A Graphical Approach to the Development of Deployment Agnostic Systems

Dr. Mark.B.Dixon
School of Computing and Creative Technologies, Leeds Metropolitan University, Leeds, England

Abstract - *The ever expanding number of environments in which computer systems are being used has led to the evolution of numerous development languages, tools and techniques. This paper discusses a predominantly graphical approach to software development that is deployment platform agnostic. The aim is to provide engineers with an approach to development that is general enough to be applied across the multitude of problem domains. By using a purely component based approach, in which target platform specifics are hidden from the language design, it has become possible to build a set of interrelated tools which allow for the development, manipulation and exchange of implementation solutions.*

Keywords: modeling; graphical; component; deployment

1. Introduction

One of the main difficulties faced by software engineers is the shear array of available development languages, run-time platforms and deployment architectures. The decision of which combination of available tools and techniques to be used is often dictated by the nature of the target system. Broadly speaking target systems can be classified as being desktop applications, mobile applications, embedded control systems, web applications, and distributed applications including Service Oriented Architectures (SOAs). Many of these systems are multi-tier of course, meaning that many modern solutions are actually a hybrid of the aforementioned categories.

The presence of multiple deployment possibilities has led to fragmented development approaches, not just at a language level but also at a tools level. For example, embedded systems development is vastly different to SOA development in terms of implementation languages, development approaches, tracing, debugging etc. The work described within this paper aims to provide a single development technique underpinned by a set of tools capable of supporting different target platforms.

The suggested technique and supporting tools, which has been named the Razor Development Environment (RDE), consists of a component oriented graphical notation supported by an underlying 3GL type language. The enforced use of a component based structure and strong support for re-use among components allows the majority of a solution to be developed using existing generic components which do not include any target specific information. A small number of platform specific components can then be linked with the main solution to produce a deployable system.

2. Background

The graphical notation of the RDE was initially devised as part of a project to develop an embedded operating system [1]. This initial work was extended upon in an effort to allow the technique to be applied to a wider variety of problem domains, including those outside the embedded systems arena. During this work much consideration was given to the best mechanism of supporting deployment of RDE based systems. The possibilities considered included run-time interpretation; native compilation; and language translation. The former of which has been implemented in a reference implementation.

The fact that there were so many deployment possibilities led to initial confusion. It became clear however that this level of flexibility was actually a strength of the approach since it provided the basis of a platform neutral technique. By removing certain constructs from the core RDE, such as direct support for dynamic component instantiation and multiple threading, a fully deployment agnostic approach was developed.

3.The Razor Development Environment

3.1 Overview and Architecture

The RDE provides a selection of tools and techniques that allow the development of software systems using a reusable component based approach. At the core of the environment sits a Document Object Model (DOM) that provides a canonical representation of the application under development. This DOM defines the available components along with how they are configured and connected via their interfaces.

The DOM may be populated using a graphical notation, a textual 3GL type language or an XML document. These three representations are 100% semantically equivalent, hence DOM contents can be manipulated using any representation, independent of the original input method. Any number of deployment tools can then be developed to

produce systems via interrogation of the DOM. A graphical representation of the environment [2] is shown in Fig. 1.

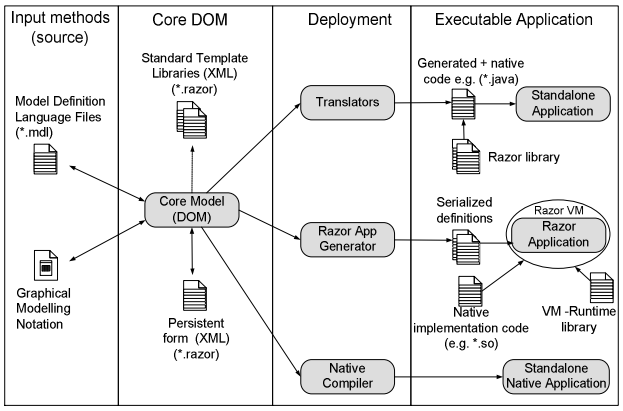

Figure 1. The Razor Development Environment

A developer can work on components with only limited regard for the architecture on which they are to be deployed, hence most components are generic re-usable objects. Only target specific components need to take into account deployment and domain specifics. Due to loose coupling and strong reusability support, both generic and platform specific components can be easily sewn together to provide a complete solution.

The RDE has implicit support for automated testing. The ability to specify compliance tests when defining component interfaces ensures better support for mature concepts such as the Programming by Contract [3] approach along with contemporary development practices such as Test Driven Development (TDD) [4]. Implicit testing support also allows for better independent development of components, since associating well defined tests with specific interfaces in effect provides a mechanism of ensuring semantic compliance.

3.2 Design Principles

The RDE was designed by taking into account many commonly agreed upon design principles, mainly derived from the Object-Oriented paradigm. Many of these concepts enhance the capability for component re-use, which is fundamental to the RDE approach.

The RDE is extremely 'interface centric' in nature. In fact, only interface definitions may represent a type within the system (languages such as Java, C++ and Objective C allow implementation classes to be used as the type of variables and parameters). This design principle is very important for re-use since only interfaces are ever passed as values between component services, also the inheritance model is simplified since implementation inheritance is no longer required or even possible. The demotion of the traditional 'Class' may seem radical but it allows for much better support for the Open-Closed principle [5] which states that elements should be open for extension but closed to modification. It also helps address the well known fragile base class problem [6] since inheritance hierarchies are interface based, rather than implementation based. Finally, the Liskov Substitution Principle [7] is well supported, not only due to the interface centric nature but due to the ability to ensure semantic compliance of interfaces via implicit testing support.

Systems are defined by identifying component instances and binding them together via their external ports. Each port represents a single interface which defines a number of services, attributes or signals. Each component instance acts as an implementer of one or more port interfaces, either directly through terminal ports or by delegation to sub-components. This multi-interface ability supports the Interface Segregation Principle [8] which promotes the idea of providing many fine grain interfaces in order to help reduce dependencies.

The rules that determine the legality of port bindings between components are based on a signature which does not include the name of the port, only the type information. This loosens the coupling somewhat between components, again promoting re-use. This in-turn also enhances support for the Dependency Inversion Principle [9] that suggests higher level components should not be dependent on lower level components. Within the RDE lower level components can be connected via ports rather than being embedded within the higher level components. The dependency injection pattern [10] is also easily supported by the binding together of higher and lower level components. The weakened port binding semantics also mean that components can be developed more independently, since their visible namespace is only defined within the component itself. Hence, the implementation never needs to be aware of port names that exist outside of the immediate component. Well known classical design patterns [11] are much easier to support in an interface centric component based approach which exhibits low coupling.

3.3 The Graphical Notation

One of the primary aims of the RDE was to support a predominantly graphical model based approach to development. Abstracting to a graphical representation not only simplifies development but better supports the ability to hide deployment specifics. It has been pointed out that model driven approaches are better placed to deal with the complexities of modern platforms, while also allowing better representation of problem domain concepts [12].

A graphical notation has been defined to allow the declarative definition of RDE based software systems. This notation, known as (Razor's) EDGE, allows for the construction of an entire system via the use of a single diagram type. Rather than base the notation on an existing notation, for example by defining a UML profile [13], the decision was made to create the simplest notation possible; while still supporting all required concepts of the underlying DOM.

Unsurprisingly the key elements represented within the notation are Interface and Component definitions. Each of these elements is capable of hosting one or more Ports. Both

Interfaces and Components are represented using the same graphical shape, which may seem odd at first but allows for the use of a single model to define all parts of a system. An interface defines a number of 'provided' and 'required' ports, which in essence determines the direction of dependency when a port based on the interface is bound. The example presented in Fig.2 shows the definition of a ‘Flow’ interface that allows controlled flow of data. A single service is ‘provided’ and two signal ports are ‘required’ in this particular example.

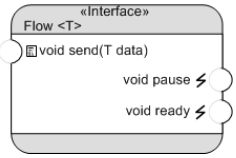

Figure 2. An interface defined using the graphical notation

A component realizes the implementation of a number of interface types. This is accomplished by either providing a terminal port; or delegating to a sub-component instance. Components can also 'provide' and 'require' a number of ports, which are independent of any ports that are defined on implemented interfaces.

An example of a Component is shown in Fig. 3. In this example several contained part instances both provide and require the previously defined ‘Flow’ interface.

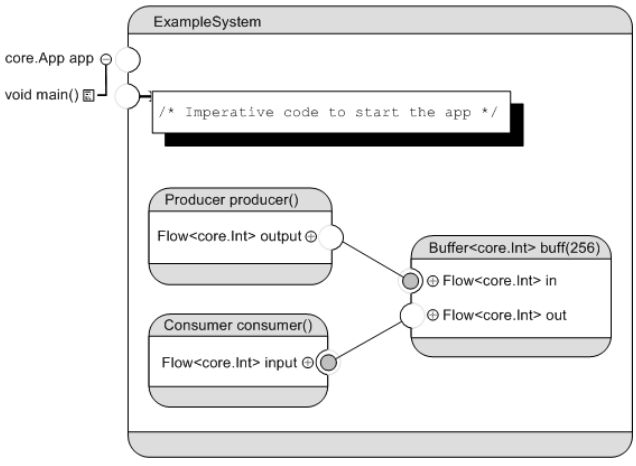

Figure 3. A component defined using the graphical notation

As well as being able to represent the internals of a component using a structural model, components can also be defined as finite state machines. This allows for the definition of control processes such as those used in real-time embedded systems or SOA orchestration languages such as the Web-Services Business Process Execution Language (WS-BPEL) [14]. Interestingly enough almost exactly the same notation can be used to show either definition style, with only the addition of transitional flows being required to support the state machine approach.

All ports have a *nature* that determines whether they provide a service, store an attribute, represent a signal or are compound, i.e. are nested ports based on other interfaces. Support for compound ports is a very important abstraction mechanism and allows interfaces which are commonly used together to be wrapped into a single conduit type port.

Configuration values and configuration types are used to support customization of components during instantiation. These are analogous to constructor parameters and generic types respectively.

Terminal service implementation code is defined using a simple 'C' like grammar. Within the initial reference implementation Javascript was used but has since been replaced as many concepts supported by Javascript are simply not required within the RDE. The language is only required to consist of standard statements and expressions. Support for creating new objects for example is handled using components, since this is a deployment specific aspect and is not always supported by certain domains such as resource constrained embedded systems.

3.4 The Model Definition Language

A Model Definition Language (MDL) was developed as an alternative mechanism for populating the RDE DOM. Although regular use of the EDGE graphical notation is the final aim, during the research phase the ability to quickly change the grammar rules within a parser make a 3GL textual language more appealing at this time.

The MDL provides the same constructs as the graphical notation as keywords within the legal grammar. The language supports definitions of interfaces, components (using the ‘implementation’ keyword) and binding of ports. The service implementation code uses exactly the same grammar as it does within the EDGE graphical notation; hence MDL is a superset of the imperative code used in the EDGE model. Example MDL code that is equivalent to the previous example is shown in Fig. 4.

```
implementation ExampleSystem {

      provides {
           core.App app;
      }

      parts {

           Producer producer();
           Consumer consumer();
           Buffer<core.Int> buff(256);
      }

      bindings {

           producer.output -> buff.in;
           buff.out -> consumer.input;
           app.main { /* Imperative code to start the app */ }
      }
}
```

Figure 4. A component defined using MDL based code

4. Related Work

The existence of component based techniques is well established. Technologies such as DCOM [15], Corba [16] and JavaEE [17] tend to focus on higher level distributed components however, and often act as a wrapper for existing languages. Hence, they represent a different layer in the software stack than the RDE which constructs systems from components but does not necessarily deploy them as such. OSGi [18] has a little more in common with the RDE since it is not specifically designed to create distributed systems. However it has not been designed to be deployment agnostic and requires a very specific run-time environment.

The work most closely related to the RDE appears to be work carried out by France Telecom within the Fractal Project [19]. This too is a component based approach in which models can be graphically defined. An Architectural Description Language [20] is supplemented with C++ or Java in order to build a full system. Although there are similarities they are also differences regarding environment semantics and the graphical notation. As with the more established technologies, Fractal is primarily based on wrapping existing programming languages. Also the bindings between components within Fractal can be representative of higher level network connections etc.

The RDE is in effect an instance of the Model Driven Engineering (MDE) approach. Hence, from a modeling point of view the UML and related MDA technologies [21] cannot be ignored. The model driven paradigm as defined by the Object Management Group (OMG) aims at providing platform independent models which are mapped to platform specific models using transformation rules. This concept however is fairly complex and requires the definition of multiple mapping and translation rules for each deployment target. Introducing more complexity into the development process is the opposite of what the RDE approach is trying to accomplish.

In terms of purely graphical development of software systems, the MIT App Inventor software [22] and its underlying technologies seem to be based on similar concepts to the RDE. However, the graphical aspects of the RDE are provided to support an architectural level of design, whereas the App Inventor supports graphical design of the user interface along with the procedural aspects of the system. Hence the RDE EDGE notation provides a higher level of abstraction with the lower level imperative code being defined using a 3GL type language (MDL).

The fundamental difference between the RDE and the related work is that its primary aim is to provide a single format that allows for the definition, exchange, and deployment of components which when combined can be used to create software solutions for a diverse set of application domains. In many respects it is synonymous to the philosophy that drove the development of the XML, but instead of being data centric, it is behavioral centric in nature. The tools and techniques developed within the RDE are all specifically designed to provide a realization of this core concept.

5. Evaluation

An initial Java based implementation of the RDE DOM and MDL parser has shown the concept to be a viable approach. These tools are likely to form the basis of an initial release of the environment and have been developed with a commercial strength product in mind. In contrast however, the current run-time environment in which the systems are deployed is unlikely to be usable for real systems. This is because it is a simple interpreter which is capable of executing DOM defined models. Although this serves as a good reference implementation for testing purposes it lacks good performance and does not currently apply any optimization prior to execution.

Although a full implementation of the EDGE notation is not yet complete a prototype has been developed as an Eclipse based GEF [23] dependent plug-in. A screenshot of the graphical editor is shown in Fig.5. At the moment this is not yet functional enough to support development of test systems, thus all current evaluation has been done by developing systems using the purely textual based language (MDL).

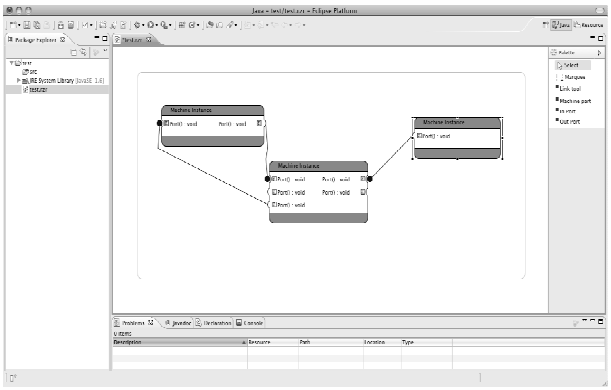

Figure 5. A screenshot of the prototype EDGE tool

The development of the test systems has gone some way to validating the supported constructs along with the mechanics of the approach. This work has also shown however that development via the textual based language alone is a fairly difficult process when compared to traditional OO based programming in languages such as Java. This appears to be due to the fact that the RDE was always designed to be predominantly graphical in nature; hence using a textual language to define the architectural properties of a system is often likely to be counterintuitive.

The development of the run-time environment has also highlighted difficulties when providing an implementation mechanism for compound natured ports. There is a large amount of complexity involved in ensuring that bindings between nested ports are configured correctly during deployment. The ability to pre-examine models and produce optimized compiled code is likely to reduce this problem in the future when compilation or language translation becomes the preferred mechanism of deployment.

6. Conclusions and Future Work

Once a full set of development tools are available the system will need to be more thoroughly evaluated via the production of some industrial strength solutions. The significant point is that the available set of tools will be applicable to the development of all Razor based systems irrespective of the target platform. Only the deployment specific compilers or run-time environments need to be target aware.

A component model such as this could only be a practical reality if mechanisms for the locating and matching of components were provided. Hence publishing and discovery services, such as the Web Services Description Language (WSDL) [24] and Universal Description Discovery and Integration (UDDI) [25], need to be adapted to work within the scope of the RDE.

There needs to be more work undertaken on creating both deployment agnostic and deployment specific components. Once a library of components is available the tool can be released to a wider audience in order to gather feedback. The creation of an open source tool chain, along the same lines as the GNU GCC project [26], would help maximize availability of the proposed approach and provide a low cost of entry for prospective developers. The compilation of Razor compliant components directly into native machine code is the next major aim of this work.

References

[1] K.Tindell and M.B.Dixon, 'Scalios', A scalable Real-Time Operating System for resource-constrained embedded systems, computer software. Published by JK Energy Ltd. 2008. Available from: https://github.com/jkenergy/scalios/

[2] M.B.Dixon, "Supporting component oriented development with reusable autonomous classes," ARPN Journal of Systems and Software, vol.1, no.5, August 2011, pp. 182-193, ISSN 2222-9833.

[3] B. Meyer, Applying "Design by contract," Computer (IEEE), vol. 25, issue 10, October, 1992, pp. 40–51, doi:10.1109/2.161279.

[4] K. Beck, Test-Driven Development by Example. (The Addison-Wesley Signature Series), Addison Wesley, 2002.

[5] B. Meyer, Object Oriented Software Construction. Prentice Hall, p 23, 1988.

[6] L. Mikhajlov and E. Sekerinski, "A study of the fragile base class problem," Proc. ECOOP'98 - 12th European Conference on Object-Oriented Programming, Brussels, Belgium, 1998, pp 355-382.

[7] B. Liskov and J. Wing, "A behavioral notion of subtyping," ACM Transactions on Programming Languages and Systems (TOPLAS), vol 16, issue 6, November, 1994, pp. 1811 - 1841.

[8] R. Martin, The Interface Segregation Principle, C++ Report, August, www.objectmentor.com/resources/articles/isp.pdf, 1996.

[9] R. Martin, The Dependency Inversion Principle, C++ Report, May, www.objectmentor.com/resources/articles/dip.pdf, 1996.

[10] M. Fowler, Inversion of Control Containers and the Dependency Injection Pattern, http://martinfowler.com/articles/injection.html, Jan 2004.

[11] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[12] D. C. Schmidt, "Model Driven Engineering," Computer (IEEE), vol. 39, issue 2, February, 2006, pp. 25-31.

[13] J. Rumbaugh, I. Jacobson and G. Booch, The Unified Modeling Language Reference Manual, 2nd Edition. Object Technology Series, Addison Wesley, Chapter 12, 2004.

[14] OASIS, Web Services Business Process Execution Language Version 2.0 standard. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, April 2007.

[15] T. L. Thai, Learning DCOM, O'Reilly Media, 1999.

[16] Object Management Group, Common Object Request Broker Architecture (CORBA), http://www.omg.org/spec/CORBA, 1997.

[17] J.Farley and W.Crawford, Java Enterprise in a Nutshell, O'Reilly Media, 3rd Edition, 2005.

[18] OSGi Alliance, "OSGi Service Platform Release 4," OSGi Alliance Specifications, http://www.osgi.org/Specifications/HomePage, 2009.

[19] OW2 Consortium, The Fractal Project, http://fractal.ow2.org, 2009.

[20] M. Leclercq, A.-E. Ozcan, V. Quéma and J.-B. Stefani, "Supporting heterogeneus architecture descriptions in an extensible toolset," Proc. 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 2007.

[21] A.Kleppe, J.Warner and W.Bast, MDA Explained: The Model Driven Architecture: Practice and Promise. Object Technology Series, Addison Wesley, 2003.

[22] D.Wolber, H.Abelson, E.Spertus and L.Looney, App Inventor, May 2011. O'Reilly.

[23] The Eclipse Foundation, The Graphical Editing Framework (GEF), http://www.eclipse.org/gef, 2010.

[24] W3C, Web Services Description Language (WSDL) Version 2.0 Part 1 : Core Language. http://www.w3.org/TR/wsdl20, 2007.

[25] OASIS, UDDI Specification 3.0.2. http://uddi.org/pubs/uddi_v3.htm, 2004.

[26] Free Software Foundation, GCC, the GNU Compiler Collection, http://gcc.gnu.org/, 2011.