Developer's Introduction to Resource-Oriented Computing

1060® Research white paper series:

Developer's Introduction to Resource Oriented Computing

Version: 1.0

July 12, 2007

Introduction

Resource-oriented computing (ROC) is a new and very different approach to building software. It is simpler and results in software that is more flexible, less complex, uses fewer lines of code, and generally runs faster than software built using conventional approaches.

This white paper introduces developers and architects to resource-oriented computing. A higher-level, less technical introduction is available in the "Manager's Introduction to Resource-Oriented Computing". A rigorous introduction touching on the fundamentals of the computer science foundation can be found in the series "Introduction to Resource-Oriented Computing" part I, part II and part III. These white papers are published on the 1060 Research website at http://1060research.com/netkernel/whitepapers/ .

Most people in the IT industry sense a problem with software. Despite decades of technological progress, software remains expensive and development projects are still prone to failure. In fact, a recent study shows that more than 50% of software projects are late and, worse, many are failing outright:

> "The outcomes of IT projects have been slipping, with 57% of respondents saying that fewer than half of the IT initiatives in their firms had a positive outcome."[1]

At the same time, some software systems are wildly successful. The World Wide Web - the largest information system ever devised - works remarkably well. Web sites can be added without disrupting the whole, sites can grow or shrink, come on-line, go off-line, or even be archived[2].

At 1060 Research we have spent the last eight years investigating the properties of the Web that have made it successful and devised a computing model that imbues software with the same characteristics. We call this new computing model *resource-oriented computing*. A full implementation of ROC is available[3] in 1060 NetKernel. The ROC coding examples used in this paper are based on NetKernel.

After reading this white paper you will have a working understanding of resource-oriented computing and how it compares with conventional approaches. You will also understand how ROC results in software that is more flexible and can scale with CPU cores and which, in fact, has the essential characteristics that has made the Web successful.

1 BBC News (June 4, 2007): http://news.bbc.co.uk/1/hi/business/6720547.stm

2 The WayBack machine at http://www.archive.org contains a historical backup of the World Wide Web.

3 A free download of NetKernel is available at http://1060research.com

Resource-Oriented Computing

Resource-oriented computing is, not surprisingly, about *resources*. Resources provide a means to model information such as the closing price of a stock, a purchase order, the current temperature recorded by a weather monitor, a list of conference attendees, a photo of an insurance claim.

Resource-oriented computing (ROC) is about processing resources and works at the same level as information. This is vastly different from current approaches that work with low-level data and objects. ROC is a *logical* computing model not a *physical* model found with Java, C, C++, Ruby, and other current languages.

In ROC each resource is identified by one or more logical identifiers. As you will see, the Uniform Resource Identifier[4] (URI) is a particularly good way to refer to resources. If the term URI is new to you, think about the Uniform Resource Locator (URL) used by the World Wide Web, a URL is a type of URI. For example, typing the identifier http://1060research.com into a browser causes the browser to issue a request for the desired resource (in this case the home page of a web site).

Identifiers

Before we explore resources in detail, we need to examine the use of a logical as opposed to a physical identifier for information within a program. Programming languages today (we will use Java as the example language) use physical identifiers. For example, this line of code:

```
counter = counter + 1;
```

uses a named reference to a variable ("counter") and increments its value by one. While it appears that we are referring to data with a logical identifier, in fact when the program runs, we are not, we are using a physical identifier. When this program is compiled into operation codes ("op-codes"), the textual identifier "counter" is replaced with a memory reference to a specific physical memory location. The Java virtual machine opcode that increments local variable 0 by 1[5] is:

```
iinc 0 1
```

Once the resolution from textual identifier to physical location is done it is not repeated and the program runs exclusively using physical addresses. In this example, the variable "counter" will resolve to and will always be known as the "variable at offset zero within the Java method". This approach results in a procedural program that runs fast as no additional logical-level address interpretation is required – everything runs at the physical level.

A program in a resource-oriented computing system uses a different approach. All resources are

4 Wikipedia: http://en.wikipedia.org/wiki/Uniform_Resource_Identifier

5 This assumes the the variable "counter" is the first declared local method variable and is of type "int".

identified using a logical identifier that remains logical. For example, the following URI[6] might refer to the closing price of IBM on June 15, 2007:

```
ffcpl:/stocks/closing/2007-06-15/IBM
```

When a ROC program needs information it issues a request using a verb[7] and a logical identifier. The verb "source" means to return the current representation of a resource, so a request for the closing price of IBM stock would be the verb and identifier pair ["source", "ffcpl:/stocks/closing/2007-06-15/IBM"]. Note carefully that this request *is the op-code* for an ROC system just as "iinc 0 1" is the op-code for the Java runtime engine. We will later explore the importance of this op-code being represented as a text string.

This is analogous to the use of the HTTP "GET" method[8] and the logical address "http://1060research.com" by a browser. In the World Wide Web, the logical identifier is resolved at the time of the request using several services (DNS, a web server, etc.) and the representation is computed by a web server. The web server may return the contents of a file, execute code, or take another action – all of which is not visible at the logical level. The result of these steps is a resource-representation being sent to the browser (in this case an HTML formatted document) for display to the user.

In a ROC computing system, the request is sent to an intermediary that resolves the identifier address within an address space and locates code (known as an "Accessor") that knows how to compute a representation of the resource, and finally returns the resource-representation. Just as with the web server example, an accessor may return the contents of a file, execute code, or take another action – all of which is not visible at the logical level.

	Logical (Web)	**Logical (ROC)**	**Physical (Java)**
Identifier	URL	URI	Physical memory location
Retrieve	HTTP "Get" + URL	"source" + URI	Direct memory read
Resolution	DNS & web server	micro-kernel	Compiler & Linker
Compute	Web server	"Accessor" endpoint	machine code instructions
Op-code	HTTP method + URL pair	verb + URI pair	machine code instruction

In summary, a crucial difference between ROC and physical computing is the use of indirection through logical addresses instead of direct access to physical memory data locations. This indirection comes with the cost of resolving logical identifiers for each request. As we will see in more de-

6 The "ffcpl: scheme in ROC is similar to the "file:" scheme – it is used to locate a resource within a hierarchical address space.

7 The verbs in ROC include source, sink, delete, new, exists, and meta.

8 The HTTP protocol calls its verbs "methods". The HTTP methods include GET, PUT, POST, HEAD, etc.

tail later, this indirection induces much needed flexibility in software. We will also see that the small performance cost of indirection is offset, and in fact repaid with interest, by the use of an intelligent cache, just as the Web uses proxies and caches to improve its performance.

Resources

Next we examine resources and see how they differ from physical level concepts such as data and objects. In ROC a resource is an abstract idea that only becomes concrete when it is requested. As we have seen, a request is resolved to a software endpoint (Accessor) that is responsible for computing the resource's physical representation each time it is requested.

In a physical computing model, each data item or object has a specific *type*. For example, numeric values in Java could be of type int, Integer, float, Float, etc. Type is important at the physical computing level as it disambiguates the binary encoded information pointed to by memory references and helps the computer perform correctly and efficiently.

We will examine the difference between a resource and physical level typed data by looking at a specific example. Here is Java JDBC code that accesses a database to get the closing price of IBM:

```
RecordSet rs;
Statement stmt;
StockPrice stockPrice;

rs = stmt.executeQuery("select * from prices where symbol='IBM' and date='2007-06-15'");
while (rs.next())
  {
  stockPrice = new StockPrice();
  stockPrice.setSymbol(rs.getString("symbol"));
  stockPrice.setPrice(rs.getInt("price"));
  stockPrice.setDate(rs.getDate("date"));
  }
```

Clearly, knowing the type of data is critical to writing this code. One must know the type of data returned from the query and the type of the parameters in the "set" methods in the StockPrice class (setSymbol, setPrice, setDate). In this example the data types returned by the database match the types required by the StockPrice class so they may passed directly to the methods. If they did not, then extra code is required to convert the type. Not only must a developer have all of this information available to write the correct code, if the type of the data changes (either the database is changed or the StockPrice class changes), then this must be re-coded, tested, debugged, released, distributed, and the application restarted. This physical level code example is *brittle* with respect to the types of data used – a small change breaks the code and may require extensive changes.

Let us now examine how a ROC system would be constructed to retrieve the same information. First, however, we must introduce the *active* URI scheme. In ROC the active scheme is used to

specify services and their parameters. In Java, a service named “average” might have a method signature:

```
public int average( Set values )
```

and would return the average of the values computed from the passed values. In ROC one would call a similar service using the active scheme:

```
active:average+values@ffcpl:/collection
```

The service named “average” expects a parameter named “values” which is set to the URI “ffcpl:/collection”, a resource that contains the values of interest. When the active URI is requested a representation with the average of the values in “ffcpl:/collection” is returned.

Getting back to our main example, a database query service included with NetKernel called “sqlQuery” is used to retrieve information from a database. The equivalent ROC code to return the representation of the closing price of IBM is[9]:

```
active:sqlQuery+operand@”select * from prices where symbol='IBM' and date='2007-06-15'”
```

which returns a representation that is the following XML document:

```
<results>
  <row>
   <symbol>IBM</symbol>
   <price>103.88</price>
   <date>2007-06-15</date>
  <row>
<results>
```

This ROC program fragment is coded at the logical level and contains no type specific code; it is therefore *not brittle* with respect to the above mentioned data types changes. Notice that the only encoded information – the name of the database table in the query and the column names within the XML document – are all logical. Relational database are themselves a logical computing model specifically design to manage persistently stored information. Using ROC to work with relational databases eliminates what is called the “impedance mismatch”[10] between relational information and objects.

There is something else interesting to note. Recall that a URI in ROC is an identifier. However, the active URI scheme is also expressing a functional program within the URI. In this example, the call to the sqlQuery service along with the SQL query to execute. This duality – an identifier that is also a functional program – allows for something very powerful: a *cache can use the URI as the key and store the identifier's resource-representation as the value.* This means that the program “active:sqlQuery...” can be associated with the resulting XML document. Each time a re-

9 This syntax is not precisely correct but serves to illustrate the concise nature of ROC code.

10 An enormous amount of intellectual effort has been expended by the industry to address this “impedance mismatch” and many products are available (such as Hibernate, EJBs, etc.) to address the problem. By using ROC the whole problem disappears.

quest is made in an ROC system, the cache can be queried using the URI as the key. If the cache has the key, the cached value is returned and no additional work is required – entire functional programs need not be run repeatedly[11]. Caching can be used with regular physical level code, but a cache at that low-level requires a physical connection with the objects it is going to save and it is essentially impossible to anticipate what to cache in a complex system. In ROC URIs are used pervasively and the cache has knowledge about all requests – from coarse to fine-grained. Because the system knows the cost to create each resource-representation (all requests go through a micro-kernel and it can monitor the cost to create a representation), the cache can retain the most valuable representations. This means that not only are redundant computations eliminated, a whole system running on an ROC platform can dynamically self-tune based on the workload[12].

But, we are getting ahead of ourselves. In our example, the sqlQuery service returns an XML document. What if the client making this request does not consume XML documents and instead requires a StockPrice object[13]? If a client needs the representation returned as a specific type, that type can be included in the request. In this case the request would be ["source", "active:sqlQuery...", "StockPrice"] as a triplet. When issued, the micro-kernel notices that the type returned by sqlQuery (an XML document) does not match the type requested (a StockPrice object). To address this mismatch, the micro-kernel can attempt to transform the representation (we call this *transrepresentation* or *transreption* – the lossless conversion from one representational type to another). The micro-kernel cannot do this transreption itself, instead, it issues a request into the address space looking for a *transreptor* that can convert an XML document into a StockPrice object[14]. When located, the micro-kernel requests such a conversion and then returns the StockPrice object in the response to the original request. Just as with other representations, the StockPrice object can be cached for later reuse using the same cache key[15]. It is important to note that all of this type conversion work occurs at the physical level and is not visible at the logical level.

In summary, the resource model abstracts information at the logical level. URI identifiers for resources are resolved for each request and the overhead of the indirection can be compensated for by using a system-wide cache. By using ROC and resources instead of data and objects that the physical level, a system becomes more flexible and is resilient to data type changes in the physical level.

11 In practice the cache in an ROC system more than makes up for the overhead cost of the logical address indirection.

12 There are other optimization benefits from ROC, some of which are discussed in Fibonacci algorithm analysis white paper.

13 In our experience using XML as the internal means of transferring information yields a more flexible system without much additional overhead compared to the use of Java objects.

14 A transreptor that converts an XML document into a StockPrice object would be written by a development team member and be made available to the ROC system.

15 The cache can save multiple representational types for each URI key.

Architectural Layers

ROC code is not brittle with respect to type changes and conversions. However, a logical change would require some sort of modification. A common architectural approach to managing the risk of such change is to isolate and carefully manage those parts of a system that may change. One technique is the use of layers.

The logical resource address space in an ROC system can be partitioned into modular units. Each module will house related resources and services. The module is a container for a logical URI resource address space. In order to compose manageable applications one module may dynamically import the publicly exposed resources and services of another.

Continuing with our example of the closing stock price of IBM, recall that the URI identifier is "ffcpl:/stocks/closing/2007-06-15/IBM". This URI is very stable – there is little reason to change it short of a complete redesign of the logical information structure. However, the URI that calls the sqlQuery service and retrieves the representation could change if the database changes. We therefore want to use the second URI to retrieve the representation of the first URI and isolate the second URI into a lower layer of the architecture. Both can be done easily with URI mapping.

A URI map is defined between two URIs in a module's configuration and is fully managed by the micro-kernel. For example, the following mapping[16]:

```
ffcpl:/stocks/closing/2007-06-15/IBM
                |
                |
                v
active:sqlQuery+operand@"select * from prices where symbol='IBM'and date='2007-06-15'
```

will cause the micro-kernel to watch all requests issued within a module for the existence of the first URI. When the micro-kernel detects the presence of the first URI, it automatically issues a sub-request using the second URI. The result of the sub-request is returned as the result of the first request[17]. This mapping creates a great deal of stability in the upper layer as any logical changes in the database can be isolated in a lower level module. For example, if the name of the table changes, the following rule can be substituted to keep the original URI valid:

```
ffcpl:/stocks/closing/2007-06-15/IBM
                |
                |
                v
active:sqlQuery+operand@"select * from stock-prices where symbol='IBM'and date='2007-06-15'
```

There is more to mapping than simple one-to-one definitions. Mappings can use regular expres-

16 The syntax used to declare mappings is not shown to keep this discussion general.

17 A more detailed explanation would illustrate that the first URI is used in an upper layer module and the mapping would be included in a lower-level module which is imported into the upper level module. All coding using the first URI would exist in the upper level module.

sion pattern matching. For example, the following mapping:

```
ffcpl:/stocks/(.*)/(.*)/(.*)
              |
              |
              v
active:fetchStockPrice+type@$1+date@$2+symbol@$3
```

will cause three matches (in parenthesis) in the first URI to generate match variables $1, $2 and $3 which are used in the second URI declaration. This effectively translates a RESTful resource request into a parameterized service request. For example, the URI

```
ffcpl:/stocks/closing/2007-06-15/IBM
```

would be detected and the URI

```
active:fetchStockPrice+type@closing+date@2007-06-16+symbol@IBM
```

would be used in the substitute sub-request. Presumably the code implementing the "fetchStock-Price" service would then issue a sub-request into the lower layers to get the representation.

URI mappings are possible because URIs are textual. Each URI is a string, a sequence of characters, and as such, URIs can be manipulated at the logical level of the ROC application. And because the URI is the key part of the verb + URI op-code, this means that op-code re-writing is possible within the operational model. The power to re-write instructions is a capability that object-oriented developers could only dream about! In ROC this can not only be done dynamically, the mapping rules themselves are resources that can be changed as the application runs.

As a final example, layers can be used to support the rapid development of an application in separate pieces. For example, the following mapping:

```
ffcpl:/stocks/closing/.*/IBM
               |
               |
               v
ffcpl:/IBM-example-data.xml
```

maps all requests for IBM closing stock information to a static resource (a single physical file with XML sample information). This allows development of the upper layer code to proceed using a static lower level while other developers independently create, test and then release the lower level. Because the isolation between layers is based on a logical mapping there is no cross-talk, no interference between the two development tasks.

Stateless Services

A design practice of using stateless services leads to flexible, high-performance systems. Stateless service design is a major reason the Web scales to a global information application. As the name implies, a stateless service retains no state between invocations. All required state is provided either through parameters or information in the environment. Each request from a Web browser

can be processed by *any* web server associated with the target address. Large web sites can use a load balancer to dispatch incoming requests to a pool of computers running web servers. Since it doesn't matter which server processes the request, the load balancer can select the least busy server. The size of the computer pool can be adjusted to match anticipated traffic (such as increases due to a marketing event).

A good example of a stateless service in ROC is the sqlQuery service. When it starts to execute it has no saved state. It gets the first portion of state by reading the value of the operand parameter and the second portion by issuing a request into the surrounding address space for the resource "ffcpl:/etc/ConfigRDBMS.xml". Let's look at this in detail using the following invocation:

```
active:sqlQuery+operand@ffcpl:/query.xml
```

In this example, the representation of the resource "ffcpl:/query.xml" is the XML document:

```
<sql>SELECT * FROM table;</sql>
```

The representation received when sqlQuery requests the resource "ffcpl:/etc/ConfigRDBMS.xml" is the XML document:

```
<config>
  <jdbcDriver>com.mysql.jdbc.Driver</jdbcDriver>
  <jdbcConnection>
    jdbc:mysql://localhost/mydatabase?user-myusername&amplpassword=mypassword
  </jdbcConnection>
</config>
```

With these two pieces of state information the sqlQuery service is able to interrogate the database and return the requested representation.

As a stateless service, sqlQuery requests its configuration each time it is called which means the service can be reconfigured while the application runs. This flexibility is important for at least three reasons – thread assignment flexibility, logical / physical level separation and operational flexibility. A stateless service can run on any thread or CPU core allowing an ROC system to scale with the number of available cores. In addition, the configuration of sqlQuery can be based on either static or dynamically generated information. How? Simply by using mapping rules. Since sqlQuery always requests its configuration from the resource "ffcpl:/etc/ConfigRDBMS.xml", a mapping run such as the following would return static information:

```
ffcpl:/etc/ConfigRDBMS.xml
               |
               |
               v
ffcpl:/source/MySQLConfig.xml
```

while the next mapping could generate a different configuration for each request:

```
ffcpl:/etc/ConfigRDBMS.xml
                |
                |
                v
active:generateConfig
```

Here, the generateConfig service could gather information by issuing a remote request to a configuration server (via HTTP for example), examining a mode configuration file that tells the system whether its in test or production mode, etc.

This flexibility is great, but what about performance? Performance is ensured with caching and transreption. When sqlQuery issues the request for the resource "ffcpl:/etc/ConfigRDBMS.xml" it specifies that it wants the representation in the form of a database connection pool. Because the resource is provided as an XML document, the micro-kernel must locate a transreptor that will create a database connection pool based on the information. Once converted, the connection pool is cached and is returned for subsequent requests. None of these optimizations are evident at the logical level; there is no intrusion into the business logic from lower-level caching or optimization code. This is markedly different from a pure physical model where such separation is hard to achieve and maintain through the life of an application.

Stateless services and a RESTful design approach allow ROC applications to scale with CPU and cores just as large web sites (such as Google) can scale using a IP load balancer in front of a large server farm.

Modularity and Isolation

While we have seen that resources in an ROC system are identified by a logical URI we have not said much about the address space in which these URIs are resolved. In the World Wide Web there is only one global address space. The URL "http://1060research.com" is evaluated the same in the US, Brazil, France, etc[18]. In ROC there can be an infinite number of address spaces and these address spaces can have relationships between themselves. In ROC, each address space is the *information context* in which resource identifiers are resolved and accessor endpoints are located and executed.

The physical container for an address space is a *module*. A module is the unit of distribution (each module has a unique name and version number) and contains its own resource loader[19], accessors, transreptors, resources, and internal private address space. Each module can elect to expose or export a portion of its private address space. For example, the following export rules exposes a portion of the internal address space and a specific service.

18 Some countries may elect to block such requests or intercept them and provide a different result, but that's another topic.

19 And Java class loader.

```
<export>
  <uri>
    <match>ffcpl:/public/.*</match>
    <match>active:myService</match>
  </uri>
</export>
```

Modules may import the exported portions of other module's address spaces. For example:

```
<import>
  <uri>urn:org:ten60:netkernel:mod:db</uri>
</import>
```

imports the DBMS services including active:sqlQuery.

Modules, separate private address spaces, imports, and rewrite rules provide capabilities at the logical level that do not exist at the physical computing level. These lead to a new set of patterns and approaches to building software systems in even more flexible and malleable ways[20].

Construct, Compose, Constrain

Resource-oriented computing helps developers focus on information. In most applications the core computing tasks involves creating, updating, and retrieving identified information, transforming related information, and finally preparing a presentation for the user.

In ROC there are three phases of development – *Construct*, *Compose*, and *Constrain*. In the construct phase, new physical level services and resource models are created. For example, programming a transreptor to convert a XML document to a StockPrice object is a construct phase activity. The composition phase is focused on building an information model and wiring services together to perform the logical work required of the application. In ROC we find that the composition phase takes the majority of the development time. The final constrain phase involves the application of restrictions at various levels such as field-level validations. In ROC constraints can be applied last, after the system is proven to work properly.

Construct

Work during the construct phase most closely resembles traditional physical level programming. That should not be a surprise as the construct phase focuses on building extensions to the ROC system to support the contemplated information model.

For example, if an application is to process images and the ROC system does not yet support that resource type, then a set of representations, services, and transreptors will be required. Let's pick the Java2D model for the representation. The following items could be built to support this new ecosystem:

20 Further discussion of these patterns and approaches is beyond the scope of this white paper.

- Standard formats to 2D image transreptors (such as to and from PNG)
- services such as “rotate”, “dither”, “scale”, etc.

Once these are developed, then the lower-level work is done and work can proceed at the logical level with the composition phase.

It is important to note that the NetKernel ROC system includes a vast array of resource models, services, and other capabilities. Many types of applications can be built without any work done in the construct phase.

Compose

During the compose phase the high-level information model is defined and the application is wired together using resources and services. Continuing with the image example, the following is now possible at the logical level:

```
active:rotate+angle@15+image@ffcpl:/pictures/boxes.png
```

If constructed to support it, the new resource model can be intermixed with existing resource models such as an XML model that creates XHTML pages.

Constrain

The application of constraints to a working ROC system is straight forward. Because an ROC application is built from layers of mappings, constraints can be added in between layers and address spaces. For example, to use a security gatekeeper, the following mapping is added to an application:

```
(active:.*)
       |
       |
       v
active:gk+uri@$1
```

The service “gk” is a gatekeeper which reads its configuration information from the resource “ffcpl:/GateKeeperPolicy.xml”. This mapping will cause all requests for services to be sent to the gatekeeper service along with the original URI as the “uri” parameter. The gatekeeper can then examine the request and its context for security credentials and then either reject the request or issue a sub-request using the original URI.

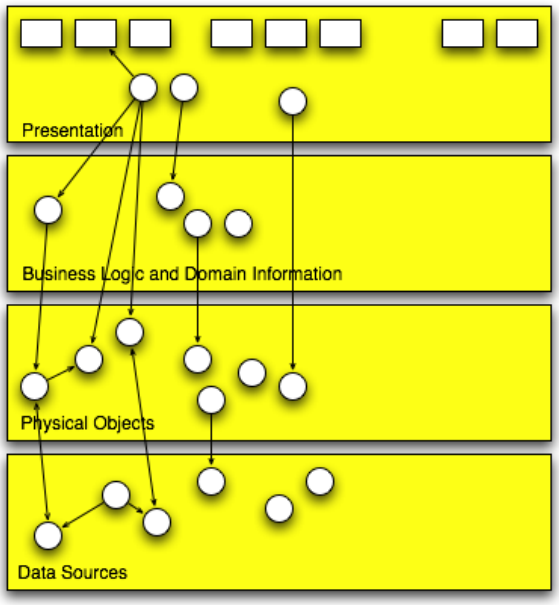
Object-Oriented Computing Comparison

While Resource-oriented and object-oriented (OO) computing emerge from different heritages they both share a common anchor in the fundamentals of computer science[21]. ROC does not displace OO it relies on it and augments it. In fact ROC and OO are complementary technologies.

Object-oriented technology emerged from decades of advances that started with assemblers, compilers, modular and then object technologies. And progress has not stopped with OO. Aspect-oriented programming (AOP) now augments OO with modularized and cleanly separated areas of concern. The Pattern movement added a layer of intellectual abstraction on top of OO that enables clean multi-tier physical computing model designs. And new niche technologies such as Spring offer a degree of decoupling and container management at the physical level. All of these steps add capabilities at the physical level to support more advanced computing abstractions.

ROC also has a rich heritage – Unix, the World Wide Web, and other systems are early examples of resource-oriented computing systems. ROC is a pure logical computing model that is implemented by a software operating system – a micro-kernel based core – that sits at the boundary between the logical and the physical models.

In this paper we have shown the basics of the ROC logical model and now we will further explore it by comparing a business system designed using OO and ROC.



Physical Architecture

This diagram depicts a four-tier architectural design for a physical computing system which includes a presentation, business and logic, data, and data source tiers. Tiers are used as an architectural organizing abstraction to increase modularity[22]. In this example, each tier is focused on a separate architectural area of concern and within each tier, best-practice object-oriented design and coding practices are presumed[23] – such as the use of patterns (Factory, DOA, Proxy, etc.).

21 ROC and OO are both grounded in Turing's papers written in the 1930s. This foundation is explored in other white papers.

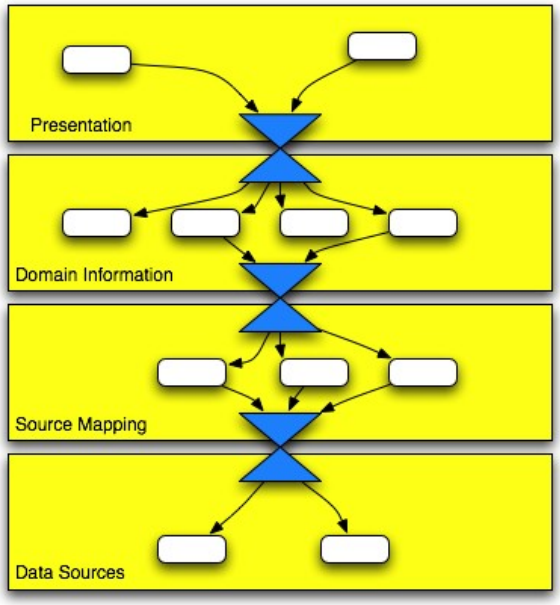
22 Modularity is a software organization concept in which related items are closely associated and non-related items are not present. A "module" can also expose a restricted interface through which non-related software artifacts use those within a module. The goal of modularity is to reduce overall system complexity.

23 There are a wide range of technology choices for each tier. In the presentation tier Struts, JSF, JSP/Servlets, Tapestry and other technologies support the creation of browser based (HTML) presentations and the management of user interactions. At the data

This design is typical of a Java, J2EE, .Net, etc. application structure. Note carefully the physical memory references between various objects that span into and across the architectural tiers. These links exist in part because they form a path across which objects travel carrying *the information that needs to flow from the sources tier to the presentation tier*. This is a critical point – while a computer system's value stems from information management, at the physical level, all that is available are data, objects, and memory references. Information and objects are *not* the same. Information is a logical concept, data and objects are physical entities. This four-tier physical architecture enables information (at the logical level) to flow to the user through the use of objects (at the physical level). In other words, the physical artifacts of this architectural design enable and facilitate the flow of information valued by the system users.

Logical Architecture

In contrast with the physical architecture, a logical architecture deals with information and information channels directly. This diagram depicts a logical ROC architecture design using four tiers and illustrates the logical relationships between resources. The blue funnels located between the tiers represents address space mappings implemented with module import/export and mapping rule capabilities. All links are logical, all are evaluated for each request, and all provide points of flexibility. Logical links allow change to be easily managed. While a change in a physical link requires recompilation, etc. logical links can be added, updated, or removed while the system is running, just as changes to the Web do not disrupt the overall operation of that information based system.



Discussion

The NetKernel ROC system not only provides a clean logical model, for the construct phase it provides all of the capabilities that OO developers long for – a clean AOP model, advanced container management, hot deployment and live system management, asynchronous thread management,

source tier, technologies such as JDBC, Hibernate, and Enterprise Java Beans (EJB) make it possible to connect to relational databases. Other technologies can be used to connect with other sources of data. At the physical data object tier objects are generally the technological artifacts of the technological choices made for the data source tier. If Hibernate is used in the data source tier, the POJOs (Plain Old Java Objects) can be used. If EJBs are used the the EJB proxy objects are used. If JDBC is used in the data source tier then factories, DAOs and other approaches are a better fit.

etc. The NetKernel ROC product is advanced enterprise software infrastructure.

OO developers will feel very much at home writing extensions for an ROC system and will be pleased to find that they are freed from writing plumbing code and can work with information channels directly. Because there is less code to write, OO developers can focus on building high-quality additions that will be used by the rest of the development team.

In Summary, OO and ROC are complementary technologies. Many of the features and benefits sought by OO designers and developers are a natural part of the ROC model. OO developers can focus on building extensions to the ROC model to enable other developers to leverage their work building complete system.

Summary

Resource-oriented computing is a logical computing model that rests above the physical computing level. URI identifiers are used to reference all information, services, etc. down to the finest level of granularity. Resolution of addresses within an address space occurs for each request and the binding to the physical level occurs for each element as the program executes. The use of indirection through logical addresses induces a high degree of system flexibility while the use of a system-wide cache, transreption, stateless services, and an operating-system caliber micro-kernel results in performance that surpasses those developed with traditional technologies and approaches.

For additional information about Resource-Oriented Computing, including ROC architectural consulting services and training, please contact 1060 Research at www.1060research.com