Introduction to EGL

Rational. software

Agenda

- Why IBM created EGL
- EGL Overview
- Discussion

Typical Issues facing IT

- Computing environment complexity
  - Heterogeneous platforms and middleware
  - Productivity decreases as complexity rises
- Skill sets restricted in platform silos
  - Newer skills cannot be easily applied to mainframe
  - Businesses unable to efficiently apply existing skills to new kinds of applications
- Platform and language choices intertwined and neither language plays well in the other environment
  - J2EE means Java
  - CICS means COBOL/PLI

How does EGL address these issues?

- Provides a higher level of abstraction that cuts across platforms and middleware
  - Say less. Do more - addresses productivity
  - Specific platform and middleware concerns handled by compiler/generator/transformation engine
    - Much less knowledge required to build applications for given platforms
    - Allows new programmers to learn modern language and efficiently build applications for mainframe systems as well as new
    - Single programming model for mixed workload style applications
    - Allows flexibility of changing out platforms and middleware later

Why are we here?

- EGL is simple, modern and extensible
- It has been influenced by COBOL, CSP, Java, J2EE, MDA and SOA
  - Stereotypes from UML
  - Service Components from Service Component Architecture (SCA) and SOA
  - Annotations from Java 5
- EGL is today a proprietary 4GL which has traditionally been an inhibitor to widespread adoption
- Customers are asking us to standardize EGL
- ADM seems an appropriate place to start
  - EGL integrates well with legacy languages and platforms
  - SOA concepts modeled directly in the language
  - IBM actively working with partners now to migrate legacy systems into EGL

General approach

- Provide a simple core language
- Provide a way to tag language elements with meta data
- Use these tags to represent complex semantics
  - Mapping Types to a Database
  - Binding of data to UI elements with validation and formatting
- Allows programmer to simply state semantics without forcing platform or middleware implementation choices
  - Same meta data can be applied in multiple contexts
- Transformation engine understands how to use meta data in mapping to a given runtime
  - Target language and platform leveraged to implement the defined semantics
- Conceptually similar to UML tags and stereotypes when used in transforming models into code

EGL Basic Concepts

- Statically Typed Procedural Language
  - But the “procedures” are componentized into Libraries and Services
- Built-in types to handle the
  - Old
    - Padded fixed length text data
    - Fixed point numeric types
  - And the (relatively) new
    - Unicode String – variable length
    - Reference types – Any, Dynamic array, Dictionary
    - Nullable value types
    - Date, Time, Timestamp, Interval
- Dynamic Memory and Garbage Collection
- Exception Handling
  - Similar semantic to Java exception handling except no checked exceptions

EGL Basic Concepts

- Various language components that programmers can define
  - **Data Item** - base types with annotations
  - **Record** - aggregate of typed fields into new type
    - Fixed overlay model onto contiguous storage – as in COBOL copy book
    - Tree - Can contain reference variables
  - **Program** - Main program starts run unit – used for batch processing
  - **Library** – Static set of data and functions – local to run-unit
  - **Service** – Remotable set of functions
  - **Interface** – Abstract definition of a Service
  - **Handler** – Handles interface to UI frameworks – JSF, Jasper Reports, etc
  - Some other component kinds
- Packages
  - Similar namespace management as Java package
  - All component definitions exist in some package.
- Annotations and Stereotypes
  - Meta data applied to program elements
  - Drives transformations as well as tooling

DataItem

- A DataItem is a definition that names an EGL base type along with associated annotations
- Field declarations that reference a DataItem as its type will automatically pick up the annotations appropriate within the context of the given declaration.
- Commonly used to define “Data Dictionaries”

```
DataItem SSN char(9) {
      displayName=“Social Security No”,
      pattern=“XXX-XX-XXXX”,
      OnValueChangedFunction=“ValidateSSN”,
      columnName=“SSN”  // SQL columnName mapping
              … }
end
```

Record

- Records are named aggregations of typed fields into a new type

```
Record CustomerRec
        custId CustomerID;
        ssn SSN;
        firstName String;
        lastName String;
        homeAddress Address;
        workAddress Address;
        email    Email;
        orders Order[];
        …
end
DataItem CustomerID int { readOnly=yes, … }
```

Service

- Services are remotable sets of behavior
  - Can be invoked locally or remotely and behavior will be the same
- More general concept than Web Service but also supports Web Service

```
Service StockQuoteService
      Function getQuote(symbol String) returns (money)
            …
      end
      …
end
```

Interface

- Interface definitions provide abstract definition of Service interface
- Can be created directly from WSDL for use by clients of any Web Service
- Service implementations can “implement” a given set of Interfaces

```
Interface IStockQuoteService
      Function getQuote(symbol String) returns (money);
              …
end
Service StockQuoteService implements IStockQuoteService
      Function getQuote(symbol String) returns (money)
              …
      end
      …
end
```

Annotations

- Annotations are used to:
  - Specify design time meta data to help drive tools
  - Specify declarative information the compiler will use to make a semantic mapping to a given runtime
- Annotations defined on types can be overridden on declarations that use the given type

```
DataItem SSN char(9){ readOnly=yes };
...
mySSN SSN { readOnly = no };
```

Stereotypes

- Stereotypes are used to associate meta data with a part definition
- Often used to drive transformations from EGL to target runtime
- Specialized EGL statements sensitive to stereotyped operands (**add**, **delete**, **get**, **replace**, **open**, **close**)

```
// A Record type definition mapped to a database table
Record Order type SQLRecord {
          tableNames = [“ORDER”],
          keyItems=[“orderID”] }

          orderID OrderID { columnName=“ORDERID” };
          dateOrderPlaced date { columname=“CREATDTE” };
          totalValue money? ;
end
// A variable defined of the above type
myOrder Order;

// A GET statement that reads from the database into the variable
// Meta data in OrderType used to transform the GET into necessary SQL
myOrder.orderid = “12345”
get myOrder;
```

Handler

- Handlers are used to define event handling functions within the context of some framework
- Stereotypes tell the compiler what that context is
- Current Examples
  - JSFHandler – Tags the handler as one that will be running within the Java Server Faces framework
  - JasperReportHandler – Tags the handler as one that will be running within the open source Jasper Report framework

Service Invocation

- Access to a service is always through a service reference variable
- Service implementation bound at runtime

```
Program MyProg
        quotes IStockQuoteService { @bindService };
        …
        Function main()
                  …
                  ibmquote money = quotes.getQuote("IBM");
                  …
        end
        …
end
```

Putting it all together

- Client Code

```
import com.acme.ordersys.api.*;
handler Client type JSFHandler {view = CustomerOrders.jsp}
   // Service references
   orderSvs IOrderService {@bindService{"OrderService"};
   utils ClientUtils { @bindService };

   // Fields bound to JSF components in the JSP
   custid CustomerId { inputRequired = yes, minInput=9 };
   orders Order[];
   // Function bound to some button on JSP
   Function getOrders()
       try
         orders = orderSvs.getOrdersForCustomer(custid);
             …
       onException( ex ServiceInvocationException )
             …
       end
   end
   // Other event handlers
end
```

- Service Code

```
import com.acme.ordersys.api.*;
Service OrderService implements IOrderService
   Function getOrdersForCustomers(
       custid CustomerID) returns (Order[])
             orders Order[];
             get orders using custid;
             return ( orders );
   end
End
```

- Common Interface Code

```
package com.acme.ordersys.api;
DataItem CustomerId char(9) { minInput=9 } end

Record Order type SQLRecord
   { tablenames=[["Order"]] }
   custid CustomerId;
   orderId OrderID;
   …
End
Interface IOrderService
   Function getOrdersForCustomer(
      custid CustomerId) returns( Order[] );
             …
End
```

EGL Build Descriptor

- Defines certain “transformation parameters” specific to transforming code for a given platform
  - Mapping of logical names to physical resources
- Also defines parameters that guide the transformation process itself
  - Output directories, Upload destinations, etc

•Client side descriptor

```
[Client.eglbld]
<BuildDescriptor
   name=“Client"
   system="WIN"
   J2EE="NO"
   genProperties="GLOBAL“
   deploymentDescriptor=“Client.egldd” >
</BuildDescriptor>
```

•Server side descriptor

```
[CICSServer.eglbld]
<BuildDescriptor
   name=“CICSServer"
   system=“CICS"
   dbms="DB2"
   sqlValidationConnectionURL="jdbc:db2://…/SAMPLE
   sqlJDBCDriverClass="com.ibm.db2.jcc.DB2Driver"
   sqlDB="jdbc:db2://localhost:50000/SAMPLE“ >
</BuildDescriptor>
```

EGL Deployment Descriptor

- Binding of service reference to a given implementation done external to EGL language definitions – EGL Deployment Descriptor File
- Used to generate code that is independent of EGL code implementations:
  - Service proxies for clients
  - Web Service layer for deploying web services

•Client side deployment descriptor

```
[Client.egldd]
<egldd>
   <bindings>
      <webbinding
         name=“OrderService”
         wsdl=“OrderService.wsdl” />
      <eglbinding
         name=“ClientUtils”
         service=“com.acme.client.Utils.>
         <protocol.local>
      </eglbinding>
   </bindings>
</egldd>
```

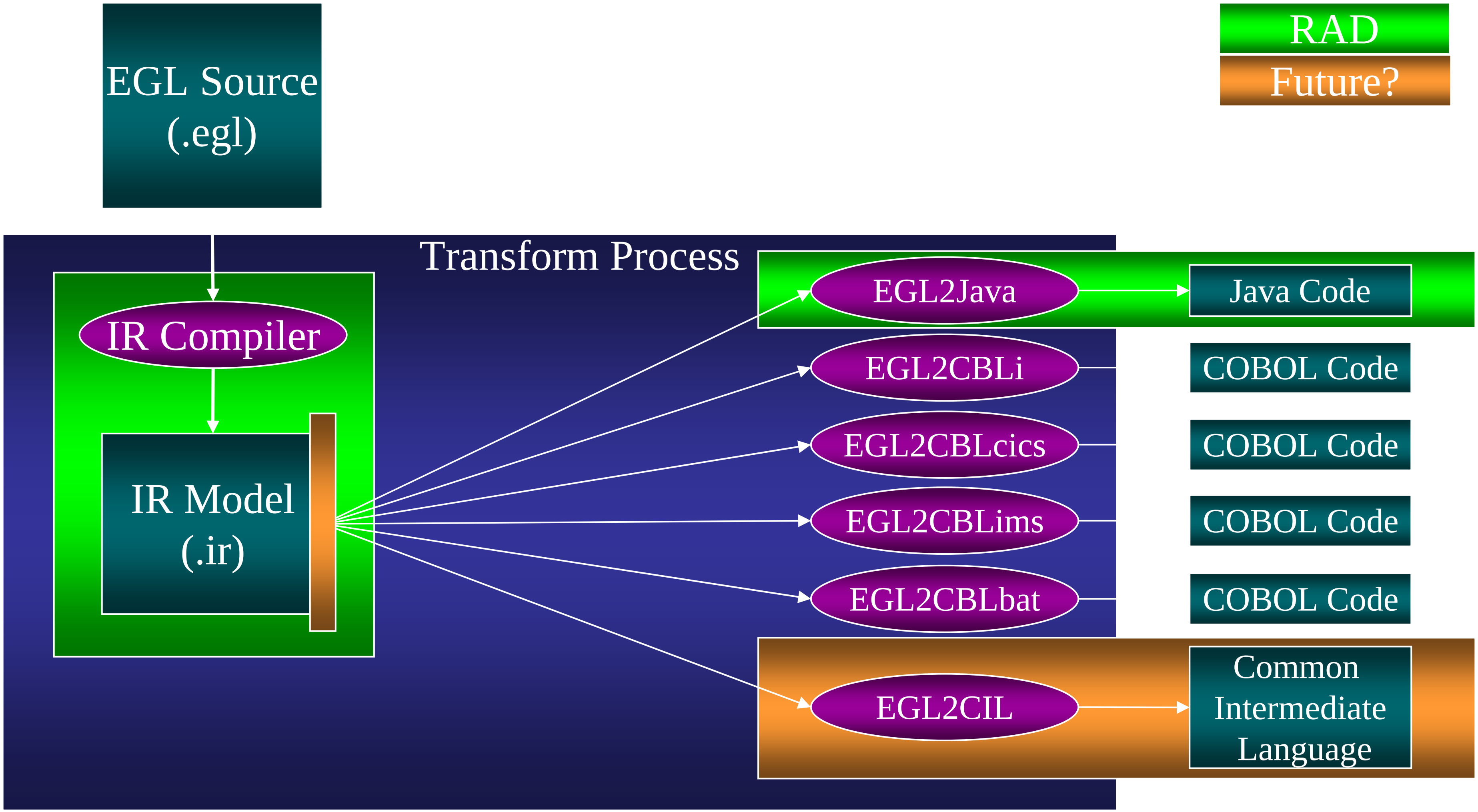
•Server side deployment descriptor

```
[Server.egldd]
<egldd>
   <webservices>
      <webService
         implementation=“com.acme.ordersys.OrderService” />
   </webservices>
   <bindings/>
</egldd>
```

Transformation to runtime

- EGL compiler creates persistent abstract model
- Transformation engine reads this model together with the build descriptor and performs code generation
- If special proxies and wrappers are necessary for deployment of the code into a given runtime then the deployment descriptor is also transformed
- Java or COBOL used as the target language depending on target platform
  - COBOL for CICS, IMS, and iSeries
  - Java used for all other platforms

EGL Transformation Process



Standard Extensions to EGL?

- What does extending EGL mean
  - New Annotations/Stereotypes
  - Extension points in the transformation engine to read and transform code based on new meta data
  - Extension points in the interpreter to implement the semantic of new meta data
- What does it not mean
  - Updates to parser and core language semantics
  - These kinds of extensions controlled by IBM today
  - Would go through standards body in future

Summary

- EGL is simple and modern
- It allows business flexibility
  - Platform neutrality allows late binding decisions on where to place application components
  - Skills in EGL can be applied to the end to end application
- It is a natural target of migration
  - Result is in a form that is close to the original
  - Result will run well in the original platform while allowing it to be moved to other platforms later
- EGL provides a bridge between the old and the new