WHITE PAPER

Elemental Development

A Unique Paradigm for Developing Reporting Applications

Abstract

The purpose of this paper is to describe Logi Analytics' unique and innovative paradigm for report application development — a concept that we have termed Elemental Development (ED). This paper is for report developers who would like to learn more about the technical details and benefits of this unique approach. ED is a development methodology that emphasizes isolation of underlying technology from an application's logic and presentation. The ED methodology introduces the idea of creating a non-procedural language, or dictionary that can be used to describe a class of software application. In this paper, we describe a specific technique for creating applications using our ED methodology. We also show how ED offers savings in time and money including how ED produces applications that are highly maintainable with a code base that is highly reusable.

http://www.LogiAnalytics.com

info@LogiAnalytics.com

Contents

Executive Summary

Logi Analytics, Inc. has developed a unique and innovative paradigm for report application development — a concept that we have termed Elemental Development (ED). Implementing an ED-centered environment is based on an extremely high level, re-usable XML-based language that fits specific business intelligence needs. And, this XML-based language can be thought of as a dictionary for application development.

Traditional high-level languages such as Java or C# focus on providing a flexible and robust framework for creating applications of almost any type. An ED language, however, is formed for a specific type, or class, of application. Once built, it can be reused to rapidly develop similar applications with a huge savings in development and maintenance time and costs. This is because ED standardizes and simplifies the development process.

Benefits for Report Developers

The ED approach offers the following valuable benefits for developers.

- **Streamlined Development** – Our approach takes advantage of the self-documenting, intuitive and descriptive nature of Logi Analytics elements. For example, if you see an Email element in a report, you quickly know that the report form sends email. If you see a User Role element, you know that role-based security is implemented, and you can determine how that security is set up by looking at the attributes that describe the User Role element.

  Also, less actual coding is required. Using wizards and drag-and-drop, developers can easily build complex reports (for example, with drill-down, data grouping and filtering) without having to build complex SQL queries, subroutine calls or advanced command constructs.

- **Increased Productivity and Faster Deployment –** You can accomplish report development in a matter of hours instead of the weeks and months that may be required of other development tools due to:

  - Ease of use and reusability of XML elements
  - Logical and hierarchical layout of elements, which makes it easy to understand and manage layout and functionality of larger reports
  - Ability to change report layouts or functionality ‘on the fly’—just by modifying elements and attributes in the report definition
  - Based on well-known, non-proprietary open technologies and standards like XML, .NET, SOAP, Web Services, and so on

- **Scalability** – Our approach leverages multi-tier development and multi-tier deployment inherent in Web-based applications, which is by nature, more scalable than license-based or traditional development.

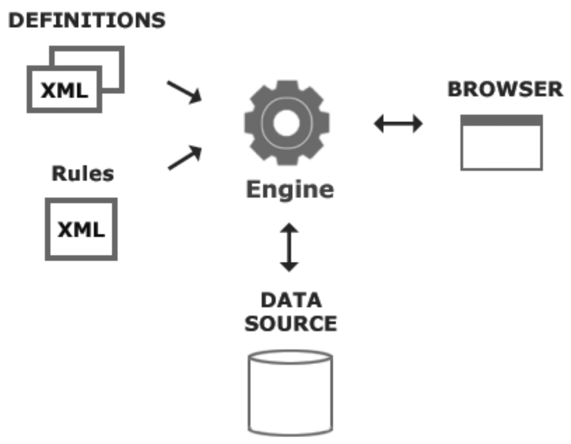
Introduction to ED

Logi Analytics' innovative development environment paradigm, Elemental Development (ED), leverages XML for ultimate flexibility and extensibility, with a Logi Analytics element being the basic unit of development. Our element-based approach separates data handling, report development and presentation, breaking them down into elements with attributes that can be easily configured.

- Elements include various types of functionality and presentation
- Attributes let developers customize that particular functionality or presentation

The ED methodology introduces the idea of creating a non-procedural language, or dictionary that can be used to describe a class of software application. In this paper, we will describe a specific technique for creating the basis for applications using our ED methodology.

Developers can create several ED languages for different purposes and then maintain these languages separately, or together, as they see fit. With strict adherence to backward compatibility rules, maintenance at the code level can be achieved without affecting the ED language and changes to the application can be made at the ED language level without ever touching code.



At a minimum, an ED architecture consists of a modeling language and an engine. At run-time, the engine interprets one or more application definition files written in the modeling language. For a complete implementation there should also be a support library of productivity tools and a baseline inventory of application and business-area templates – written in the modeling language.

Traditional Problems

Traditional development is slow, repetitive and monotonous. The methodologies that the majority of the software world has adopted do not incorporate good strategies for maximizing code reuse or code sharing. Code is either proprietary or it is "spaghetti code" which has been thrown together by a large development team. Sometimes technical architecture can be passed from one project to the next, but this architecture usually deals with a very small subset of the overall application development. Code is rarely reused effectively.

There are many solutions to this problem. Some involve finding better ways to maintain and distribute code fragments. Other strategies involve packaging these code snippets into reusable objects and then moving these objects around as they are needed. No solution is perfect, but few organizations even have a solution. Many projects simply reinvent the wheel every time it is needed which wastes time and money.

ED provides a solution to this problem. It establishes a framework through which developers and organizations can organize their development in a highly logical fashion that will not only save time up-front, but will also greatly reduce future development and maintenance costs.

ED Supports Change

As with knowledge, technology has and continues to be dominated by change, and ED is a concept that is modeled to support the fast-paced advancement of computer science.

Some might ask why other computer programming languages fail to support change. The answer is that they represent the evolution of languages – each is a next step towards better languages, however they do not necessarily support underlying changes in technology without major rewriting. Other than the growing knowledge bases of those that use the language, there is nothing inherent that improves from one implementation to the next. ED provides a framework, which allows developers to easily build knowledge into a reusable architecture and then grow and expand that architecture with each implementation.

As an ED modeling language serves more and more applications, it will become more complete and will require less and less modification between implementations. If the developer adopts a strict policy of not removing elements or attributes of elements, then the language can retain its backward compatibility throughout its evolution.

Implementing ED

ED compartmentalizes a great deal of functionality into each function call while allowing great configurability for that code through parameter attributes. Furthermore, ED enforces strict validation rules for nesting, embedding and function placement. This enforcement not only gives rise to simplification during development, but also allows greater optimization and speed at runtime. Its primary focus is to:

- Minimize redundancy by isolating any reused code into packages
- Making all packages highly configurable via parameter attributes
- Establish parent / child relationships between all unique and reusable code
- Strictly enforce all parent / child relationships

Because of the validation, parsing and parent/child structure of ED, XML is an ideal technology to implement the language. What results is a highly logical structure, which can be used to define almost any application at almost any level.

Implementation of ED

The ED solution is akin to traditional best practice application development, but it adds some new and unique twists. To create an ED:

1. Identify the type of application being developed
2. Break the application down into its main (possible) elements. For example, a website can be broken down into header, menu and body.
3. Break each individual element down into elements. For example, let’s say a header can be broken down into style sheet, logo, text, buttons, text, links and graphics.
4. Keep breaking down elements until you can *basically* define each element using the underlying language
5. If an element appears more than once or if it has children, it remains an element; otherwise, it becomes an attribute of its parent element.
6. Elements are children of their parent elements
7. Every element except the root element must have a parent element

Notice an interesting distinction, which sets ED apart from other development methodologies — the focus is not in developing architecture; nor is it in separating business rules from presentation code. And as we break down every element into more elements, these distinctions become even more obvious.

The next steps are more complicated, but basically entail:

1. Defining XML rules files for your ED based on your definition
2. Building your application based on the XML rules file
3. Building an engine to interpret files conforming to the XML file

Why this order? The first step must always be to define our rules file, but the second step could either be to build the application or to build the engine to run the application. We recommend you build the application first. This not only allows you the opportunity to make sure you have defined all the necessary elements needed to build your application, but also gives you a working application to test your engine as it is developed.

XML Rules File

To begin defining our XML ED rules let's look at a simple XML ED rules file. This file defines a very simple application called DataLists.

```
<Elements>
  <Element Name='DataLists' Label='DataLists'>
    <Attributes>
      <Attribute Name='Color' Status='Required' Label='Color' ValueType='String' >
    </Attributes>
    <ChildElements>
      <ChildElement Name='DataList' Status='Optional' />
    </ChildElements>
  </Element>




  <Element Name='DataList' Label='DataList'  >
    <Attributes>
      <Attribute Name='DisplayField' Status='Required' Label='DisplayField'
ValueType='String' />
    </Attributes>
    <ChildElements>
      <ChildElement Name='SQL' Status='Required' />
    </ChildElements>
  </Element>

  <Element Name='SQL' Label='SQL' >
    <Attributes>
      <Attribute Name='Source' Status='Required' Label='Source' ValueType='String' />
    </Attributes>
  </Element>
</Elements>
```

DataLists has one attribute named Color, which can be any text string. DataLists may have a single child, DataList. So, why does DataLists *(plural)* contain DataList *(singular)* as a child element? In this case, it is to logically organize lists of various elements. The DataLists element can contain multiple DataList elements and set attributes of all of its children DataLists.

Next, look at the DataList element, which has the DisplayField attribute. This attribute sets a field that will be displayed and SQL database call we will make to get data. The DataList element has a single required child, SQL, which defines a connection string and SQL call to a database. By defining various attributes, a developer could easily pull data from a database table or tables. This data would then automatically be used to populate its parent element, the DataList element.

This example defines a very simple data list, but it is easy to see the power of the methodology. As an application grows, so does the ED that supports it. The ED becomes available for all future applications, allowing an organization to build standardized, organized, highly maintainable and highly reusable applications.

XML Definition File

Now, it is time to start building the application based on the rules files defined above. The application is going to be two DataLists, one that displays a client list, and the other, which displays recent orders.

```
<?xml version="1.0" encoding="ISO8859-1" ?>
<DataLists>
    <DataList DisplayField='Full_Name'>
        <SQL Query='SELECT * FROM Clients' Source='somedatabase' Color='Green'/>
    </DataList>
    <DataList DisplayField='Order_Name'>
        <SQL Query='SELECT * FROM Orders' Source='anotherdatabase' Color='Red'/>
    </DataList>
</DataLists>
```

So that is it! Using the rules, an application definition that fulfills our requirements has been built.

XML Interpretive Engine

Once the XML rules file and the application have been created, development of the engine can begin. The engine is obviously the most complicated piece and will require the most time. However you will find that development will often be much easier than it might be traditionally simply because of the highly defined structure of your rules. Depending on what language you will develop the engine in, the process will vary greatly, but it should be written using an established application server technology such as .NET or Java EE.

The rules file we have defined above is very simple, and does not go into any detail about other aspects of the application such as formatting (other than color). This is absolutely fine. Your rules can either be small and provide for minimal configuration of other options, or they can be expanded to incorporate formatting, security, or just about any other aspect of an application that you can think of. If

you keep your rules small, then most of these aspects of the application will be defined in the engine and will be unchangeable. As you expand your rules to allow for more flexibility, the engine will also become more flexible. In this case, the rules set is simple, so our engine will have to do most of the work to define the details of our application.

For this example, we have built a modest ED engine in ASP.NET using C#. This could have just as easily built in Cold Fusion, Java, Perl, PHP or any other language for that matter. We used C# because it offers a relatively simple example.

```
<%@ Page Language="C#" %>
<%@ import Namespace="System.Xml" %>
<script runat="server">
    protected string myDataList = "";
    private void Page_load(Object sender, EventArgs e) {
        XmlTextReader xmlReader = new XmlTextReader("C:\\Documents and Settings\\allen\\My
Documents\\AppDef.xml");
        xmlReader.WhitespaceHandling = WhitespaceHandling.None;
        xmlReader.MoveToContent();
        string displayField = "";
        while(xmlReader.Read()) {
            if(xmlReader.NodeType.ToString() == "Element") {
                switch(xmlReader.Name) {
                    case "DataList":
                        displayField = xmlReader["DisplayField"];
                        break;
                    case "SQL":
                        string fieldValue = getData(xmlReader["Source"], xmlReader["Query"],
                                                    displayField);
                        myDataList += "<font color=" + xmlReader["Color"] + ">" +
                                      fieldValue + "</font><br>";
                        break;
                    default:
                        break;
                }
            }
        }
    }
    private string getData(string connString, string sourceSql, string displayField) {
        return DateTime.Now.ToString();
    }
</script>
<html>
    <head>
        <title>DataLists Application</title>
    </head>
    <body>
        <form runat="server">
            <h1>ED DataList Example</h1>
            <% =myDataList %>
        </form>
    </body>
</html>
```

That is it! Okay, so this is a pretty simplistic example. One might even ask themselves why they bothered going through the hassle of defining a language to write something this simple. Well, they probably wouldn't. Where ED is useful is when applications are big, when they require reuse of many similar processes, or when they need to move seamlessly between multiple platforms.

As we mentioned before, this example could have been done just as easily in other languages and/or platforms. Imagine if we build an interpretive engines using .NET and Java. Then we could run the exact same application on each platform just by copying over the rules and application definition files. Our model becomes not just platform independent, but also independent of all the technical architecture that sits on top of that platform!

Gray Areas

Knowing when to write an ED, when to reuse an ED or what exactly to include in an ED are possibly the most difficult questions that any developer will face. Do not lose hope however. Generally the answers to these questions become very obvious as the ED is formed. These are not new questions; these are the same questions any development team has when developing highly modularized systems.

Extreme Cases

To define a web application using ED there are two extremes. The first extreme is to completely minimize the number of elements and rely on parameter attributes to do the majority of the work. The other extreme is to use an ED to essentially override or mimic an existing language, giving the developer the tools to develop the system at a very low level.

In the first extreme case, one could simply define an ED with single element *Application*. Let's say the element *Application* takes as arguments 1) the type of application, and 2) the name of the application. So by making the call:

```
<Application type="pinball game" name="Super Pinball">
```

We can define a pinball game. This of course assumes that you already have a model for a pinball game, and that the developer will be happy with this model. It is doubtful however that any developer would be happy with this, as they have almost no say over what the game will be like. The simple answer is to just add many more attributes. By adding "ball color, number of balls, can bump, bump warning message, etc..." the developer can start to better define the application.

In the second extreme case, let's say we just overrode the C++ language. Now we have an XML language based on C++. Does this help us? Probably not, as we have just added a one-to-one mapped layer to the C++ language. While it isn't always clear, there is a happy medium ground, which is the right place for an ED to rest. The typical goal is a balance between flexibility and ease-of-use.

Different Applications

Consider Web applications for an example. *Web application* is a fairly generic term which can mean anything from a simple shopping cart website to a robust intranet document management system. While it may seem that what works for one would work for the other, in fact, these systems differ greatly!

Assume that an ED is initially developed to handle shopping cart websites. As this ED evolves, the features that will become important are optimization, scalability, flexibility in dealing with financial institutions, order tracking, personalization, and so on. It should grow vertically, becoming more and more specialized with more advanced features for shopping cart sites. It should not grow horizontally to support other aspects of an organization's business.

If this same company were later to implement a document management system, they should keep the standards they have developed for the shopping cart website, but use them as a guide to develop a new ED. A tool such as a document management system will have much different requirements than a shopping cart. For one, consider the number of people using it. While a shopping cart site may be used by hundreds of thousands of people, a document management system is more likely to be used only by individuals within the company. Focus will not be on optimization or scalability, but rather on providing the most robust and user-friendly tools possible to speed the business process.

The second scenario, a document management system, introduces an entirely different set of requirements and priorities. It does not logically fit in with the first system, nor does it necessarily need to be fully integrated into that system.

Adding certain features to the document management system would neither add value nor take value from the shopping cart system. In every respect it makes sense that this should be a separate ED.

ED is not MDA or UML

ED (Elemental Development), UML (Unified Modeling Language) and MDA (Model Driven Architecture) are each different methodologies used for different purposes. UML and MDA mostly help architects to design and draw a big picture of the applications. ED on the other hand is a framework that guides both software developers and business analysts into creating highly logical reusable components after the application's requirements and data needs have been defined.

UML

UML is for modeling large applications before they are coded. It provides a set of rules that allows a development team to describe a large application in a standard, non-technical language.

> "A model plays the analogous role in software development that blueprints and other plans (site maps, elevations, physical models) play in the building of a skyscraper. Using a model, those responsible for a software development project's success can assure themselves that business functionality is complete and correct, end-user needs are met, and program design supports requirements for scalability, robustness, security, extendibility, and other characteristics, *before* implementation in code renders changes difficult and expensive to make." – Object Management Group, Inc.

MDA

MDA is for modeling architecture. As software grows more complicated, the time and technical resources it takes to implement it, increases. Building software to be dependent to a specific architecture can be a massive commitment, as newer technologies could be, and usually are, just around the corner. MDA allows developers to model their architecture and build software on that model, allowing them to build hugely complex systems without fear of being tied to a specific architecture.

> "The OMG MDA™ separates the fundamental logic behind a specification from the specifics of the particular middleware that implements it. This allows rapid development and delivery of new interoperability specifications that use new deployment technologies but are based on proven, tested business models. Organizations can use MDA to meet the integration challenges posed by new platforms, while preserving their investments in existing business logic based on existing platforms." – Object Management Group, Inc.

ED

ED is for building applications by separating the application/logic layer from the underlying technology. It provides a framework that software engineers (technology experts) and application developers/business analysts (application/business experts) can use to work together. When technology changes, software engineers will improve the model elements. And when there are new features and functionalities, application developers/analysts can bring their business knowledge and subject matter expertise to the table.

Benefits

There are many benefits to using an ED language including:

- Higher quality applications due to optimization of elements
- Cheaper
- Faster to develop
- Easy to implement new technologies into older applications
- Increased overall return on investment
- Reuse, reuse, reuse
- Highly maintainable
- Strict uniformity
- Shorter learning curve

Platform Independence

Consider this scenario. A XML ED language has been developed along with an engine and several applications. Now all of these applications need to be moved from a .NET platform to a Java EE platform.

Impossible… right? Not at all.

First, it will depend on what language has been used to develop the engine. If it has been developed in a platform independent language such as Java, then the move will require little work. The brunt of the work will be in moving over the code for the engine and configuring it properly on the Java server. After that, moving over the applications is no work at all.

If the engine has been written in a platform specific language, then the move is slightly more difficult. The engine essentially needs to be re-written in a language supported for the new platform. However, once that initial work is done the applications will function exactly as they did before. This may seem like a lot of work, but when compared to the work involved in rewriting many existing applications, it is a much more desirable alternative.

Portability

Because the environment is so self-contained, re-using and managing the ED language can save a great deal of time and money. Not only can updates be made to applications rapidly, but also many applications can be installing a new version of the engine in a single place.

Productivity

Using an ED greatly improves the productivity of everyone on the application development team. Because a single language has been developed at an organizational level, it applies well to developers, designers and managers, saving time in each instance.

Application Developers are not worried about the underlying infrastructure or architecture; they are only concerned with assembling the various components to build an application, not with dealing with components.

Managers don't have to worry about low-level code review and testing. Testing can be done at an assembly level since each individual component has already been tested. Energy can be spent on making the system conform to given specifications.

And finally, graphic designers do not have to know anything about code. Their job is to generate template pages, graphics and images, which then surround and present the pieces that have been defined by the ED.

Some Issues with ED

Some of the drawbacks of ED should already be obvious, but it is important to really look at and understand each. These problems are only noteworthy in a few situations, and in these situations, it might be best to avoid using an ED strategy.

EDs are Different

Different groups may come up with different language elements for a similar approach.

The fact that every ED is different is both a benefit and deterrent of the methodology. This means that any developer wishing to use an existing ED must learn each language.

However, this is not as bad as one might think. First, since each ED is modeled using XML, the code is highly readable and highly logical. It takes very little time to learn and understand how to assemble applications in any new ED unless it has been poorly constructed.

Second, since the idea of an ED is to drastically reduce the amount of code in a project, much less has to be learned. Instead of a language such as Visual Basic, which might have close to 2000 different functions and objects, an ED could theoretically be a single element. Of course in practice a relatively simple ED is more likely to have around 50 to 100 elements.

Finally, EDs can be standardized. This means that as various EDs grow and overlap, they can be easily combined and standardized. While the end goal of an ED is continuous evolution of the language, the intermediate goals must also include a continued ease through homogenization. This may take place on a global level, but more likely it would take place on an organizational level.

Difficult to Implement

Some may feel that the process of creating an ED is more complicated than the alternative of just building what it is they need to build. For those individuals who must rush to get something out the door, it probably is too complicated. ED is a concept that works for organizations that see the value in dedicating some extra resources up-front to capitalize on those resources later down the line.

For example, if one were creative, it might be relatively simple to paint a picture of a flower. Using a few paints and brushes, something that resembled a daisy could be reproduced in a matter of hours. Alternatively, one could spend their time on carving a ceramic print. It would be more difficult and take longer up-front, but when finished, they now have the capabilities to create many of the flower prints in just minutes.

Rather than just coding the application, a developer invests time in the architecture of the language. By identifying and building the components that will be needed to form the application, not only can the application be built quickly, but also developing future application will take a fraction of the time.

And, most importantly, suppose an ED framework has already been created for the type of software that we need to develop. And, even better, suppose it is available commercially and at an affordable price?

The Future of ED

The future of ED is promising. Unlike many technologies, ED does not require vast proliferation to be successful. ED itself benefits little from its own widespread adoption since each organization can develop and implement its own ED.

As ED becomes more and more well know, the theory behind it will evolve into more strict guidelines for optimal development practices. Additionally, tools will emerge that assist developers in development and implementation of ED based applications.

Predictions

One drawback of ED is that this technology is still not simple to understand. While using an ED that has already been created is much easier than coding an application using a traditional programming language, it is undoubtedly beyond the abilities of a great majority of small and medium sized businesses to create an ED from scratch. Instead, such a business will most likely implement ED languages, which have been developed by other organizations. They will leverage the knowledge and work that has gone into many implementations of a language to create their own, similar applications.

As a few specific languages become more widespread there will be a standardization of syntax across all the most popular languages; that will be followed by an increase in the number of tools for ED implementations. The most difficult aspect of this open source effort will be versioning control, but most likely as with any open source project the code will be controlled by a specific organization or group.

It may seem from this prediction that ED will eventually become nothing more than another standardized language. Is it doomed to meet the fate of all languages before it? Not at all. It is important to keep in mind that what is being described is the evolution and standardization of many languages, each of which is used to describe a different type of application.

As the demands on applications grow, so will the EDs, which describe them. In the same way that any application grows to meet the new demands of the organization, the ED will grow with each added feature or each new business requirement. And as new applications are thought-up, completely new EDs will be required to create them.

Possible Applications

There are an endless number of applications that can be built using the theory of ED. The most challenging aspect for the developer is attempting to separate different applications into different categories. At issue is knowing how far to take an ED, and when to stop expanding one ED and build another.

A good rule of thumb is that an ED should describe applications for a business process or group of business processes. For instance, selling merchandise online is a business process; order fulfillment and order tracking is a group of business processes; and tracking employee hours and invoicing is a group of business processes. While some business processes will be similar enough to group together, many will not.

Once these groups have been defined, expansion of the various EDs should move towards developing more enhancements, which fit only those business processes. Enhancements should never be made to incorporate a new set of processes into an existing ED.

Conclusion

ED describes a framework for developers and organizations to define and maintain their own application languages. This allows a maximum return on investment when initially developing internal systems by providing a method for quickly and easily reusing and maintaining that code.

ED is implemented by creating a set of rules that define an application, defining the application using the rules, and then building an engine to interpret and run the definition. While it has two more steps than traditional development, these two steps are what makes it so beneficial. By taking the time to isolate the definition from the code, developers gain great flexibility later in the development cycle because 1) the application can be changed without changing the engine code, and 2) the engine code can be changed without affecting the application.

After an ED has been created, it can be used for many more similar applications. As these applications are built and expanded, the ED used to build them will also expand. After many generations of this process, the ED will contain a great deal of information about that particular type of application, and will allow for rapid development of those applications.

The long-term benefits are primarily saving time and money associated with a highly reusable and maintainable system. Other benefits include fast development cycles, higher quality product, strict adherence to standards and a shorter learning curve.

ED is simply a methodology to help developers optimize their efforts, and for organizations to optimize their ROI. In this new economy of careful spending, no one wants to see their prior investments in software vaporize every time a new technology appears, nor do they want to throw away their investments in developing proprietary applications.

Where ED will go is difficult to say. This paper is just an introduction to the theory. Perhaps in time others will adopt and improve on the ideas that have been expressed. While nothing in computer science is new, we must always look for new and better ways to build a mousetrap. Especially in an economy that values efficiency, we must adopt strategies that will allow us to maximize our efforts in both the short and long terms.

References

Object Management Group, Inc.. 22 Aug. 2002. 17 Sept. 2002 <http://www.omg.org/gettingstarted/what_is_uml.htm>.

Object Management Group, Inc.. 02 May 2002. 17 Sept. 2002 <http://www.omg.org/mda/executive_overview.htm>.