Enfinity's architecture.

Details of the project system:

- Enfinity 2.1.0.0.5188, NT version. Database: Oracle 8.1.6, NT and Solaris versions.

Chapter 1. The Good: What Enfinity is Built to do.

This chapter describes what Enfinity is good at.

Provides a straight-forward web storefront for selling fixed-price, durable, non-customised goods offered by a single vendor directly to consumers.

This is the classic web storefront application, and Enfinity appears to do this well.

Provides high scalability for the readonly-catalog case

In a B2C environment, one important characteristic is the very high browsing to transacting ratio. In extreme cases this ratio can be in the tens or even hundreds to one. To this end, Enfinity is architected to have an arbitrarily large number of catalog browsing servers, thus allowing a minimisation of the load on a single transaction server. (In Enfinity version 1, it was not possible to have multiple servers altering data, so multiple transaction servers were not an option.) In addition to the eCS/eTS split, each eCS can have a staging double. In this configuration, catalog changes can be preloaded onto the staging double for each eCS, then all of the active and staging eCSs have their roles switched, the changes can then be loaded on the now-staging-double server at leisure. This ensures that all changes get rolled out to all servers at once, thus reducing situations where a product's price will appear to change back and forth as the customer browses. The corollary is, of course, that it must be acceptable for the entire catalog to have a controlled simultaneous release (as distinct from the continual, un-coordinated updates to the catalog that are likely to occur in a B2B environment).

Provides a complete, working store.

It includes a complete operational demo that could be customised to become a live site for a business wishing to sell goods of this type directly to consumers. (Goods with continually variable pricing, pricing altered at will by multiple vendors in the same marketplace, goods which are perishable or distressed or goods that are customised will require site-specific customisations.)

Builds on some Industry Standards

Intershop has resisted the temptation to reinvent absolutely everything. In many places, widely used industry standards are in use. This does facilitate some degree of integration and extension.

Chapter 2. The Bad: Limiting Choices in Enfinity's Design

This sections summarises more general concerns about the Enfinity platform, architecture, and implementation.

Split into two separate servers

The basic idea of Intershop to split the server into two independant halves is one of the major drawbacks for complex sites. The idea behind the separation (to have a fast and scalable catalog server [eCS] and a separate transaction server [eTS]) was only good for the Enfinity 1 platform. In this version on readonly servers could be scaled, and therefore scalability meant to have only the non-transactional data (i.e. the catalog) scaled. The transaction server was therefore not scalable and was/is a severe bottleneck in Enfinity 1 platforms (as independant performance test by Sun has shown). In essence, Intershop has built two seperate, but interconnected, systems with some degree of overlap. This introduces all of the problems that splitting functionality across different systems brings in any other context. In xWine's case, this split caused such severe problems that the development team undertook a complete migration off the eCS (with the corresponding reimplementaion of the product catalog that this implies) in order to overcome them.

A further consideration is the impact that this has on personalisaion. Whilst it is always possible to add session and user support to the eCS, doing so immediately eliminates Enfinity's scalability (session data has to be written somewhere), at which point the eCS/eTS split becomes an unneccesary inconvenience. In a B2C environment, customisation is desirable, in a B2B environment, it's critical (customised pricing displayed during browsing, not just during buying, different functions offered to different roles, etc.). It appears that any B2B application built on Enfinity is going to have to fight Enfinity's architecture.

Given that Enfinity version 2 (apparently) provides support for multiple eTSs anyway, it is possible that the entire argument is now moot and that the continuation of this split is just the weight of history. On the other hand, perhaps the multiple eTS support doesn't scale far enough.

Unmotivated use of different standards

The Enfinity platform is a good example for the unmotivated use of a large number of standards and tools. Nearly everything is included into the system somewhere and somehow, but does not contribute to the system consistency or functionality. For example "CORBA" is used to connect the Enfinity management console [eMC] to the administration server. The server managemenet console (similar to the eMC in its impact for maintaining the system) uses standard HTML and XML protocols to connect. Standards like EJB or XML are used without the possibility to use the real strength of these standard (e.g. it is not possible to use session beans and de-

scriptive bean security – an integral part of the EJB specification).

Unjustified claims to be standard conformant

Yes, it is usual that software does claim standard conformity. But Intershop is a bad example for this. To quote from their Enfinity home page “Die offene Architektur von Intershop Enfinity verwendet modernste Technologien auf der Grundlage von führenden Standards wie XML, WAP und J2EE, um eine optimale Interoperabilität, Erweiterbarkeit und Integrationsfähigkeit des Systems sicherzustellen.” [Rough translation: “The open architecture of Intershop Enfinity uses most modern technologies on base of leading standards like XML, WAP and J2EE, to assure an optimal inter operability, extensibility and possibility of integration of the system.”] It is not very difficult to show that Enfinity is in neither way a J2EE platform or even tries to be. Apart from Java, entity EJBs, and a little bit XML here and there, there is nothing about J2EE in Enfinity.

Not standard confirming software leads always to not reusable code. Software written for Enfinity can't easily (or at all) be ported to other application servers.

“Use a bicycle as a moonrocket”

Enfinity is – seen from its basic architecture – a shop solution. It is designed to be a plug-and-play eCommerce solution for simply shop sites. Trying to build with it complex B2B plattformen, ERP-frontends, or multivendor marketplaces is a very questionable procedure, since the work which has to be done to re-implement order, basket or product maintenance stuff is larger than implementing this stuff on a clean (pure) application server platform.

Basic architecture of customizable entity beans

An essential problem that arises when attempting to reuse objects is that, frequently, the objects to be reused do not exactly match the objects required by the problem that is being solved. Perhaps the most common occurrence of this (and the simplest to solve) is the need to add attributes to existing objects, e.g. fields to describe particular information common to the type of product being sold. Rather than providing a means for new business objects, attributes or relationships to be added seamlessly at will, Enfinity provides two different means, neither of which works in all cases and neither of which is particularly clean. The first is to add custom attributes, the second to add "own EJBs".

To handle the addition of attributes, Enfinity provides a single object type which has a name, a value and an owner object. This is untidy, but acceptable for adding simple fields for humans to manipulate and see. The moment that they become part of the application logic (used to wire up custom relationships, queried on, used to store entire objects, etc.), a similar problem to that with the Visual Pipeline Man-

ager arises: people will tend to take the mechanism far beyond its practical use because there is no clean transition to a more sophisticated mechanism. Instead, each small change is added and the resulting data model incrementally becomes unintelligible.

Here is what tends to occur: Initially no-one switches to a more powerful mechanism because the requirement is simple enough to use the simpler mechanism. Then as more and more attributes are added the effort required to move everything to a more powerful mechanism is so large that no-one has time. Next, the structure is so complex that no-one can understand all of it, so even if the time is made available, fear of destabilising the existing code is great enough to prevent movement from occurring. Finally, the structure is so complex that no-one can understand any of it, so it has to be thrown away and rebuilt from scratch. Typically this might take years, but in an environment where getting into the mess is made so easy, and getting out of it made so difficult, it can happen in months. Indeed, in xWine's case it happened more than once during the initial implementation phase (four months) of the project.

The addition of own EJBs is described in a later section of this document.

Flowcharts do not foster reusability

Enfinity's basic mechanism for specifying high-level functionality is not, as one might expect, a modern, robust, type-safe, exception-safe, object-oriented mechanism, but rather a very attractive GUI interface for creating and editing flowcharts. Unlike the discarded flowcharting tools from decades past, Enfinity doesn't merely use flowcharts for modelling, it actually executes them at runtime. Naturally, to give them a more contemporary flavour they are not called flowcharts but 'Visual Pipelines'. However, with start nodes, end nodes, decision nodes, processing nodes (pipelets), jump and call nodes and little else (two special interaction nodes to handle the disconnectedness of HTTP and a processing node placeholder for not-yet-implemented processing nodes), the resemblance is striking. Unsurprisingly, Visual Pipelines inherit all of the limitations of flowcharts, most importantly that they do not handle complexity well and they do not scale. As if modelling behaviour without abstraction, polymorphism, exception handling and type-safety were not enough, the only means of exchanging data that is made available to the flowchart modeller is a global variable space called the Pipeline Dictionary. Finally, the Pipeline Dictionary is not, in itself, type-safe. Typically, when global variables are being used, some type is declared for a variable of a given name and only values of that type can be associated with that name. Enfinity doesn't even provide this facility, the modeller is required to assume that there is no way that a value of the wrong type can land in a variable with a given name. These features combined have for decades given rise to software that is utterly unmaintainable, reuse is not even an issue.

Another way to describe what is wrong here is that Visual Pipelines actually undo the data/functionality coupling that object-oriented approaches have sought to es-

establish. Visual Pipeline actually go one step further by forcing two completely different means of describing functionality: one is to create flowcharts that use existing processing nodes, the other is to create new processing nodes in Java. There is no obvious way to decide which should go where. The result tends to be unmaintainable, non-robust chaos.

It may be that the Visual Pipeline Manager is a way for non-programmers (business analysts perhaps?) to assemble functionality created by programmers. Programming without programming. In practice, the people who work with it are programmers of one sort or another and will tend to get it to do more complex things than it is good for. In fact, no-one has ever come up with a workable means to separate 'business logic' from 'program logic', other than by drawing it out of and refactoring existing code. To assume that Java programmers will tend to create exactly the chunks of functionality that business analysts need and that business analysts will go anywhere near the tool and spend the time working with the programmers to get workable solutions out appears naive.

One final comment is worth making here. As with most other visual notations, flowcharts are useful for complete ideas that can be modelled in seven (plus or minus two) nodes. If it were the case that all (or even the vast majority) Visual Pipelines were of this size, then it could reasonably be claimed that the tendency to make unmaintainable models was not asserting itself here. In fact, many of Infinity's own flowcharts have upwards of twenty nodes, some of the flowcharts created for xWine have upwards of 50.

Precompiling template language to JSP introduces a performance hit and discourages maintainability.

Some years ago, Sun published JSP as a standard template language for use in the Java environment. At the time, it was assumed that a hard-coded template class with a lot of System.out.println(s) in it would execute faster than a template processor which parsed the template at runtime. As a result, the JSP processor works by precompiling the JSP source to Java source code, then invoking a Java compiler to produce bytecode (which is cached for future use), then loading and running the result. Naturally all of this means that loading a template for the first time is very slow, which is very wasteful during development, but the intention was to achieve maximum runtime speed, so the design was acceptable. This performance trade-off may have been reasonable in an interpreted or JIT environment, but with virtual machines now performing adaptive recompilation (e.g. Sun's HotSpot), the speed of execution of a piece of code is a function of its degree of optimisation, in turn a function of the frequency with which it is executed. Therefore, having the template execution code spread in near identical copies in hundreds of places (once per template) is likely to perform worse than having a single copy of a more generic processor being executed hundreds of times more often.

The result of all of this is that a template engine that depends upon precompilation to Java and then Java compilation to bytecode is likely to perform worse both in de-

2. The Bad: Limiting Choices in Enfinity's Design

velopment and live deployment than one which has a single generic processor (with some limited tweaking). Unfortunately, because JSP was designed with Java in mind, it allows the embedding of arbitrary Java source code in templates. The consequence is that the templates almost have to be passed through a Java compiler.

This is all relevant to Enfinity because its template language (ISML) is defined and implemented as a precompiler to JSP. Not only does this introduce a large performance hit during development and a mild performance hit in live deployment, it also encourages the embedding of Java code into templates which further blurs the line between application and presentation logic and makes maintenance of the templates and of the underlying code far more difficult.

Proprietary template language leads to vendor lockin.

Even worse than the performance and maintainability problems related to precompiling to JSP, Enfinity's template language is proprietary to Intershop. This means that if a website is to be moved off Enfinity to, for example, J2EE, then every single template will need to be rewritten. It is possible that this could be automated to some extent, but it's not likely to be trivial. The size and expense of this exercise may well keep sites on Enfinity which would otherwise be far better off moving to J2EE. This is not bad for Intershop, but it is a problem for Integra and its customers.

The above comment begs the question 'so what to use'? In fact there are any number of templating engines available in open-source. Tea and webmacro are two examples that spring to mind, there are numerous others. The function of a template processor is in fact rather simple, this is not a strategic value-add component.

Chapter 3. The Ugly: Serious Implementation Problems

This section lists a number of found bugs, inconsistencies, and other problems, which leads to an unnecessary complication and therefore cost increase in application development with Enfinity.

Documentation

The documentation is sparse and misses to explain the most interesting stuff and is sometimes even wrong.

Enfinity is not a framework

Enfinity seems not to be designed to be a “framework”, but to be a customizable platform. E.g. it is not possible (in standard Enfinity) to set up schedulers in the source code or during initialization. Scheduler's has to be created manually using the eMC.

Template architecture flaws

The architecture of the template processing unit is fatally flawed. The solution Intershop has chosen is a wrapper for the standard Java Servlet engine, which means, that no templates can be processed from the backend, they can only be processed as a result of frontend activity (e.g. an HTTP request). There is only one exception (which is a clear hack) with the email sending mechanism, which allows to use templates for emails. But it is not possible to use templates e.g. to create invoices (as pdfs), parsed files, etc. – at least not without knowing Enfinity internals.

Transaction flaws

Enfinity's transaction handling is unsound and not at all obvious. What appears to be going on (although this is not clear from the documentation) is that Enfinity places Oracle into auto-commit mode, meaning that every database update is processed as soon as it is executed, rather than upon execution of a database commit. This in turn means that bugs in any part of the system can cause the database to be placed in an inconsistent state. Specifically, if a pipeline performs several related updates (e.g. adds a line to an order and decreases available inventory) that must be performed atomically and an `RuntimeException` (e.g. a `NullPointerException`) is thrown midway through the operation, the template processor will generate the dreaded George (system error) template, but the operation will have been half performed. In this particular instance what is likely to occur is that the user will retry the operation, note that his order line had been saved anyway and checkout his order. This is fine, except that the available inventory information will not have been updated. If this customer in fact purchased the last available item(s) of a particular

type, the system will not notice and will continue to offer the unavailable item to other customers and perhaps even accept an order for it. Clearly there are ways to limit the damage that this type of error does, but atomicity has been readily available in database servers for more than a decade and is best handled there. That Enfinity's default mode of operation disables this fundamental database facility is cause for concern. (A pipeline can have a transaction added to it, but this is not the default behaviour. Additionally, it appears to be done in such a way as to place each pipelet into its own transaction which does not solve the atomicity problem. Once again, the documentation appears to be inadequate.)

A complicating factor is Enfinity's interaction with a PowerTier bug. When an EJB is deleted in a transaction context, PowerTier notes the deletion but occasionally (not reliably - so it's hard to debug) throws a `RemoteException` if a query is executed over related data immediately afterwards. This is a known bug and the PowerTier documentation provides a workaround, but Enfinity appears not to implement the workaround (and a developer building on Enfinity is not able to), so it is the case that sometimes, at random, object deletion will cause a subsequent query operation to fail.

Instability

The PowerTier version included with Enfinity 2 has a severe bug: Trying to store a Java `null`-pointer reference to an entity EJB (to a slot which allows to be null!) crashes the Java virtual machine. Until NT finishes writing its Dr. Watson log and the supervisory program restarts the virtual machine (and the latter completes its startup), the website is down, and there is nothing that can be done to prevent this.

Oracle and Enfinity timezone integration

Oracle and Enfinity use different timezones. Oracle seems to use the timezone it is configured to (and by default it is configured to use UTC), whereas Enfinity uses the system's timezone. These timezones are transformed correctly by the powertier layer, but not in all places by direct JDBC connections.

Integration of own EJBS

(This is the second of the two extension mechanisms mentioned earlier.)

It is possible to integrate new EJBS, but due to lack of documentation and any support information in the standard installation, user defined EJBS are not covered by the cache synchronization layer build by Intershop in Enfinity 2. All EJBS seem to need at least an OCA attribute, otherwise cache synchronization does not work, and therefore the scalability (one of the real advantages of Enfinity 2) is not usable.

Further, whilst PowerTier is capable of maintaining relationships between objects,

it can only do so for objects defined within a single 'project' file. As no PowerTier project file is provided for the standard Enfinity objects (indeed, it seems that Intershop uses another tool entirely to build its EJBs), having relationships between own EJBs and Enfinity objects maintained automatically is not possible. As soon as you define own EJBs, you take on the responsibility for maintaining relationships.

Missing exception handling

Java is normally used to assure stability and safety. Therefore exceptions are one of the features of this platform, i.e. the user is always told about breaking pieces of source code, not correct compiling or running code, and this even during runtime (instead of simply crashing the application like normal C or C++ applications do). But Intershop chose to catch a large number of exceptions and *not* to tell the user that this exception has occurred. This could lead (and leads!) to a vast overhead in bug tracking and testing.

Development tools

The development tools for Enfinity are rather basic. The most part of the compile, startup and configuration tools are written as undocumented perl scripts, which disallows extension and customization. This is not a problem if an application is developed by one (or at maximum two) developers only. If a code repository and automatic building environments with controlled and repeatable building and deploying is required (e.g. in large developer groups), this tools needs a lot of (time consuming and cost intensive) wrappers and work arounds in second level tools (e.g. make-files or ant scripts).

Most of the tools are additionally not “intelligent” enough, to provide only a compile on demand, which doubles or triples compile time on each code change or rework. Development tools supported such features increase development speed normally rapidly.

Additionally, support for dynamically reloading class files did not appear to be present (or at least not operational). Much development time was spent shutting down and restarting Enfinity in order for a recompiled class file to be loaded.

Debugging interfaces are maimed.

For reasons which Intershop does not explain, standard Java debugging mechanisms do not work correctly with Enfinity. Certainly a non-optimising virtual machine can be used with debugging access turned on, but attaching a debugger and attempting to work with Enfinity objects causes the debugger to abort itself. Intershop's suggested solution is to use Borland's debugger which requires the (runtime) linking of Borland .DLLs to the virtual machine. If you have deployed on Solaris and have encountered a bug which only appears in your deployment environment

(e.g. only during load conditions which you are unable to simulate), then you are simply not going to be able to debug it. This may not impress your customer.

Visual Pipeline implementation is flawed

Setting aside, for a moment, the problems described earlier with the ideas behind the Visual Pipeline Manager, Intershop has made serious errors implementing its own vision.

The defeating of the transaction mechanism and the silent eating of exceptions has already been described. The flowchart execution mechanism would have been the place to handle this properly, to start a transaction upon receiving an HTTP request and to commit it upon normal exit or roll it back if a `RuntimeException` was thrown.

Another problem is apparent in the handling of exceptions thrown during pipelet configuration. During startup, Enfinty instantiates each flowchart and its constituent processing nodes. A means to configure the processing nodes is provided and the `init()` method called on each processing node (pipelet) to do this. If an exception is thrown at this point, the instantiation of the pipeline is aborted. This is not entirely unreasonable, a pipeline that aborted its configuration phase should not be asked handle a service request. However, the Visual Pipeline Manager is only able to edit pipelines that were successfully instantiated. So, if the user using the Visual Pipeline Manager made an error in the configuration of a processing node in his flowchart which caused an instantiation problem (e.g. a mandatory piece of configuration information was not provided for a processing node), the pipeline won't instantiate, the Visual Pipeline Manager won't load it and the error cannot be fixed. Oops. The easiest way out of this mess (it happened during xWine) was to manually edit the pipeline description file; fortunately it's a fairly simple XML file.

Chapter 4. Conclusion

Enfinity remains a potentially interesting product for customers who want to do exactly what Enfinity already does, or something very similar such that only a little customisation.

What does "a little" mean? Clearly the amount of design, content and frontend work is somewhat independent of this (it depends upon how fancy an appearance the customer desires), but to state a somewhat arbitrary baseline for the backend: if two backend developers need more than four weeks to customise something, then they aren't customising it, they are building substantial new functionality. In this case, it is worth looking seriously at building on a generic platform (e.g. J2EE) and integrating existing open-source or other third-party components to fill any perceived gaps in functionality is likely to provide a cleaner, faster, more maintainable solution, sooner.