Supporting Component Oriented Development with Reusable Autonomous Classes

Mark B. Dixon
Computing and Information Systems Group, Leeds Metropolitan University, England.

m.dixon@leedsmet.ac.uk

ABSTRACT

Reuse during development of software systems has long been touted as a mechanism of reducing costs, increasing quality and speeding up development. At the same time the ability to develop systems using a predominantly graphical approach has long been promised but has never really delivered. The described development approach aims to address both of these issues, by providing a development framework designed to maximize reuse, while simplifying the development effort. A secondary aim, to include mechanisms that implicitly offer support for contemporary software development processes, is also addressed by the proposed development framework.

Keywords: Software Components, Model Driven Development, Visual Programming.

1. INTRODUCTION

This paper introduces the Razor Development Framework. This experimental framework provides a selection of tools and techniques that allow the development of software systems using a reusable autonomous class based approach. The benefits of reuse within software system development are well documented [1] and have driven the design of many modern programming languages. The Object Oriented (OO) paradigm goes a long way to supporting reuse during development, but falls short when it comes to supporting truly autonomous components. The Razor Development Framework aims to extend the Object Oriented model to allow direct language level support for Component Oriented Development (COD). Additionally it provides direct support for contemporary development practices such as Agile development and eXtreme programming, which have been popularised by Martin [2], along with Test Driven Development as described by Beck [3].

Many component models have been developed over the years. Examples include COM [4], Corba [5] and OSGi [6]. The Razor Development Framework is not this type of component supporting environment; it is aimed at improving re-use and allowing construction of systems using build time integration of component like elements. In fact, in many respects it has more in common with existing OO type programming languages than the existing component models. The term “autonomous class” was used to ensure a clear differentiation between what is commonly understood to be a component and the primary construct discussed within this paper, which is more like a Class that exhibits component like features. Existing component models tend to provide higher level support and are retrofitted to existing programming languages, rather than being developed to support a lower level Component Oriented Development approach.

One of the key opportunities presented by both the OO and component based approaches in general is the potential to develop systems using a predominantly graphical notation. Graphical techniques such as the Unified Modeling Language [7] combined with the Model Driven Architecture [8] have long promised this possibility. Although such techniques have been successful in helping support the development process, they have never achieved their full potential of becoming the core artifact within the development of a system. One of the reasons for this is that dynamic behavior often exhibited by imperative type languages is not easily captured using diagrams1. Hence the more declarative a development technique becomes, the more easily it is represented using structural diagramming techniques.

The Razor Development Framework attempts to enhance the claimed advantages of reusability while also providing a graphical approach to development. Lower costs, better quality and faster development should indeed be possible. Mature and extendible Integrated Development Environments (IDEs) such as Eclipse [9] and Netbeans [10] now provide a basis on which cooperating supporting tools can be developed, thus allowing the various facets of such an approach to be realized. Such tools can seamlessly integrate support for design, development, testing and deployment. Most notably a graphical notation can be included as the primary tool for development.

2. BACKGROUND

The Razor Development Framework was instigated as part of a project to develop an embedded real-time operating system compliant with the AUTOSAR standard [11]. Such systems are often highly optimized in the target environment, due to limited resource availability, and thus are often configured prior to

---

1 State charts are excellent at capturing certain types of dynamic behavior, but tend to represent control sequences based on external events rather than algorithmic processes.
deployment. The configuration process for an application that targets such an operating system tends to be a fairly laborious process. To help simplify the configuration a graphical approach was developed. Although this was fairly specialized the resulting notation exhibited some interesting features, which clearly could be applied in a wider systems development domain. Since then a much more mature and wider scoped framework has been developed. Also a textual based 3GL style language, known as the Model Definition Language (MDL), has been added in order to provide a secondary mechanism of developing systems using the same approach.

Many aspects of the framework have been developed by examining some of the core principles regarding the construction of reusable object based systems. Commonly agreed upon ideas such as those discussed by Martin [2] and Pressman [12] including the Open-Closed Principle; the Liskov Substitution Principle; the Interface Segregation Principle; the Dependency Inversion Principle; and the Programming by Contract approach are all well represented within the framework's design, especially with regard to the definition of the autonomous classes themselves.

3. KEY COMPONENTS

There are a number of key tools within the Razor Development Framework. Together these provide a way of developing and deploying a Razor based application. The hub of these tools is a Document Object Model (DOM) which provides a representation of the application under development. More precisely the Razor DOM contains a description of the available autonomous classes and how these are used together to implement an application. The Razor DOM may be populated in a number of ways. This means that a Razor based system can be described using a 3GL style textual language, an XML document, or a graphical notation. A development team may use any combination of these input methods.

A reference implementation of the DOM has been created using the Java programming language. This provides a single API to allow instantation, population, manipulation, interrogation and serialization of the DOM. Once a model is defined within the DOM a target system deployment can be created. There are various ways in which a system of this type can be deployed. Possible approaches include run-time interpretation, language translation, or native compilation. The former of which is currently supported by the reference implementation. Current work is focusing on building an efficient, natively compiled run-time environment in which systems can be deployed via the Internet. Since the representation of each autonomous class is stored as an XML document it becomes possible to dynamically construct client side applications by downloading and caching autonomous class definitions. This approach aims to provide a basis on which Rich Internet Applications (RIAs) can be easily developed and effectively deployed without being constrained by the limitations of current browser technology.

The idea of language translation may widen the appeal of Razor somewhat. For example, this approach could allow a development team to deploy a Razor designed system as a JavaScript [13] only application, thus supporting development of Web-based applications. The same development team could also target Android [14] with another Razor designed system, thus supporting development of mobile platform applications. Native compilation of the DOM for a specific target could be achieved by creating a front end tool for the GNU Compiler Collection [15]. Although this approach would undoubtedly result in faster execution times, the portability of developed systems would be lost. Native compilation is a longer term goal of the research work. A graphical representation of the Razor Development Framework is shown in figure 1.

Fig. 1 The Razor Development Framework architecture

4. FUNDAMENTAL CONCEPTS

4.1 System Construction

Razor is principally based on system construction rather than implementation. The primary aim is to support Component Oriented Development (COD) by the plugging together of existing pre-developed autonomous classes. The realization of this primary aim therefore requires that classes are as reusable as possible. The DOM and the associated input methods have been designed to ensure
that this is indeed the case. From a developer’s point of view the framework is extremely Interface centered in nature.

Systems are constructed by identifying autonomous class instances and connecting them together via their external interfaces, which are defined as a number of ports. An interface identifies provided ports and required ports. Hence the required ports of one instance are connected to the provided ports of other instances. This provides/requires port interaction pattern is commonly used within existing component frameworks and modeling notations. The autonomous class implementations typically realize (implement) several different interfaces simultaneously.

4.2 Port Connectivity Rules

An important aspect of the Razor approach is that port compatibility is based on a signature that does not include the names of the ports. This is significant since it allows the ports of instances to be connected to a wider variety of target ports. Additionally this allows an autonomous class implementation to refer to names defined within its own namespace, with no concern for any names outside of the class itself. This is a key concept for improving reusability and can be referred to as Signature Based Binding (SBB). The loosening of connectivity semantics promotes reduced coupling between autonomous classes. It also allows for easier externalization of dependant classes thus supporting the Dependency Inversion Principle [16], which effectively states that high level components should not be dependent on low level components, i.e. Lower level autonomous classes can be easily accessed via ports rather than being directly incorporated within high level autonomous classes.

4.3 The Interface Centric Nature

Within the Razor framework there is a very clear separation between types and implementations. This allows multiple implementations of a specific type (interface) to coexist within the system. It also ensures that developers don’t accidentally hard-code the need for a specific implementation into the type information of the system. Such situations are often seen in languages such as Java, where a variable or parameter is declared using a Class name as the type instead of an Interface name. Within the Razor framework implementations cannot be used as type names. This concept may be referred to as Type Only Typing (TOT). An implementation does not need to be explicitly declared as realizing specific interfaces. This information can be derived at run-time. This has important implications for reuse, since it allows new interfaces to be developed and retrofitted to existing implementations. This process can be described by the term Interface Retrofitting (IR).

The explicit separation of types from implementations, along with the interface retrofitting ability, improves reusability of autonomous classes and simplifies team based development. Also an implementation has no restriction on the number of interfaces that it implements. These aspects allow for better application of the Interface Segregation Principle [17], which states that it is better to have many fine grained interfaces which are client specific rather than a single interface. Supporting ISP reduces coupling and dependency between autonomous classes.

4.4 The Inheritance Model

The Razor framework does not support implementation inheritance in the classical sense, but does support type (interface) inheritance, i.e. Type Only Inheritance (TOI) is provided. Support for implementation inheritance in programming languages often degrades support for the Open-Closed Principle, which states that elements should be open for extension but closed for modification [18]. Within the Razor framework implementation inheritance is achieved using a composition and delegation style pattern. An autonomous class “inherits” from another class by realizing the same interfaces as the super-type and then implements these by delegating to a contained instance of the super-type. The contained super-type instance acts like any other contained instance, in that access is via the externally accessible ports only. This pure black-box inheritance model, combined with explicit separation of types from implementations\(^3\), ensures strong support for the Open-Closed Principle and also helps address the Fragile Base Class problem [19].

4.5 The Typing Model

The Razor framework is strongly typed but supports both static and dynamic type checking. Types can be explicitly assigned during development allowing static checks to be performed at build time. Alternatively a special dynamic type can be assigned that indicates the need for run-time checking. Applying dynamic typing means that the type is determined at run-time by the Interfaces implemented by an object, rather than via the type assigned to the associated identifier. This approach is sometimes referred to as Duck Typing; a phrase originally coined by Martelli [20] and later discussed by Eckel [21] and is applied within the Razor run-time using the Interface Retrofitting capability.

Such a mixed typing approach provides a mechanism for allowing rapid script style prototyping to be undertaken, while allowing more stringent static type checking to be employed when required. An autonomous class can be developed purely with static typing, with all run-time type conversions from the dynamically typed

\(^3\) All external ports must be declared within an interface and cannot be implemented otherwise. This is in contrast to languages such as Java, C++ and Objective-C, which permit method implementations to exist without the need to explicitly declare these within an interface.
values being undertaken external to the class. This means developers may choose to spend extra time on developing statically typed core classes, which can be easily deployed within larger systems where dynamically typed values exist.

The ability of autonomous classes to be tailored into a specific context is very important and therefore they can be parameterized during instantiation. The parameterization allows implementations to react more appropriately to differing usage scenarios. Both value and type parameterization (generics) is possible. Indeed the support for generic types has been developed into the Razor framework. The ability to ensure an implementation complies semantic with a defined interface provides much stronger support for the Liskov Substitution Principle [22]. This principle specifies that sub-types should be substitutable for base types, not only from a syntactical, but also from a semantic point of view.

The ability to associate tests with both implementations and interface types aims to improve quality; support Test Driven Development (TDD); and aid in team development efforts. For example, a team can independently define an interface along with its compliance tests and then delegate the implementation work to a different team. Associating tests with interface definitions is a method of supporting the Programming by Contract approach [23]. The application of this idea within the Razor framework can be referred to as Type Conformance Testing (TCT).

4.6 Implicit Testing Support

Automated testing is directly supported within the core DOM of the framework. Tests may be associated with both autonomous class implementations and interface type declarations. The former allows testing of a particular implementation within a specific context. The latter ensures that an interface type has been correctly implemented by any available implementations (including sub-type implementations). This is a rather interesting concept since it means that interface declarations, combined with a set of associated compliance tests, describe both the syntactical and semantic nature of the interface. This is in contrast to many OO programming languages where interface declarations express syntactical information only, with the semantics being inferred from the name of the interface or natural language descriptions. The ability to ensure an implementation complies semantically with a defined interface provides much stronger support for the Liskov Substitution Principle [22]. This principle specifies that sub-types should be substitutable for base types, not only from a syntactical, but also from a semantic point of view.

The ability to associate tests with both implementations and interface types aims to improve quality; support Test Driven Development (TDD); and aid in team development efforts. For example, a team can independently define an interface along with its compliance tests and then delegate the implementation work to a different team. Associating tests with interface definitions is a method of supporting the Programming by Contract approach [23]. The application of this idea within the Razor framework can be referred to as Type Conformance Testing (TCT).

5. THE MODEL DEFINITION LANGUAGE

A 3GL type language has been developed to allow population of the Razor DOM. Within the reference implementation this is known as the Model Definition Language (MDL). The MDL parser was developed using the JavaCC lexical analyzer and parser generator [24], [25]. The MDL was developed to provide a low cost of entry to the Razor framework approach since it allows development of Razor based systems using a standard text editor. Also some developers feel more comfortable with code rather than models. The fact that Razor development is more about construction than implementation however means that the 3GL language is much more declarative in nature than many existing programming languages. The language has been carefully designed to ensure there is minimum “surprise” for experienced developers. When traditional procedural style grammar is required it follows the common C style syntax as defined by Kernighan and Ritchie [26]. In fact the use of ECMAScript is currently employed to represent service implementation code.

The central construct within the MDL is the Interface. This defines the ports which are to be provided by an associated Implementation. Interfaces may optionally contain a default implementation within the same source file, but this is not enforced. One or more tests may also be associated with an interface to ensure semantic compliance by implementations. Each defined port can be either a service port; an attribute port; a signal port; or a compound port. Service ports provide some sort of functionality and are similar to methods within existing object oriented type languages. Attribute and signal ports hold a value which can be accessed directly via a port. A signal port is designed for use within multi-threaded systems or state based environments. The compound port allows management of complexity through abstraction. A compound port contains sub-ports which are generally handled as an atomic unit, but can be accessed independently when required. Compound ports are based on an Interface type. Ports can also be identified as being synchronized, thus allowing thread-safe access to be provided within multi-threaded environments.

An implementation is defined using a number of parts and port bindings. The parts represent contained autonomous class instances. The bindings define how the ports are implemented. Ports can either be bound to the port of a contained part or implemented by the class itself. In the latter case such ports are known as terminal ports. Terminal service ports may be implemented directly within the MDL or be declared as being “native”. In this case an implementation of the service must be provided within a native programming language. This allows development of core autonomous classes to be undertaken in an existing programming language such as Java or C. It also allows existing code libraries to be wrapped into autonomous classes and made available within the Razor development framework.

A simple example of a “Counter” is given in figure 2. This makes use of a default implementation that may be associated with an interface. This autonomous class counts to a value determined via a required attribute port. Once the wrap value is reached the counter resets and a call is made to an outgoing service port.

4 In future versions ECMAScript may be replaced. The ability to contextualise the language however makes it ideal for use in the Razor framework at this time.
Notice how the wrapped service port of part “c1” is bound to the incCounter service port of part “c2”. Although the signatures of these ports vary somewhat, i.e. the incCounter returns an integer and the wrapped port has a void return type, this binding is permitted and the returned value is simply discarded. This is an example of how Razor relaxes type rules when it is permissible to do so. If the ports had been bound in the opposite direction this would not have been allowed.

There are of course additional constructs present within the MDL. Those shown are of primary importance however. The grammar for the declarative parts of the MDL language is shown using Extended Backus-Naur Form (EBNF) [27] within the appendix. As mentioned earlier the imperative implementation code is currently handled using ECMAScript and hence omitted from the provided grammar. The full grammar for ECMAScript is available within Annex A of the ECMA-262 specification [28].

6. THE GRAPHICAL NOTATION

A graphical notation has been developed as an alternative mechanism for populating the Razor DOM. The longer term aim is to make this notation the primary development artifact. Although the Razor philosophy is to promote declarative type development as much as possible, there will always be a need to support algorithmic detail. Hence the graphical notation is augmented with the MDL when necessary, i.e. to provide implementation details of terminal services in an imperative manner.

An Eclipse plug-in is currently under development to support the Razor development framework including the Razor Graphical Notation. A prototype tool has been developed using the Graphical Editing Framework (GEF) plug-in [29] which provides a Model-View-Controller (MVC) style infrastructure. This graphical tool provides a mechanism of defining implementations by binding the ports of various parts. The declarative nature of the underlying DOM ensures that the majority of the system can be defined using a graphical approach.

The notation itself was initially designed as a UML profile in an attempt to use existing recognized standards. It became clear however that this was resulting in an over complex notation which could be better represented using a bespoke design. As work on the notation progressed a very small and concise technique developed, with only a single type of graphical model being required to allow definition of a whole system. The key elements of the notation are shown along with annotations within figure 4.
intuitive to use, which is a very important aim of the Razor Development Framework. The UML, although very available.

more abstract the need to specify such algorithmic detail graphical notation in figure 5 and figure 6. The figure 3 from the MDL section is shown using the reduces, hence a system definition can become completely the definition of a very limited number of elements.

graphical notation is the opposite of this, in that it supports of development using many different views. The Razor This is because the UML aims to cover all possible aspects

Fig. 4: Key elements of the Razor Graphical Notation

Most of the notation is self explanatory and intuitive to use, which is a very important aim of the Razor Development Framework. The UML, although very powerful, is large and complex to use in many situations. This is because the UML aims to cover all possible aspects of development using many different views. The Razor graphical notation is the opposite of this, in that it supports the definition of a very limited number of elements.

The “Counter” example presented in figure 2 and figure 3 from the MDL section is shown using the graphical notation in figure 5 and figure 6. The algorithmic (functional) details of the services provided are specified using the MDL. As implementations become more abstract the need to specify such algorithmic detail reduces, hence a system definition can become completely declarative providing the required autonomous classes are available.

Fig. 5 “Counter” Autonomous Class Implementation Model

7. COMPOUND PORTS

Compound ports are an extremely powerful mechanism for handling complexity within the Razor framework. Non-trivial systems often have a large number of ports that need binding. In many cases a related set of ports are all bound together, i.e. they all have the same source and destination class instances. Rather than having to bind all of these ports independently, compound ports allow them to be wrapped together and dealt with as an atomic unit. Compound ports can also be nested, in that a contained port may itself be a compound port. This enables the building of structural port type hierarchies. This ability to group a number of interrelated ports into a single conduit enables developers to better identify ports that are either commonly used together, or necessarily used together.

In many respects compound ports are very similar to autonomous classes in that they host many contained ports. This similarity enables compound ports to fit very naturally within the Razor framework, since the contained ports are simply specified using an interface. In fact there is no difference between an interface realized by an autonomous class and an interface used to define the ports within a compound port. Hence a single interface can be used for both purposes if necessary.

Since compound ports carry many contained ports there inevitably comes a point at which the contained ports need to be individually bound, i.e. the compound port needs to be ‘popped open’ to allow access to the contained ports. The Razor framework supports such access using qualified path names within the MDL paired with an equivalent visual mechanism for the graphical notation. Hence it is possible to independently bind contained ports to different targets when necessary.

Compound port structural hierarchies should not be confused with inheritance hierarchies. The former refers to type nesting, whereas the latter refers to type reuse and substitutability.
Consider a compound port designed to support logging of information. Such an activity may require an interface such as that defined in figure 7.

```java
interface Logger {
    provides {
        void logValue(String value); // log a value
    }
    requires {
        String getHeader(); // access a 'header' string
        String getFooter(); // access a 'footer' string
    }
}
```

Fig 7: An interface to support data logging activities

The provided port enables data to be logged, whereas the required ports allow access to any header and footer information that is to be amended to the generated output. This example highlights how a compound port can support bi-directional communication. A typical application of this particular compound port is shown in figure 8. The binding of the single compound port effectively causes the binding of all contained sub-ports.

As can be seen from the graphical model, the logValue service port appears as a provided port within the FileLogger, but as a required port within the StatsCalculator. Hence, the directional nature of contained ports is inverted when the host compound port is used as a required, rather than a provided, port.

The necessity for compound ports becomes more apparent within complex systems, since they can substantially reduce the amount of binding information that needs to be specified within a system’s model. Hiding complexity through abstraction is of course a time honoured tradition within both software development and electronics engineering.

8. COLLECTION PORTS

Up until now it has been assumed that binding takes place between only two ports. In some cases it is desirable however to be able to bind a single source port to more than one destination port. This fan-out type pattern can be achieved using collection ports. The most obvious use of this facility is to provide a simple mechanism for broadcasting. E.g. a single call made to a service port that is bound to several destination ports results in multiple calls being made to each of these destinations. The caller does not need to be aware of the number of services actually invoked as a result of the call. This ensures the autonomous nature of the classes is maintained.

Collection ports can be used with all port types, even compound ports. During the original design of the Razor framework collection ports were treated differently to regular ports. It soon became clear however that any port has the potential to become a collection port providing the connectivity constraints are not broken, i.e. providing the signature of a source port is suitable for use with multiple targets, then that port can be bound to multiple targets. Consider the binding examples shown in figure 10 that highlight valid and invalid attempts at binding to multiple targets.

As can be seen from the graphical model, the logValue service port appears as a provided port within the FileLogger, but as a required port within the StatsCalculator. Hence, the directional nature of contained ports is inverted when the host compound port is used as a required, rather than a provided, port.

The necessity for compound ports becomes more apparent within complex systems, since they can substantially reduce the amount of binding information that needs to be specified within a system’s model. Hiding complexity through abstraction is of course a time honoured tradition within both software development and electronics engineering.

8. COLLECTION PORTS

Up until now it has been assumed that binding takes place between only two ports. In some cases it is desirable however to be able to bind a single source port to more than one destination port. This fan-out type pattern can be achieved using collection ports. The most obvious use of this facility is to provide a simple mechanism for broadcasting. E.g. a single call made to a service port that is bound to several destination ports results in multiple calls being made to each of these destinations. The caller does not need to be aware of the number of services actually invoked as a result of the call. This ensures the autonomous nature of the classes is maintained.

Collection ports can be used with all port types, even compound ports. During the original design of the Razor framework collection ports were treated differently to regular ports. It soon became clear however that any port has the potential to become a collection port providing the connectivity constraints are not broken, i.e. providing the signature of a source port is suitable for use with multiple targets, then that port can be bound to multiple targets. Consider the binding examples shown in figure 10 that highlight valid and invalid attempts at binding to multiple targets.
getTotal port returns a single integer value however ensures that it can only be bound to a single target port. Finally, the getValues port does have a return type, but because this is an array of values it can be bound to any other service that returns either a single integer, or an array of integers. In this case the result values will be automatically aggregated into the returned array.

The availability of collection port semantics provides a great opportunity to 'tap' port bindings between autonomous class instances. E.g. a special class can be easily attached to the external interfaces of class instances in order to provide monitoring information with regard to the usage of certain ports. This can be done without the knowledge of the autonomous class instance being monitored since the connectivity is undertaken outside the scope of the class itself.

9. RELATED WORK

The main body of work with which the Razor framework appears to have most similarities is the Fractal project [30] which began within the research and development department of France Telecom. The Fractal project aims to provide full support for design, implementation, deployment and reconfiguration of systems and applications. Many of the underlying software engineering principles championed by the Razor framework are also emphasised by the Fractal project. Compared to Razor however, the Fractal approach provides a somewhat heavier solution and is delivered as a much more complete component model.

The Razor framework primarily focuses on language level support for Component Oriented Development, whereas Fractal is programming language agnostic. Fractal models can be defined using an Architectural Description Language (ADL) [31], which is then supplemented with either Java or C/C++ code in order to provide the full implementation. A graphical notation is also defined within the Fractal project that provides a higher level means of generating the ADL. The development process within Fractal however can be achieved without the use of either the ADL or the graphical tool, since components can be implemented directly within a supported language. For example, the JULIA framework supports the construction of software systems with Fractal components written in Java [32]. In contrast to this the Razor autonomous classes are exclusively defined within the DOM, since the framework includes a component oriented language, rather than encapsulating existing languages.

Another well known development environment that shares certain aspects with the Razor approach is the Qt cross platform application and UI framework [33]. Qt is a C++ based framework that emphasises decoupling of classes using a signals and slots mechanism. Signals are emitted from objects when specific events occur, whereas slots provide functions that are called in response to such signals. During development signals are connected to compatible slots. This approach is similar to the port binding mechanism provided within the Razor framework, especially since compatibility is based on function signature rather than matching of names. Therefore Qt supports the concept of Signature Based Binding (SBB). Although Qt’s signals and slots are more restrictive than Razor’s port mechanism, the success of Qt is a strong indicator that decoupling of classes using such an approach is acceptable to developers and practical within applications development.

10. EVALUATION

Given the immaturity of the Razor development approach it has proved difficult to evaluate on a large scale. However, the reference implementation has provided an opportunity to provide an environment to allow the evaluation of the mechanics of the system. The counter example discussed earlier within the paper was the first system to be developed and executed within the reference run-time environment. In addition to this, a simple buffer type autonomous class was created. This proved very useful since it was possible to create a bidirectional buffer simply by reusing two instances of unidirectional buffers. It also provided an opportunity to test the signal type ports, which were used to indicate when the buffer contents reached a high-watermark level.

Although these systems are very simple, and used independently provide limited application, their mere existence and ability to be correctly configured; loaded; and executed shows that the suggested approach to be generally valid. Basing an evaluation on such simple implementations may seem trivial, but the philosophy of the Razor approach is to create complexity from simplicity. Hence the ability to implement simple autonomous classes using other autonomous classes as part instances, combined with the ability to connect these via their well defined interfaces is the real capability that needed evaluating.

The evaluation proved useful in terms of showing that the semantics of the approach are fundamentally correct. However the evaluation also showed that basing a system on the Java platform tends to lead to too much overhead during configuration and execution, especially when combined with the use of Javascript to provide support for procedural service implementations. This evaluation has thus influenced the next stage of development. Rather than create a portable run-time within Java the next generation of the Razor system is being developed using C++, with a purpose built procedural language being developed. The procedural language is a simple sub-set of the C grammar and is to be supported using a just in time (JIT) compilation approach, thus providing native execution speeds for called services. This approach allows for the removal of concepts provided in the Javascript language that are not required to support the semantics of the Razor approach, e.g. the ability to dynamically create objects and assign properties is no longer supported.
The evaluation has also highlighted several weaknesses with the existing approach. One such weakness was the inability to implement a full interface simply by binding a single port to a nested super-type part instance. This issue has been resolved by a slight redesign of the Razor grammar. The evolving language now restricts implementations to only providing compound type ports. This means that only entire interfaces are presented as ports within implementations, with non-compound type ports now always being nested within compound ports. Also an implementation now not only implements specific interfaces, but also expects the availability of specific interfaces via outgoing ports.

11. SUMMARY

The Razor Development Framework is an approach to software development designed to maximize reuse and to simplify the development effort. The use of reusable autonomous classes allows for a more declarative style of development, leading to support for a predominantly graphical development approach. Particular attention to core reuse principles along with contemporary development practices has driven much of the underpinning design.

The Razor Development Framework aims to extend the Object Oriented model to allow direct language level support for Component Oriented Development (COD). Additions to existing Object Oriented concepts include the following.

1. Signature Based Binding
2. Type Only Typing
3. Interface Retrofitting
4. Type Only Inheritance
5. Type Conformance Testing

The framework is Interface centric rather than Classifier centric, with ports playing a major role within the development of systems. The central artifact of the framework is a DOM that defines each autonomous class. This DOM can be populated in a number of ways using either a graphical notation or a textual Model Definition Language (MDL). Although these technologies are still under development they have reached a level of maturity sufficient enough to show that the Razor framework can indeed be used to create software systems.

12. CONCLUSIONS

The original aim of the research work was to develop a mechanism that allowed for component oriented development using a graphical approach. This aim appears to have been achieved, with the availability of a prototype reference implementation showing the approach to be a practical reality. The design of the framework has highlighted the fact that new approaches to software development should not only consider technical aspects, but should also consider the preferred development practices.

There are certain aspects of the Razor Development Framework that require more detailed discussion. The testing framework is a particular element of interest. A closer examination of generics and the typing system would also be of benefit to interested parties. Additionally, the intricacies associated with compound collection ports and synchronized compound ports have not been explored. There is also a large discussion to be had regarding the ability to include a reflective interface within the generated systems. Omissions of such details are unfortunately necessary within a small introductory text. The important point however, is that although these aspects are often complex in their underlying implementation, their application from a developer’s point of view is fairly simple.

The Razor framework has the potential to be extended in several areas. One concept not covered is the ability of the framework to provide language level support for the definition of finite state machines. Their declarative nature means that they can be used to represent the internal definition of an autonomous class. The ports act as stimulus events that cause state transitions to occur.

The existing definition of the graphical notation and the DOM needs to be better formalized into a specification, including mappings, constraints and target translation rules. Also, a full Software Development Kit (SDK) needs to be developed to provide tools such as profiling, tracing, debugging and automated documentation generation.

Overall the Razor Development Framework appears to offer an alternative approach for software systems development. The move towards declarative development using reusable autonomous classes should simplify the development process. The acid test of such techniques however is the level of adoption by developers, hence only time will tell whether the availability of such an approach will entice developers away from the existing development techniques. It should be recognized that the success of development approaches is likely to be closely linked to the available application deployment mechanisms. Strong support for both mobile applications and RIAs is going to be a key battleground within the software development arena.

REFERENCES


APPENDIX

(*
This grammar assumes that the input stream is already pre-processed. i.e. comments removed, white space removed, terminal symbols tokenized. *)

root = [package] , { import } , annotations , typeDef , [';'] ;
package = 'package' , qualifiedName , ';' ;
import = 'import' , qualifiedName , [identifier] , ';' ;
modifiers = ('public' | 'private' | 'synchronized' | 'static') ;
typeDef = { modifiers } , (interfaceTypeDef | implementationDef | stateMachineDef | enumTypeDef | namespaceTypeDef | testSuiteDef) ;
interfaceTypeDef = 'interface' , identifier , [templateDefs] , [extends] , [';', { annotations , ( inPorts | outPorts | typeDef ) , [';'] } , ')'];
implementationDef = 'implementation' , [identifier] , [templateDefs] , [implements] , [';', { annotations , (parts | portBindings | initDef | finalizeDef | ports) , [';'] } , ')'];
stateMachineDef = 'statemachine' , [identifier] , [templateDefs] , [implements] , [';', { annotations , (states | transitions | portBindings | initDef | finalizeDef) , [';'] } , ')'];
enumTypeDef = 'enum' , identifier , [extends] , [';', { annotations , identifier , [';'] } , ')'];
namespaceTypeDef = 'namespace' , identifier , [extends] , [';', { annotations , typeDef , [';'] } , ')'];
testSuiteDef = 'testsuite' , identifier , [templateDefs] , [extends] , [tests] , [';', { annotations , (initDef | finalizeDef | testDef) , [';'] } , ')'];
inPorts = 'provides' , [';', { annotations , portDef , [';'] } , ')'];
outPorts = 'requires' , [';', { annotations , portDef , [';'] } , ')'];
ports = 'ports' , [';', { annotations , [ 'synchronized' ] , portDef , [( implementationBinding | valueBinding | targetPortsBinding )] , [';'] } , ')'];
portDef = (servicePort|attributePort|signalPort|compoundPort) =
servicePort = ['service'] , (typeRef |'void') , identifier , [';', [paramTypeList] , ')' , [throws] ;
attributePort = ['attribute'] , typeRef , identifier ;
signalPort = 'signal' , (typeRef |'void') , identifier ;
compoundPort = 'compound' , userType , identifier ;
initDef = 'initialize' , [';', [paramTypeList] , ')' , [throws] , statementBlock ;
finalizeDef = 'finalize' , statementBlock ;
testDef = 'test' , identifier , statementBlock ;
parts = { annotations , instanceDef , ';' } ;
states = { annotations , instanceDef , ';' } ;
transitions = 'transitions' , [';', { annotations , transitionDef , [';'] } , ')'];
instanceDef = ['static'] , implName , identifier , ['(', [exprList] , ')'] ;
transitionDef = [ triggerDef ] , '->' , (identifier | '*') , [ conditionExpression ] , [ statementBlock ] ;
triggerDef = portRef , ['&', portRef ] ;
portBindings = 'bindings' , [';', { annotations , bindingDef , [';'] } , ')'];
bindingDef = ['synchronized'] , qualifiedName , [( implementationBinding | valueBinding | targetPortsBinding )] ;
implementationBinding = statementBlock | 'native' ;
valueBinding = '=' , expression ;
targetPortsBinding = '<>', portRef, ( { '&', portRef } | { '!', portRef } );

portRef = ['synchronized'], qualifiedName, [conditionExpression];

typeRef = 'const', typeName;

typeName = ( primitiveType | userType | 'var'), { '[', ']'};

primitiveType = ( 'byte' | 'short' | 'int' | 'long' | 'float' | 'double' | 'boolean' | 'char' | 'string' | 'native' );

userType = qualifiedName, [templateValues];

implName = qualifiedName, [templateValues];

paramTypeList = formalParam, ['=', expression], {',', formalParam, ['=', expression]};

formalParam = typeRef, identifier;

exprList = expression, {',', expression};

tests = 'tests', userType;

extends = 'extends', userType, {',', userType};

implements = 'implements', userType, [alias], {',', userType, [alias]};

alias = identifier;

throws = 'throws', userType, {',', userType};

templateDefs = '<', templateParam, {',', templateParam}, '>';  

templateParam = identifier, ['extends', userType {&', userType}];

templateValues = '<', templateArgument, {',', templateArgument}, '>';  

templateArgument = typeName | '?', ['extends' | 'super'], typeName;  

annotations = { '@', qualifiedName, [] };  

annotationValuePairs = [identifier, '='], annotationValue, {',', [identifier, '='], annotationValue};

annotationValue = literalValue, [annotationValue, {',', annotationValue}];

qualifiedName = identifier, {',', identifier};


value = literalValue, [newName, implName, [type, [exprList], type, [type, type]], qualifiedName, [value, [value, value]], ''];

literalValue = ( stringLiteral | charLiteral | integerLiteral | booleanLiteral | floatLiteral | 'null');

stringLiteral = "", { ? any printable character ? - " | backslashEncoding }, "";

charLiteral = "", { ? any printable character ? - ' | backslashEncoding }, "";

integerLiteral = ( ? '1'..'9' ?, { ? '0'..'9' ?}, '0x', { ? '0'..'9' ? | ? 'a'..'f' ?}, { ? '0'..'9' ? | ? 'a'..'f' ?}, [ '(' | 'L' ]);

booleanLiteral = ( 'true' | 'false');

floatLiteral = [0..9], { [0..9], '.', [0..9], [.0..9]};  

backslashEncoding = 'n', ( (n|t|b|r|f) | [""\""|""\"""] | '0'..7 ? , [ ? '0'..'7 ? ] | ? '0'..'3' ? , [ ? '0'..'7 ? ], [ ? '0'..'7 ? ]);  

conditionExpression = [', expression, ']';

expression = value | ? ECMAScript expression ?;

statementBlock = ['?', ECMAScript code ?];

ACKNOWLEDGEMENTS

The author wishes to thank Julian Old and Ken Tindell for the very useful discussions regarding many aspects of this work.

AUTHOR INFORMATION

Mark.B.Dixon achieved a B.Sc. in Computing (1994) followed by a Ph.D. in Computer Science (1997). His employment history includes working as a software engineer for Dabs Press (now Dabs.com); an embedded software engineer at Taylor Nelson Sofres (TNS - London); a real-time systems software engineer at Live Devices (York); and as a consultant in software engineering and model based development. Current research interests include software engineering, model based development and embedded systems development. Dr. Dixon is currently employed at Leeds Metropolitan University in England and is responsible for the leadership of the postgraduate Computing courses; he also undertakes Ph.D. supervision and various research activities.