Fusion Column

ODTUG Technical Journal

July 2010

The reusability test

"I need to write a query that returns all actual status details for the order items for a customer. I need it for the customer service application that we are developing. It is fairly complicated, as I need to join and outer join multiple tables together, calculate aggregates at several levels and apply some analytical functions. ' says one developer.

'Well, isn't this your lucky day!', replies another. 'We have had to deal with a similar challenge in the order tracking system we built last year. I worked on a query that returns almost the same results!'

It sounds like our first developer is fortunate indeed. But is that really the case? And how fortunate exactly? And why exactly is it that we consider her fortunate?

What we are discussing in this article is *reuse*. Putting an existing artifact to another use that was not foreseen when it was originally created. Or something to that effect. That seems a good thing: the developer who can reuse an existing artifact does not have to build something herself. So that is an efficiency gain, as it saves the team some time. The original query discussed above has been used in a production system for the last five months, and before that it has been extensively tested. So there we have another benefit: we do not need to test the artifact again, as its quality has been proved already.

However, even though that query has been written and was tried and tested in a production system, it is not necessarily suitable for reuse. First of all we have to ensure that it really provides the functionality that is required: does it return all the columns we need, does it support the filter criteria that we need.

Then we have to also assess how the query is exposed: is there a database view or a PL/SQL package that we can invoke to retrieve the query results from? Any improvements made to the query - like new functionality or more appropriate query hints - are immediately inherited by our application. However, if there is no such run time object available to us, the reuse we get may consist of copy-paste of the SQL statement into our application. In general, duplication of code is the lowliest form of reuse, that we should only draw on as a last resort.

This article discusses reuse. Why would we want reuse and what do we have to do to realize the potential benefits of reuse? What are important considerations around reuse that we should pay attention to? How does a structural approach to reuse impact our organization? Which types of reuse should we discern? And which mechanisms are available to us in general and in Oracle Fusion

Middleware (FMW) in particular? We will take a close look at the ADF Taskflow - one of the primary vehicles in FMW for achieving reuse.

Why? Benefits

The most glaringly obvious benefits of reuse are time savings, efficiency gains and shorter time to market. Not having to build and test functionality will usually save us some effort - even though some time will have to be spent on finding, assessing and integrating the reusable components. When development through the reuse of existing artifacts becomes a little more like assembly, the time required to produce new functionality is likely to become shorter as well. This in turn will make our development organization and as a result even the business more agile.

In addition, the quality of newly developed systems is likely to be higher when they can draw upon components that have been tested and tried extensively.

If - and that is a big if as we will discuss next - the reusables are reused in a proper way, for example through well defined interfaces, not breaking encapsulation and as loosely coupled as can be, then a change in the implementation of a reusable will not impact the system that reuses the artifact. That in turn then means that improvements need to be created and tested only once and the positive effects will be experienced by all re-users. If the interface through which the component is reused were to change, some ripple down effects are to be expected in the consumers, but typically even then the net effect of 'changing once, absorbing the change in all consumers' is positive.

A big bonus is waiting for us when we are really serious about reuse. Because when we are, we have to adapt both the architecture of our application and the way we design and develop the application. The question you may ponder: What do we have to now in the way we develop our application to be able to a give a constructive response when someone comes along and says: "' I like that part of your application, can I reuse that?'

Developing for reuse - with the explicit objective of producing reusable assets - as well as developing with reuse - assembling parts of the application using reusables rather than developing from scratch - requires a different approach in process and architecture. An approach that is much more structured, focused on smaller parts that can be designed, developed and tested in relative isolation before being published and consumed. An approach also that requires thinking ahead, and outside the bubble of the project and its immediate local objectives. An approach that can bring different teams together, resulting in energy and synergy and a much better use of resources (through reuse of sources).

Considerations

A serious strive for reuse has several consequences for an organization. We should take time to think those through and act accordingly.

In order to reuse assets, those assets must be available. That means that whenever new components are developed, we should assess whether these components have some reuse potential. When they do, they should be developed in such a way that makes them suitable for reuse. That means for example publishing meta-data about the asset that makes it discoverable by potential re-users and allows them to assess the component. It may also mean that a more generic component is developed, with more functionality and a more general interface than the project requires and had in mind. Usually this requires additional effort that has to come out of some budget. Especially in the initial stages of reuse in an organization, it may be useful to have a form of 'reuse sponsorship' where a central committee awards funds to compensate projects for the extra effort required to develop for reuse.

This same committee takes on responsibility for the 'governance' of reusable assets. Governance consists of identifying and helping to shape reusable assets, enforcing guidelines for proper reuse, gathering and exposing meta-data on these assets to make them discoverable, (co-)managing the life cycle of these assets (ensuring they are kept up to date, coordinating introduction of new versions and retirements of old ones).

In general, developers and projects of assets that are deemed reusable will suffer a loss of control. Their work becomes subject to governance and they have to take on a responsibility for (potential) consumers they may not even know about. It can be a challenge to persuade development teams to accept this relinquishing of power and freedom. Explicit recognition for their contribution to the common good - one of the drivers of the open source community that is all about resource - can be a factor.

Interface based loose coupling

Proper reuse requires several things including: loose coupling, encapsulation and well defined interfaces that specify the interaction. Even though we want to foster reuse, we still want to minimize dependencies between components. Components that are being reused should still be able to further improve their implementation and extend their functionality, without being [overly] constrained by their consumers. To that end, the interaction between consumer and asset should be through a well defined interface only; there should be no direct dependencies on the underlying implementation of the asset (in fact, that implementation should be hidden from view: encapsulated).

An excellent example of this approach is seen with PL/SQL packages: the interaction is based on the Package Specification and the Body is the encapsulated implementation of what the Specification makes available to consumers. There can be many procedures in the body that are not published through the specification. Those are completely encapsulated and can be changed at will without impact (unless they start raising exceptions that can bubble up through the publicly exposed procedures).

In general reuse is based on an interface and a contract. The former defines the interaction in a functional way using input and output parameters, faults and possibly events as an additional interaction mechanism and of course a description of what the reusable does. The latter coordinates the non-functional interaction; this includes performance and availability specifications, configuration options, authorization requirements, possibly the cost involved with reusing the component. A reuse

contract can be formalized through a Service Level Agreement (SLA), especially when reuse occurs across organization boundaries.

Dependency Injection

A valuable mechanism that helps increase the reusability of assets without sacrificing the encapsulation is through the mechanism of dependency injection. In short: components that we want to equip for reuse may have dependencies on other components. A reusable asset for example may be able to send emails, print documents, write debug information to a trace file and allow custom validation on the data it processes. Instead of building all these capabilities into the component itself or requiring the consumer of the asset to configure the mail server and print channels exactly in the way the original develop of the asset had in mind, dependency injection allows us to 'inject' a component that does email sending that the asset will call upon. Similarly, we can inject components to print, log and validate - or inject the configuration of the printer or the log file. The reusable asset can make use of the infrastructure and facilities in the environment of the consumer, even though the developers of that asset knew nothing about the consumer when they created it.

Levels of reuse

Reuse can happen in several ways and at various levels. It can be very loose, such as 'merely deriving inspiration', or very tight. And reuse can take place at design time or meta level - I would take to use your data *model* - or at run time - I want to use your data. The consequence of reuse vary with the type and level of reuse; reuse at runtime for example obviously has further reaching consequences than mere design time consumption.

design time

Reuse at design time revolves around sources, often packaged in libraries such as JARs, JavaScript files, ADF Libraries or even Forms PLL/PLX and OLB/OLX files. Other forms for design time reuse include ADF Declarative Components, Page Templates and Taskflows, CSS stylesheets, Utility PL/SQL Functions and XSD and XSLT documents. And of course Oracle Designer's Reusable Module Component and List of Values. Reusing artifacts at design time typically means that in order to benefit from improved and new functionality in the reusable, the application needs to be redeployed (and perhaps retested as well).

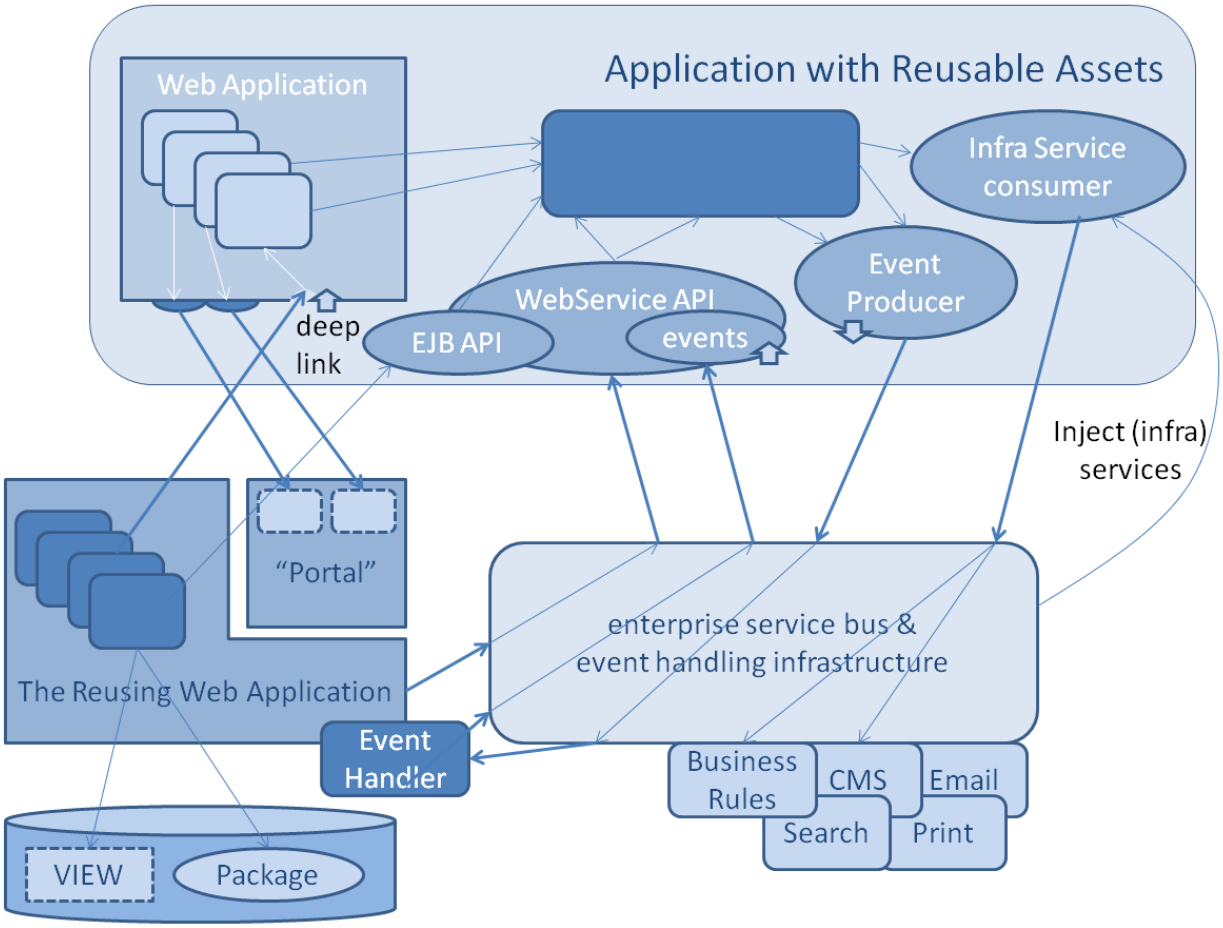
Assets reused at design time are usually absorbed into the consuming application or linked into it and deployed along with it. There is bound to be duplication of artifacts which can make it difficult to migrate to later versions of the reusable asset. It is important to keep the reusable as much isolated from the application as possible with a single reference to it if at all possible. It is important to beware of code duplication - copy and paste does not constitute reuse in a meaningful, lasting form!

run time

Assets reused at design time lack a certain spark - they do not have a life of their own and are dependent on their host to provide essential lifelines such as access to a database and a JVM to exist and run in. Runtime reuse typically means consumption of assets that run independently of the consumer, potentially serving other consumers at the same time. Changes in the asset are typically

immediately picked up by the consuming application. The run time dependency introduces a risk as well: the consumer may not be able to run properly when the asset is unavailable for some reason. Such a critical dependency may not be required and is certainly not desired. Well known examples of run time reuse include calls to packages or queries from views - possibly via synonyms or database links - , Enterprise Java Beans, Web Services (Servlet, REST, SOAP), Portlets and Embed (HTML) via iframe or JavaScript Widget, such as Google Maps.

Two special forms of run time reuse with minimal dependency are achieved through subscription to events and by using deep-link navigation from one application into the other. The latter describes a situation where we want to reuse one of the UIs in some web application and do so not by embedding or integrating that UI but rather by opening the application in a popup window or iframe with the appropriate context passed as URL parameters to the "reusable" UI.



An interesting in-between situation exists with a PL/SQL package that has been created with 'invoker rights'. This package contains PL/SQL code - but any references to external objects are evaluated among the objects available to the invoker of the package. To use the package, you need to bring your own objects to the party.

Reuse in Fusion Middleware

Reuse is an important theme through many of the products in Fusion Middleware. When you think about the scale of the development effort undertaken by Oracle in creating Fusion Applications, it is only to clear that the potential efficiency gains of reuse as well as the structured approach that comes with a focus on reuse are of extreme importance.

Service Oriented Architecture revolves around reuse, next to business agility, and services are the primary example of reusable assets. Services exist to be (re)used. In Fusion Middleware, the SOA Suite and the composite applications deployed on it, are the implementation of these services. The Oracle Service Bus complements the SOA Suite, primarily to extend the reach of services across the boundaries of departments or enterprises. Other FMW products that help organize and foster reuse of services include Oracle BPA - for thorough process and service oriented analysis and design, Oracle Enterprise Repository - supporting mature, enterprise level governance of services - and Oracle Service Registry for design time and run time discoverability of available, reusable services.

The (web) services discussed above are programmatic assets, without user interface. The support in FMW for reusable assets *with* a user interface is founded on ADF Taskflows. Taskflows are self contained, encapsulated units that contain activities, navigation rules, UI pages, Java classes and managed beans and associated web resources. Taskflows may contain their own Data Controls to access enterprise resources, including ADF BC components - although database connections and web service endpoints are typically passed into them from the consuming application.

Taskflows have an interface that accepts input parameters that configure the taskflow for a specific usage. A taskflow can return a single result value. Communication to and from taskflows is based on contextual events: asynchronous signals mediated by the ADF infrastructure that a taskflow can publish - to make results or findings available to the consuming application and indirectly possibly to other consumed taskflows - or consume - to accept additional instructions for example to refresh or synchronize the taskflow with the environment in which it is consumed.

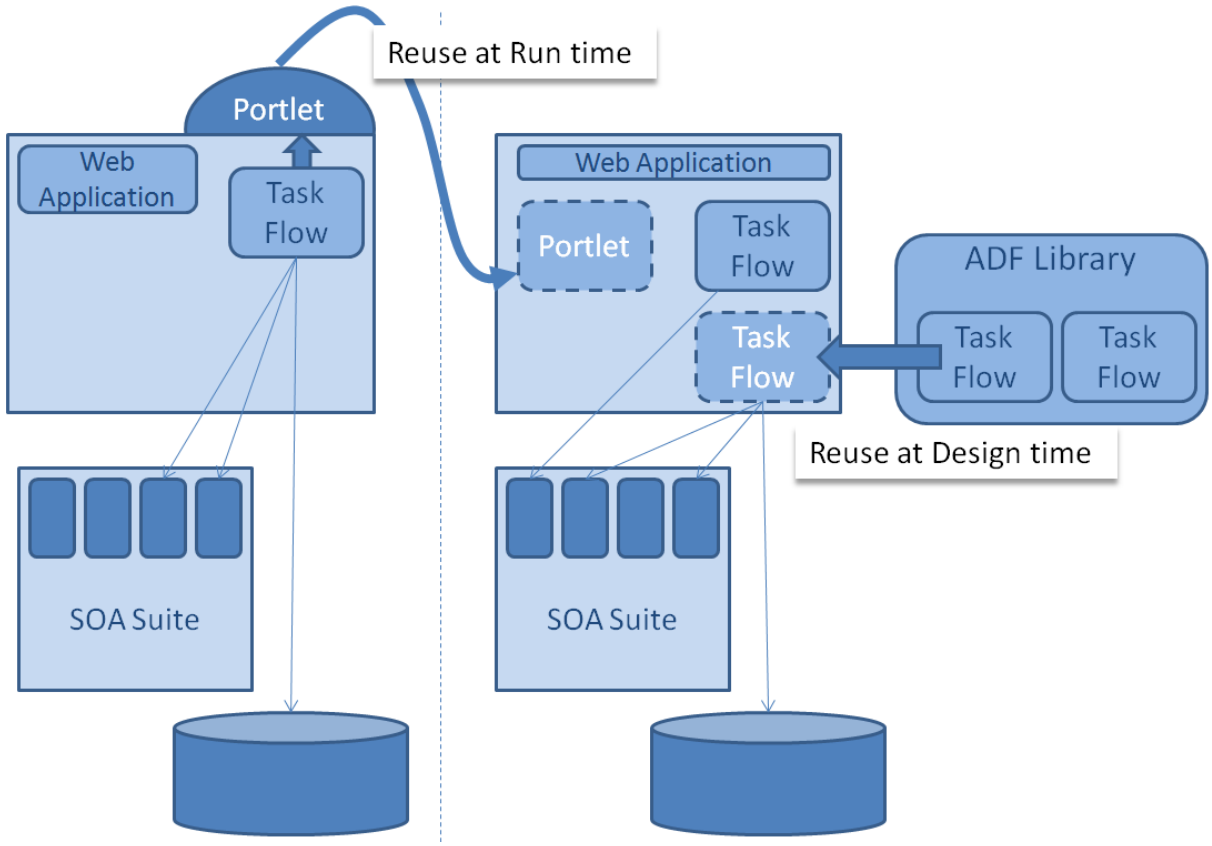
Taskflows can be developed and tested in stand-alone applications. They can be published and subsequently reused in several ways, both at design time as well as run time:

- Taskflows can be deployed to an ADF Library. This library can be imported into consuming applications that can reuse taskflows from the library by embedding them in the application's pages. When the consuming application is deployed, the ADF Library is deployed along with it. The Oracle WebCenter Services are an example of reusable assets that are distributed as taskflows in ADF Libraries.
- Taskflows can be exposed as Portlets. Portlets are run time Web Services with a User Interfaces, exposed by a portlet container (such as an ADF application using WebCenter Framework). ADF applications can also utilize WebCenter Framework to consume Portlets (either Taskflow based portlets or others). Portlets are self contained run time units who take care of their own run

time enterprise resources such as database access. These portlets accept the input parameters defined in the interface of the task. They also produce and consume events.

- Taskflows that have been deployed as part of an ADF Web Application can also be called remotely by other ADF application. A consuming application can use the Call Taskflow activity to embed or deep link into a remote taskflow. The context set through a taskflow's input parameters can be passed along in this remote call and the result is returned to the consuming application. The contextual events are not supported in this scenario. Note that the remote taskflow call does not require use (and licensing) of Oracle WebCenter Framework.
- Finally, taskflows can be registered in the WebCenter Composer catalog. These taskflows can then be consumed at runtime by adding them from the catalog to an ADF web page that contains the pageCustomizable component that enables such run time editing.

The next figure depicts the difference between design time taskflow reuse and runtime portlet based reuse of taskflows. It also shows how a taskflow reused at design time does not have its own runtime resources but works with the database connection and the local SOA Suite instance accessed by the consuming application, whereas the taskflow reused as portlet has its very own, remote environment of enterprise resources that it draws upon.



Customization

ADF provides support for a mechanism that at once increases the reusability of assets as well as breaks the encapsulation of those same assets. Through (seeded, design time) customization, it is possible to not only inspect the inner workings of reusable taskflows, but create modifications on top of those taskflows. The net effect is that at run time the reusable taskflow may look and act considerably different from the original reusable asset. These modifications are recorded in separate files, applied at run time on top of the imported definition of the taskflow.

Conclusion

Reuse potentially benefits your organization in various ways, including cost savings and efficiency gains resulting in higher quality and shorter time to market. Less obvious though perhaps even more important is the potential improvement in application architecture and development process when you become serious about reuse.

To illustrate the importance of the two main reuse mechanisms discussed for Fusion Middleware: Fusion Applications is built from taskflows. Every task or functional workflow is implemented through an ADF taskflow. And there are over 10,000 of them. Additionally, for every taskflow - that provides the user interface for a certain user task - there is a web service supporting a programmatic interface for that same task. Both taskflow and web service are reusable assets. And just as important: by architecting the applications for reuse, designing and developing using stand-alone mechanisms such as taskflow and web service, the development process is well structured and far easier to manage. Instead of creating a small number monolithic modules, the teams assemble those modules from individually designed, developed and tested reusable assets.

Future installments of this column will revisit reuse and that crucial reuse vehicle in ADF: the taskflow.