



Automating application mapping and replacing Application Descriptor Files with relationships

Document version 0.1

November 5, 2013

Morten Moeller (moellerm@us.ibm.com)

Preface

This paper provides guidance on how you can automate the definition of business application and collections using rule-based definitions. As an added benefit, this paper demonstrates how you can replace existing application descriptor file based solutions for application discovery with the easier-to-manage rule-based business application definitions, and thereby avoid the administrative nightmare associated with maintaining application descriptor files.

For many years, IBM® Tivoli® Application Dependency Discovery Manager (TADDM) has supported the use of application descriptor files to associate certain discovered components and servers with specific business applications. This functions has never become particularly popular because it require that you maintain and deploy application-component specific files to each system in your infrastructure that host resources you want to associate with an application. The application descriptor files must be maintained every time the lifecycle state of a component changes, and require that you discover the components in order for TADDM to recognize their existence. In a dynamic, virtualized environment it is almost impossible to keep up with the speed of the automated change processes. The manual management of the application descriptor files is cumbersome, time consuming, and error-prone, and represent such a huge manual effort, that most organizations decline to use it to automatically map business applications.

In TADDDM 7.2.1 the Grouping Composer function provides an alternate way to associate specific resources with a business application or a collection. With the Grouping Composer, you can define rules that specify the particular components and servers that support certain business applications or use cases. The rules identify the resources based on queries against the TADDM database, so it is centrally managed, does not require deployment of files throughout the infrastructure, support bulkloaded and placeholder components, does not require discovery, and supports maintenance of collections in addition to business applications. As you can see, rule-based definition of business applications provide superior functions compared to the use of application descriptor files.

Be advised, that to benefit from reading this paper, you should be familiar with the Common Data Model, and in particular how it support object inheritance and relationships.

It is the hope of the author that you will find the information in this paper useful - even though you do not have access to the example environment, and therefore cannot reproduce the exact results as they are described in the this paper. If you have any comments or questions, please do not hesitate to contact the author.

Table of Contents

1	Application mapping and application descriptor files	1
2	Replacing application descriptor files with relationships	4
3	A deployed application example	6
4	Designing application rules	13
4.1	<i>Marker rules.....</i>	13
4.2	<i>Functional groups</i>	14
4.3	<i>Dependency rules.....</i>	15
4.4	<i>TADDM agents</i>	16
5	Defining rules for the Trade application.....	17
5.1	<i>Marker module rules</i>	17
5.2	<i>Dependency rules.....</i>	28
5.3	<i>Service dependencies</i>	35
6	Creating application patterns	37
7	Automating application definition	42
8	Summary	45
Appendix A.	Installation	46

1 Application mapping and application descriptor files

For many years, Tivoli Application Dependency Discovery Manager (TADDM) has supported the use of application descriptor files (ADF). These are tiny xml files that contain information on how various application components are associated with business applications. Application descriptor files reside on each of the systems in the IT infrastructure that host components that support business applications. When the systems subsequently are discovered, TADDM will recognize the ADFs and automatically make the discovered components members of the business application that is referenced in each application descriptor file.

While application descriptor files provide a convenient way to automatically maintain the application topologies, its acceptance has been limited because of the administrative efforts required to take advantage of this feature. To use application descriptor files effectively, it is necessary to:

- Ensure that the correct files are deployed to each target that is part of a business application or a business system.
- Ensure that the files are located in the correct subdirectories in order for TADDM be able to discover and interpret the information
- Create special custom server templates to discover ADFs associated with application server components for which a standard sensor does not exist.
- Manually remove components from the application definition when they no longer support an application.

This approach is labor intensive, cumbersome, error prone, and requires additional change management tasks to maintain the ADFs when a system changes role, lifecycle status, location, business unit association or is sunset.

The use of application descriptor files has to a large extent been overtaken by the dynamic application definition capabilities in the Grouping Composer. These capabilities allow the TADDM administrator to define rules that govern how members of an application are automatically added – or removed. The rules use MQL queries to identify discovered resources with specific characteristics (attribute values or relationships) and associate these resources with a particular application.

Rule-based application definition provides several benefits over the use of application definition files. The most important benefits of using rule-based application definition are:

- Do not require deployment and maintenance of files residing on the target systems.
- Is centrally defined and maintained.
- Can be configured to automatically remove resources that no longer should be associated with the application.
- Does not require discovery, so it supports resources that have been added to the TADDM database through other means than discovery (bulkload, proactive REST calls, web services, and so on.)
- Can be defined in a generic way so the same rules apply to different applications.

It should be obvious, that the administrative efforts required to maintaining rule-based application definitions are significantly less than those required to maintain application definition files, while at the same time being less error prone and more powerful.

1.1 Providing business context

You will often find similarly named resources in both the development, test, Q/A, staging, and production environments and one of your challenges is to be able to differentiate the instances because they must be treated differently depending on the context in which they exist. In other words, you need to apply business rules that can provide the context for the resource so you can apply different processes or policies for monitoring, configuration and change based on the context.

When using application descriptor files, each individual ADF defines one or more components to be associated with a specific functional group within specific application instance. File names, functional group names, and application instance names are all hardcoded. It should not come as a surprise, that if a resource is promoted from test to staging, you must update all the application definition files as part of the process in order to reflect this change of context. Besides a specific application instance name, application descriptor files do not provide the means to apply the business context in which the resource exists.

If you use rule-based application association things are very different. The resources that are associated with an application can be identified by resource type, name, and a number of additional attributes that provide the context in which the resource participate. The context can be deduced from meta data associated with each individual resource and maintained through custom server or custom template extensions. Often, general attributes such as location, CIRole, CIClassification, generalCIClassification, lifecycleStatus, and primaryOwner are used to provide the context, but associated adminInfo, extended attributes, userData, or userDefinedValues can also be used for this purpose.

It should be obvious that the rule-based application association provides far greater flexibility for providing business context compared to using application descriptor files. Naturally some work is needed to maintain template extensions to provide the business context, but this is far less than managing and deploying files to the individual systems in the infrastructure. The new CustomTemplateSensor in TADDM v7.2.2 provides a great leap forward in your ability to apply custom extensions to any sensor so you can apply business rules to provide the context for any resource that is discovered by TADDM. Prior to TADDM version 7.2.2, you could only apply extensions through the use of Computer System Templates and Custom Server Templates, so the only option to provide business context for resources discovered by built-in sensors was through custom extensions applied at the Computer System Template level.

1.2 Application descriptor files or marker modules

Associating specific infrastructure resources with specific business applications require insight into both the business and the technical application architecture. Typically, the TADDM administrator does not have that insight, and must rely on others to provide the details needed to create and deploy application descriptor files or create rules to associate the correct resources with an application. Service Delivery Managers, developers and the operational team are all excellent sources that can help identify business applications that must be defined, as well as the low-level resources that are critical to the successful operation of the applications.

Using application descriptor files to associate resources in the environment with specific applications, you create and deploy multiple ADFs to each target system. The specific resources that must be associated with each application vary depending on your requirements, but typically resource types such as J2EEDeployedObjects (includes J2EEApplications and J2EEModules), J2EEResources, WebVirtualHosts, and Databases are most relevant to associate with a particular application. You can say that these resources represent markers - or marker modules - that are unique to each application.

2 Replacing application descriptor files with relationships

If you look closely at the information in an application descriptor file you will find, that all it contains is a reference between an application instance and a server or a module component. For all practical purposes, this information represents nothing but an explicit relationship, so if you can find a way to register the same relationship in the TADDM database, without having to deploy and maintain files throughout the server infrastructure, you can achieve the same automated assignment without the administrative pain.

Furthermore, the application descriptor files only support two types of resources - servers or modules. This means that you cannot associate for example a database or a J2EEDataSource to an application because these types of resources are considered neither servers nor modules. As a matter of fact, only resource types that are subclasses of the AppServer or SoftwareModule classes in the CDM are supported by application descriptor files. If, on the other hand, you use relationships, you can associate any type of resource with an application. Using relationships you can associate important resources such as databases, WebVirtualHosts, and JDBCConnections with an application - in addition to servers and modules.

For top-level resources such as AppServers, IT Services, and Clusters it is pretty straightforward to define the rules to associate these resources with an application. For that reason associating these top-level resources with an application through application descriptor files has, in most instances, been replaced by rules. However, for low-level components like databases, J2EEServlets, queues, and virtual hosts things gets a bit more complicated.

Most of the low-level resources can be uniquely identified through MQL queries, but it requires intimate knowledge of the implicit relationships defined in the Common Data Model, and in some cases limitations in the MQL language prevent building rules that can identify the resources you want to associate with a particular application. This is the reason why most of the application descriptor files that are deployed today are component descriptor files.

You might consider associating the low-level components with an application statically, using the Grouping Composer, but as for all GUI interaction in TADDM, only top-level resources can be statically assigned to an application through the GUI.

To overcome these obstacles, you can define an explicit relationship between the application (or one of its top-level members) and the low-level component you want to associate with the application. Then, this relationship can be referenced in a rule so that the target of the relationship is added as a member of the application. This way, the relationship you define replaces the application definition file, and you do not have to worry about deploying and maintaining application descriptor files.

Sounds easy? Sure, but hold your horses.

As already mentioned, the Data Management Portal only allow you to manipulate top-level resources, and now you want to create relationships that involve low-level components. This means that you need a programmatic way to inject relationship information into the TADDM database. The command-line API cannot be used, so you need to find some other way.

To help you overcome this hurdle, a jython script named `relationshipManager.py` is provided with this paper. You can use this script to list, create, and delete relationships of any type between any resources known in the TADDM environment. The script identifies both the sources and targets of the relationships using MQL queries, which allows you to use any attribute or existing relationship as the basis for qualifying the resources that are the source or target of the new relationship.

Using the script to create explicit relationships unfortunately does not solve ALL your challenges. In some rare cases, for example when you want to use multiple source resource attributes to qualify a target, the MQL language is inadequate. However, TADDM v7.2.2 provides a similar facility to dynamically add and remove relationships, but this is based on SQL, so you have all the capabilities of the SQL language at your fingertips. Agreed, coding SQL statements require even better understanding of the Common Data Model, and the implementation of the TADDM database, but the dependency management feature in TADDM 7.2.2 can be used to create even the most intricate non-discoverable relationships such as those between a J2EEDataSource and a database. For more details regarding the TADDM 7.2.2 dependencies function, please refer to http://pic.dhe.ibm.com/infocenter/tivihelp/v46r1/index.jsp?topic=%2Fcom.ibm.taddm.doc_7.2.2%2FUserGuide%2Ft_dmp_manual_create_cidependencies.html?

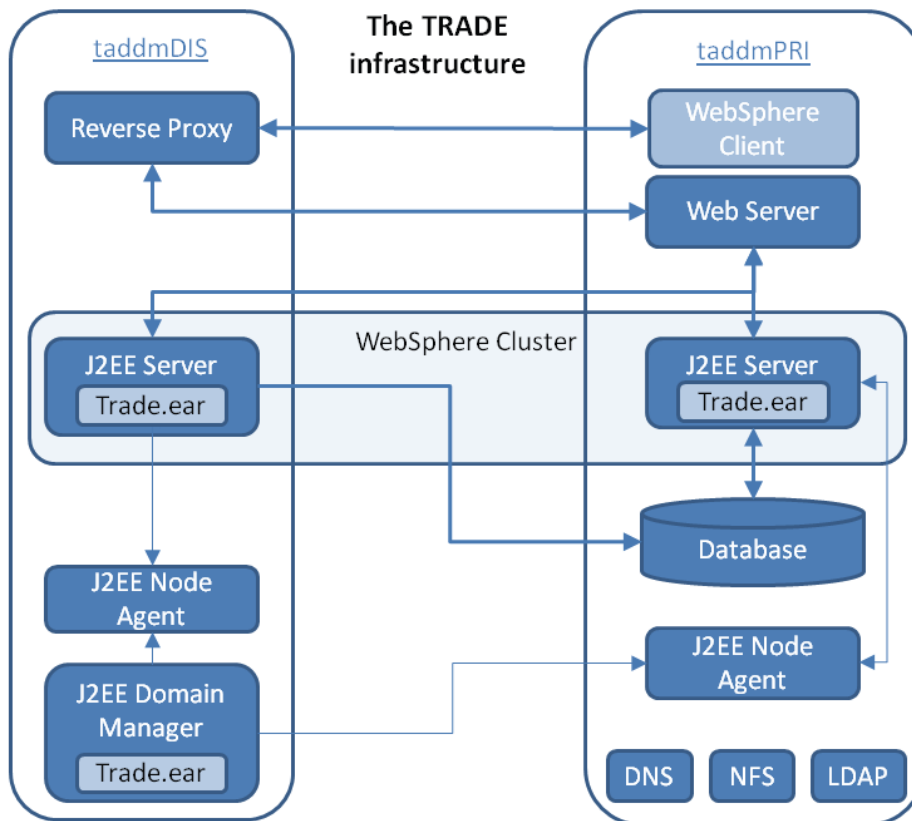
3 A deployed application example

The following introduces an example application that is used in the reminder of this paper to demonstrate how you can use MQL rules to define your business application components, and avoid having to deploy application descriptor files throughout your IT infrastructure.

In the following, it is assumed, that all the J2EE servers have been discovered at level 3 so that all J2EE applications, modules, connections, and resources are registered in the TADDM database.

3.1 Trade application overview

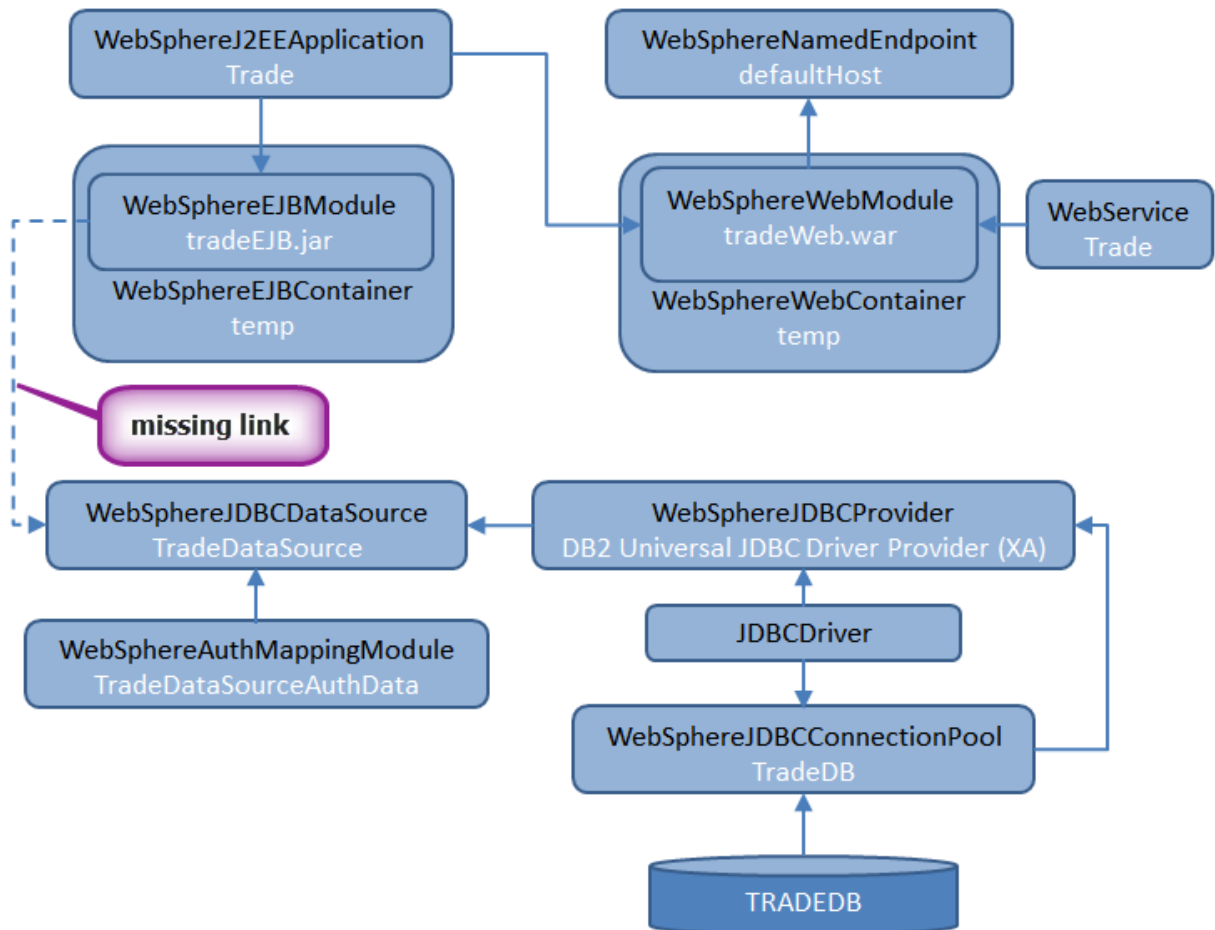
The target application for this example is the sample WebSphere application Trade deployed in a two-node cluster configuration. Both nodes access the TRADEDB database hosted on a single DB2Instance, and the cluster is front-ended by a Web Server. Users access the application through a reverse proxy server. The application architecture can be depicted like this:



In addition, the application complex uses both DNS, and an LDAP services, and the files for the Web virtual hosts are stored on a shared NFS file system for immediate updates. The demonstration environment also includes a client component which is used to generate transactions against the Trade application so that the appropriate monitoring tools can be used to verify availability and performance of the application.

All the application components are hosted on two virtual systems in order to limit the size of the demonstration environment. In a production environment, the total number of systems would probably be 10 – one dedicated system for each major service and application server component.

The overview of the logical architecture of the J2EE application, as it is defined in the Common Data Model, is depicted below:

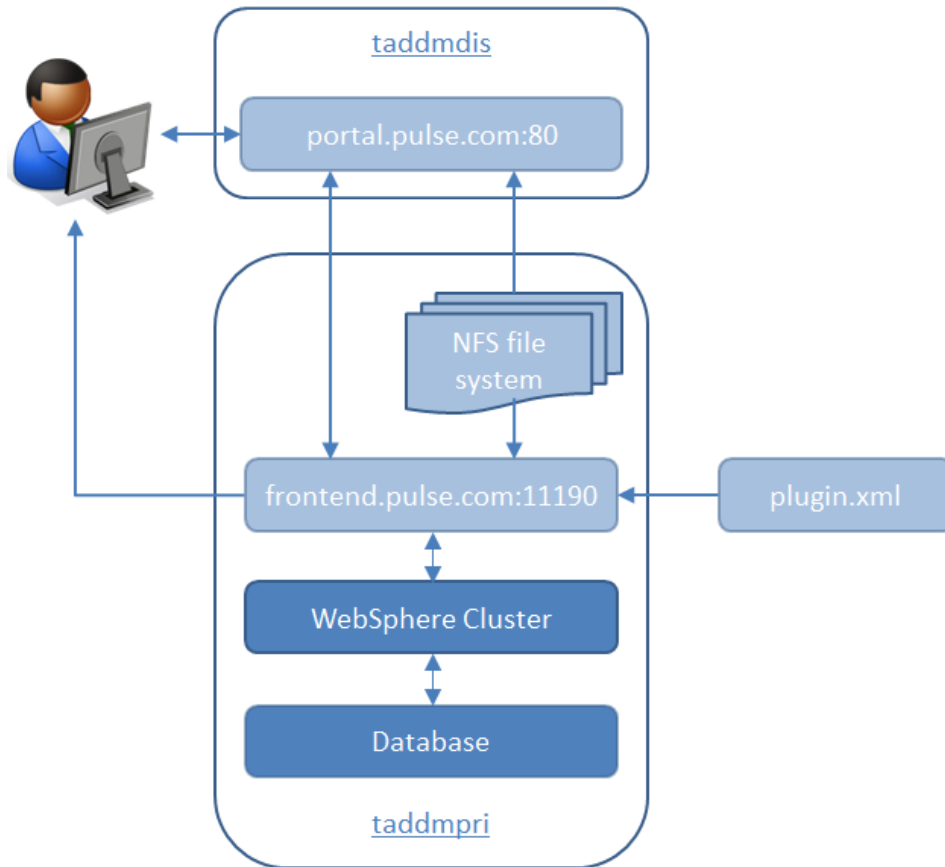


As you can see, most of the WebSphere components contain pointers to one another, so when they are discovered by TADDM, the data stored in the TADDM database contains implicit relationships between them. However, a fundamental principle in J2EE application design is the separation between presentation (servlets), logic (EJBs), and data (data sources). Because this principle has been applied to architecture of the Trade application, there are no discoverable references between the EJBs and the data source. Inside the EJB, data sources are referenced by a logical name, so from an application mapping perspective, there is a missing link between the EJB and the data source. As a matter of fact, the same is true for the relationships between servlets and EJBs, but because the J2EE application definition contains references to both types of resources, these references can be used to identify both which EJBs and servlets are required for the successful execution of the application.

The high-level knowledge about the Trade application provides enough information to identify marker modules that uniquely identify the application. The *tradeEJB.jar* and *tradeWeb.war* modules are obvious marker modules – both should be included to handle implementations in which the containers are split between multiple servers. In addition the *TradeDataSource* *JDBCDataSource* could be an obvious candidate because it can provide information about the databases used by the application. And it goes without saying that the *Trade J2EEApplication* can be used as a marker module too.

To identify the DBMS systems that host the database used by the application, you need a combination of the details specified in the data source (host, port database name), and a discovered transactional relationship between the J2EE Server and the DBMS hosting the *TRADEDB* database. By using the discovered transactional relationship you ensure that only DBMS systems that actively participate the provisioning of the application are included. However, because of limitations in the MQL language, you cannot create a rule that extracts information from the data source and use that information to identify the databases and DBMS systems they reference. For that reason, the *TRADEDB* database itself should be used as another marker module for the Trade application so that the DBMS can be identified.

To identify the two web servers you need to understand that they fulfill different roles in the application, and therefore must be treated differently. The web server on the *taddmpri* system implements a virtual host named *frontend.pulse.com*, which is the main interface to the Trade application. This is a normal web server that uses the WebSphere HTTP Server plugin to forward all requests received on port 11190 to the WebSphere cluster.

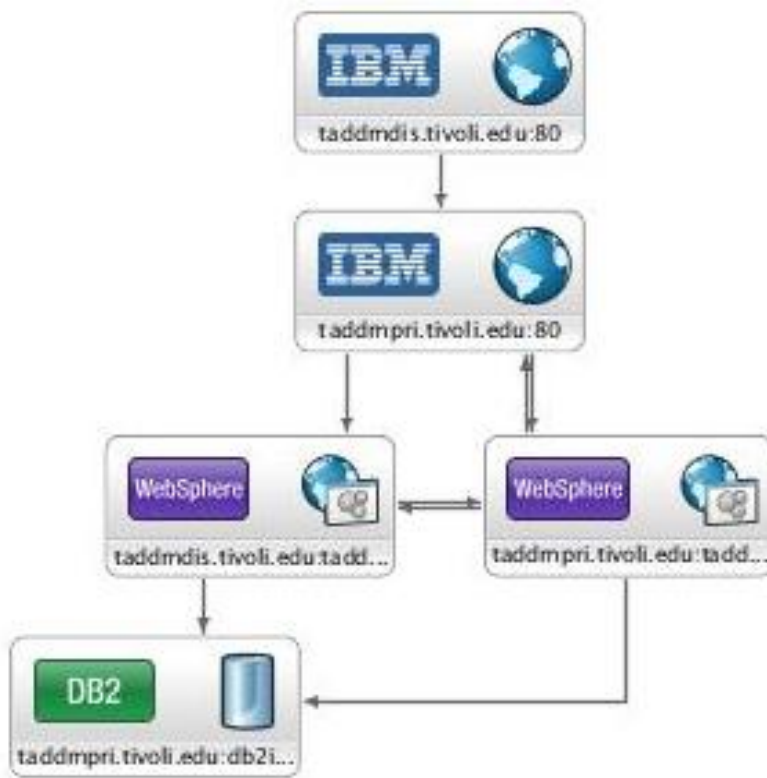


The web server on the *taddmdis* system is a proxy web server that provides access to the infrastructure from external sites. The virtual host name is *portal.pulse.com* and it listens to the default port 80. All accepted traffic is forwarded to the *frontend.pulse.com* host. Both web servers use a shared NFS file system that hosts the content of their sites.

Given this design, the two virtual hosts can be used as marker modules to identify the web servers that support all applications hosted in the WebSphere cluster. The web servers have no components or configuration settings that are specific to the trade application (except for a single entry in the *plugin.xml* file). This implies that the web server marker rules you specify for the Trade application can be applied to all TADDM business applications – if you care to create special rules at all. Since the markers are generic, you can just as well use the discovered relationships to identify the web servers that support the applications.

3.2 Discovered transactional dependencies

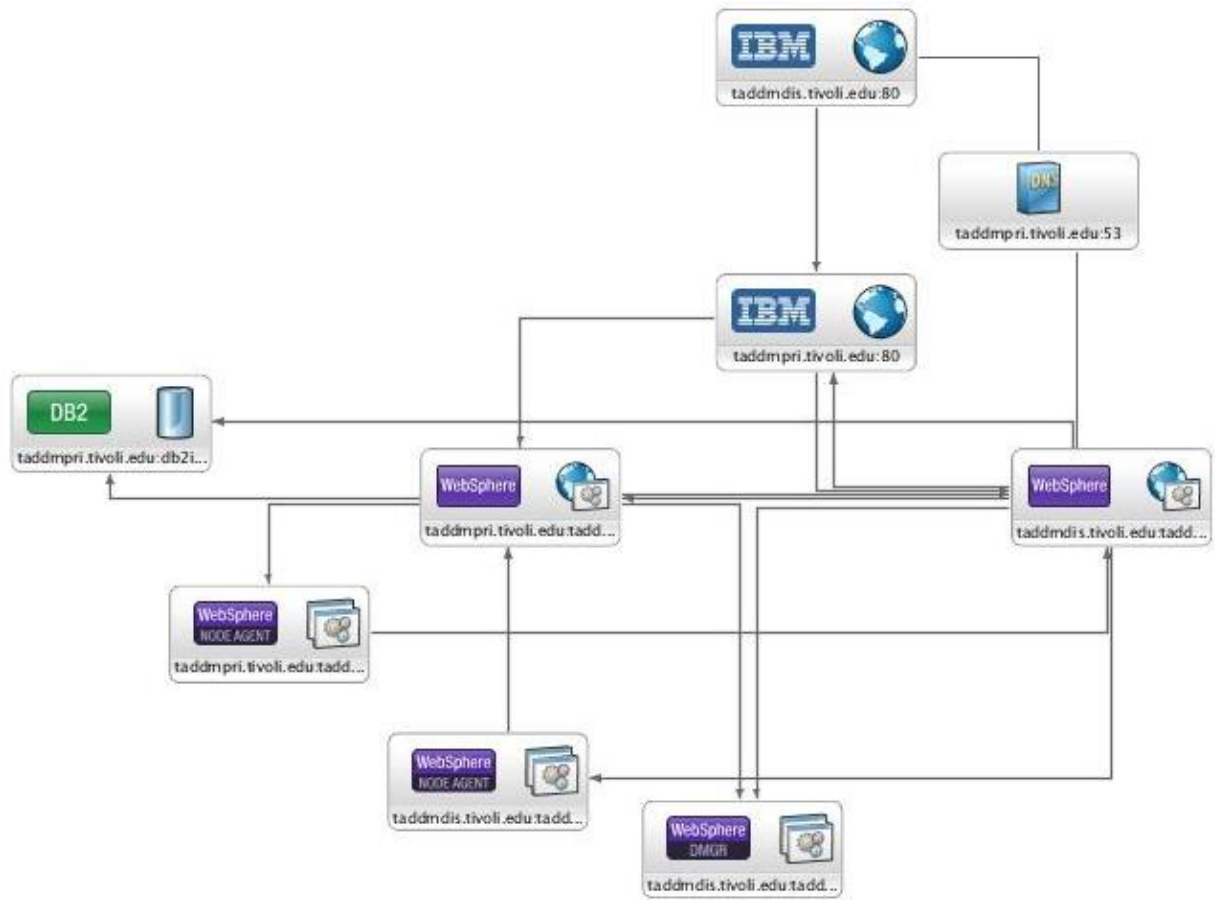
If you have discovered the infrastructure while the application was in use, TADDM will have recorded transactional relationships as depicted in the following diagram:



These relationships are critical for the application mapping because they represent active sessions that have been discovered. This means, that TADDM has observed active communications between the involved application servers, and has stored this information in the database as transactional relationships.

In the diagram above, you see, among other, that the two WebSphere servers both talk to the Db2Instance, and that both of them receive requests from the frontend Web Server. The proxy web server only forwards requests to the frontend web server. In this example, an interesting fact is that all results are parsed back to the caller through a single WebSphere server, but that simply reflect how the application has been configured.

The transactional diagram above shows only the discovered communications between resources that directly take part in the delivery of the Trade application service. In addition to the components shown above, the infrastructure contains supporting resources, such as WebSphere Deployment Manager, WebSphere Node Agents, and IT Services such as LDAP, NFS, and DNS. If these resources are included, the technical relationship map (including only TransactionalRelationships and ServiceRelationships) looks like this:

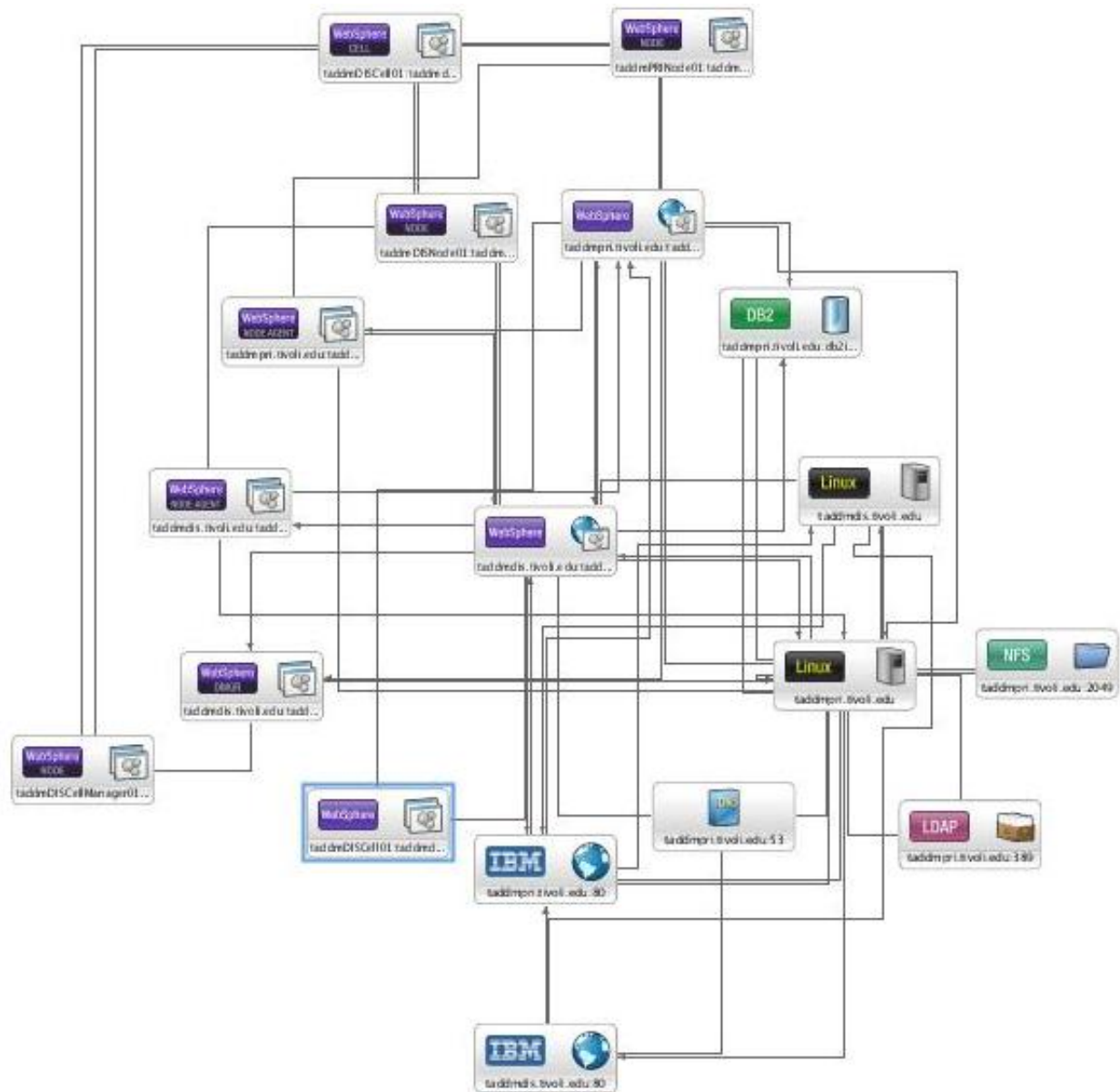


In the diagram you see that besides communicating with themselves, the two application servers are in contact with the WebSphere Node Agents and the WebSphere Deployment Manager. These extra resources are not directly involved in the delivery of the application, but are used internally in the WebSphere environment to manage the configuration and performance. Whether you want to include these management resources in the application definition or not is a matter of your internal best practices.

It goes without saying, that in a setup like this you might define multiple business applications that use the same WebSphere infrastructure, but all of these are managed by a single management environment.

Above, you also see that both the proxy HTTP server (at the top) and one of the J2EE Servers (to the far right) use the services of a DNS. If the computer system information had been included, you would also have seen that the operating systems on each WebServer system mounts a NFS file systems so that the web site files are immediately updated.

If you reveal all the dependency information TADDM has discovered, you might get confused. This is how it looks in this small demonstrations environment:



Besides the resources used to deliver and manage the Trade business application, you see operating systems, additional services (LDAP and NFS) and you also see additional logical WebSphere resources such as WebSphere Nodes and a WebSphere Cluster.

4 Designing application rules

To create the MQL rules that are used in the application definition to determine which resources to include in the application, you first use the marker modules to identify the key AppServers that host the marker modules. Each rule populates a specific functional group, which in turn is related to the application.

4.1 Marker rules

Marker rules are used to add marker modules to the application. As a matter of fact, low-level modules such as databases and J2EE modules can be added to the application, but they will not appear in the application topologies or inventory shown in the Data Management Portal. In the database, the low-level members will be defined as members of specific functional groups, and this information can be leveraged by the TADDM topology builder to visualize their parent (and AppServer) in the TADDM topology views. The same information can naturally be leveraged in related tools, for example IBM SmartCloud Control Desk, and Jazz for Service Management, that consume information from TADDM.

So, in many situations, primarily determined by the level of detail you use for change and configuration management, you may find it more appropriate to associate the application server that hosts the marker module with a functional group instead of the marker module itself. This approach may result in fewer functional groups, and thereby easier application management in the long run. However, the drawback is that you cannot see which specific components are associated with the application in the Data Management Portal application Detail View.

Using the Trade application as an example, you see that each of the marker modules can be used to identify the server that hosts it - and since all applications servers are sub-classes of the AppServer resource type, you should theoretically be able to build a single rule that identifies all the application servers that host marker modules. However, the MQL query language does not allow union operations, so you will not be able to use this approach. Instead, you must create individual rules, and functional groups for each marker module type.

In addition to using the marker module to identify the hosting resources, you will in many situations use additional attributes related to hosting resources in order to be able to differentiate between various implementations of your marker modules. This allows you to create application definitions that are specific to a special environment, for example a location, a domain, a customer, or a site. Basically you can use any attribute that has been populated for the marker module, or any resource related to the marker module, for example the hosting resource. You can also use discovered relationships between the marker module - or one of its related resources - to identify the specific instances you wish to include in the application.

4.2 Functional groups

When associating resources with an application, all resources become members of a functional group. Functional groups are used to group resources with similar functionality – but for most practical purposes you would want to have as few functional groups as possible.

Be advised that functional groups are not displayed in the application topologies.

Each rule that is used to associate members with applications populates a specific functional group, and these groups cannot be shared between multiple rules.

When planning your functional group layout, you should strive to create rules that represent super-classes rather than type specific classes. This approach will help you to standardize the use of functional groups, and ultimately allow you to create new applications simply by cloning application definitions. Imagine an application that is identified by specific a WebSphereEJBModule. In the Common Data Model, a WebSphereEJBModule has the following super classes: EJBModule, J2EEModule, J2EEDeployedObject, and SoftwareModule. This means, that a WebSphereEJBModule can be referenced as any of those object types. If you define the rule used to identify the WebSphereEJBModule using references to the J2EEDeployedObject resource type, you can reuse the rule for EJBModules that are deployed to other J2EE server types such as JBoss, Oracle, and WebLogic. So by using the CDM derivation hierarchy to your advantage, you can standardize the rules and names of functional groups to make your definitions as flexible as possible.

In the Trade application example you want to include application resources of at least three different types: WebSphereServer, Db2Instance, and WebServer. However, there may be more, depending on whether or not you want to include the resources used to manage the application. For WebSphere, this implies that you would include WebSphereDeploymentManger, and WebSphereNodeAgent resources. When deciding which resource types to include, you need to remember that under the covers TADDM keeps track of the dependencies, so when the information from the TADDM database is used to augment events, or assess the impact of a change, the discovered relationships can still be leveraged.

4.3 Dependency rules

Besides marker rules, you will also need to create rules that identify other related resources such as clusters, and services. Whether or not to include these additional resources on which your application depends should ultimately be decided by the needs of the IT management process that use the information in the TADDM database. Remember, no matter if you make, for example, an LDAP service a member of an application, TADDM discovers which AppServers and ComputerSystems use that particular LDAP instance, so the information is not necessarily relevant to the specific business application. When the TADDM information is consulted, the LDAP dependencies can be taken into account even if they are not directly related to the application. On the other hand, if you encourage incident and problem managers to use the TADDM topology information to identify root-causes and view change history, you may want to include the services and other related resources in the application definition in order to provide the *complete picture*.

Contrary to IT Services, you should consider including logical components and groupings such as WebSphere nodes, and clusters to the application definition. For problem management purposes it is important information that is useful and provides contextual information to better understand the application architecture. However, the logical components used to manage the application does not provide any application specific functions, so you may decide not to include these components.

Normally you should refrain from adding computer system, hypervisor, network, or storage resources to a business application definition. TADDM keeps track of the dependencies between your application server resources and all the underlying hosting platforms, networks, and SAN/WANs so they are automatically considered, and included in the topologies you see in the Data Management Portal. As a matter of fact, one of the only reasons to include the supporting hosting platforms, networks, or storage resources in the application definition is to record the dependency between an application and the special components (application firewalls, application specific SAN volumes, etc.) on which it relies. If you want to use the Business Application Inventory as the basis for accounting or similar administrative processes you should also consider including the computer system, storage, and network components directly in the application definition. From a technical perspective, there is no reason to specifically add hardware-like resources to the application definition.

4.4 TADDM agents

When working with application topologies, you have to remember that all the information TADDM needs to build a topology is not created from discovery alone. TADDM uses a couple of background tasks, called agents, to analyze the discovered information, and augment the information in the database with additional details, for example explicit relationships, and dependencies. To ease the invocation of related agents, agents are grouped into agent groups that can be referenced to run all agents within the group.

In order for you to be able to see the application topology in the Data Management Portal, the *dependency* agent group must have been executed subsequently to your discoveries. This agent group analyzes the discovery results and creates the dependencies between various resources. The group contains agents for almost each top-level resource type defined in the Common Data Model.

In addition the *dependency* agent group, you must also ensure that the *BizAppsAgent* is executed when you modify the application definition. This agent is responsible for executing the queries in the application rules so that the functional groups are populated.

Agents are executed on a scheduled basis, and you can see the timestamp for the last run in the Data Management Portal. In certain events, for example after discovery or after edition of an application definition, TADDM will automatically execute the relevant agents. If you cannot wait until the next invocation of a specific agent or agent group, you can invoke them manually using the *runtopobuild* utility in the `$COLLATION_HOME/support/bin` directory.

5 Defining rules for the Trade application

In the following it will be demonstrated how you can define the MQL rules required to associate the components and servers used in the sample Trade application to a business application. Using MQL rules, you do not have to maintain application descriptor files, and you can associate discrete resources that you otherwise cannot associate with an application.

We will make a distinction between rules that reference the marker modules - hereafter called *marker rules* – and rules that identify resources which are related to the marker modules or resources hosting the marker modules. Rules that associate resources related to marker modules will be referred to as *dependency rules*.

5.1 Marker module rules

In the following four marker rules are described; one rule for each marker modules that were identified earlier for the Trade application. In addition, a number of dependency rules will be defined in order to include additional resources and services, such as WebSphere Clusters and NFS services, that are used by the Trade application.

The naming convention used for the functional groups is the plural form of the specific object type of the marker modules, or the object type returned by the MQL queries.

5.1.1 The J2EEApplications marker rule

The first rule to create uses the Trade J2EEApplication marker module to identify all the WebSphere Application Servers (super class: J2EEServer) that host a J2EEApplication named *Trade*. In this example, only servers are associated with the location named *TEST* are included.

Both the rule and the functional group it populates are named *J2EEApplications*.

The first part of the rule is used to identify the WebSphereServer components (as J2EEServers) that host a J2EEApplication by the name of *Trade*, and is hosted on a system with a Fqdn that ends with *tivoli.edu*.

```
Select * from J2EEApplication where displayName=='Trade' and  
J2EEApplication.parent.locationTag=='TEST'
```

In this query, all the J2EEApplications with a name of Trade are selected. This includes all J2EEApplications independent on the J2EE platform or operating system. Next, only applications for which the locationTag of the hosting server (represented by the parent attribute) equals the string *TEST*.

The test for the content of the locationTag attribute is used to only search for components that exists in the test environment. If you want to implement environment separation in your queries, you use any attribute that is relevant in your environment. Often, the ContextIp, LocationTag, Fqdn and CIRole attributes are used for this purpose.

Notice how the rule uses the type and name of the discovered component to identify the server that hosts it. This is exact same information you would use in a component application descriptor file - but using rules, you do not have to deploy, and manage, and discover all the ADFs throughout the IT infrastructure.

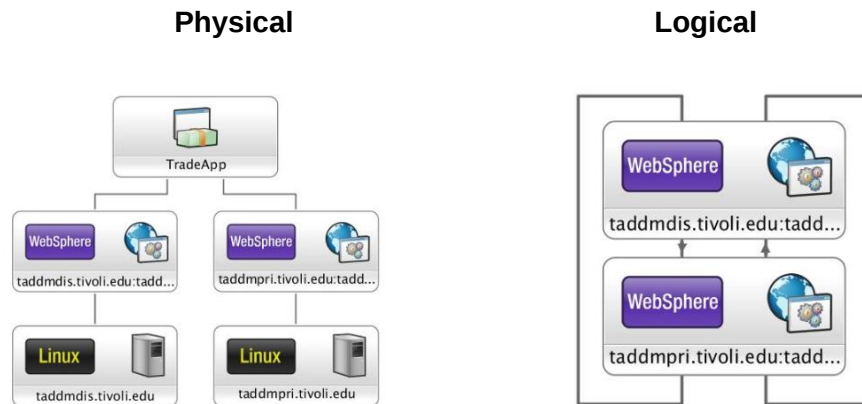
Hopefully you see how convenient it is to use the super class J2EEApplication to reference WebSphere2EEApplication resources. Using the super class you easily copy the rule, and reuse it for example in an application definition for a JBossJ2EEApplication named HomeBanking.

If you only want to use the marker modules to identify the J2EEServers that host the Trade J2EEAppliation, and associate the J2EEServers with the Trade TADDM business application, you can modify the rule to look like this:

```
Select * from AppServer where J2EEApplication.displayName=='Trade'
and J2EEApplication.parent.locationTag=='TEST' and
J2EEApplication.parent.guid==AppServer.guid
```

In this example the AppServer super class is used as a generic placeholder for any application server type. If you want include multiple application servers of different types in a single query, you can use the AppServer super class to treat all of them as the same type. By selecting from the AppServer class, and assigning the Guid of the application server to the AppServer.guid, you force MQL to treat your application server as an AppServer object.

If you view the topology after applying the first rule, it will look like this:



Hopefully it does not come as a surprise that the topology is limited to the two WebSphere Servers that host the Trade J2EEApplication. Notice how the physical topology include the two computer systems that host the WebSphere servers – even though only J2EE Server components are included in the application definition.

5.1.2 The EJBModes marker rule

The marker module rule that associates the *tradeEJB.jar* file with the Trade business application is very similar to the J2EEApplication marker module rule. Instead of looking for servers hosting a specific J2EEApplication, this rule identifies the servers that host a specific J2EEModule, or EJBModes.

Note: If you have already applied the J2EEApplication rule, there is no reason to also include the J2EEModule rule – the J2EEModules are included in the J2EEApplication, so the two rules will associate the same components with the application. However it is included here to demonstrate how you can include a specific EJBrule, in place of the J2EEApplication rule.

The query that identifies instances of the *tradeEJB.jar* hosted on systems with a locationTag of TEST looks like this:

```
Select * from EJBModes where EJBModes.displayName=='tradeEJB.jar'
and EJBModes.parent.guid is-not-null and
EJBModes.parent.locationTag=='TEST'
```

This example uses the super class `EJBModule` to find EJB modules that are named *tradeEJB.jar*, and the `locationTag` of the parent to include only those instances for which a parent is defined, and the `locationTag` of the hosting server is *TEST*. The test for the existence of a parent attribute ensures that EJBs deployed to a WebSphere Deployment Manger are not included.

When you analyze the application topology after having added this second rule, you will not notice any differences. In the test environment both the `J2EEApplication` instances and the EJBs reside on both WebSphere servers, and because the TADDM topology viewer only shows top level resources, you see only WebSphere Server instances that are related to any of the identified components.

To transform the query so that it returns the `AppServer` resources representing the `J2EEServers` hosting the `EJBModules`, you only need to change the from clause and add a reference between the `EJBModule.parent.guid` and the `guid` of the `AppServer`:

```
Select * from AppServer where J2EEModule.displayName=='tradeEJB.jar'
and EJBModule.parent.guid is-not-null and
EJBModule.parent.locationTag=='TEST' and
J2EEModule.parent.guid==AppServer.guid
```

No matter if you use the component query or the server query, an appropriate descriptive name for the functional group that is populated by the query is: `EJBModules`.

5.1.3 The WebModules marker rule

This rule is similar to the two previous rules. However, The Common Data Model has two resource types with the name `WebModule`, so in this case you have to use the long version of the class name in the select and where clauses in the query. The long name for the `WebModules` class that is used to identify J2EE `WebModules` is:

```
com.collation.platform.model.topology.app.j2ee.WebModule.
```

In addition, to make things a bit more interesting, in this example, instances are qualified based on the site attribute in the AdminInfo object that is associated with the application server hosting the modules. So, to create a marker rule that associates J2EE Web module marker modules to the application, you would use a query like this:

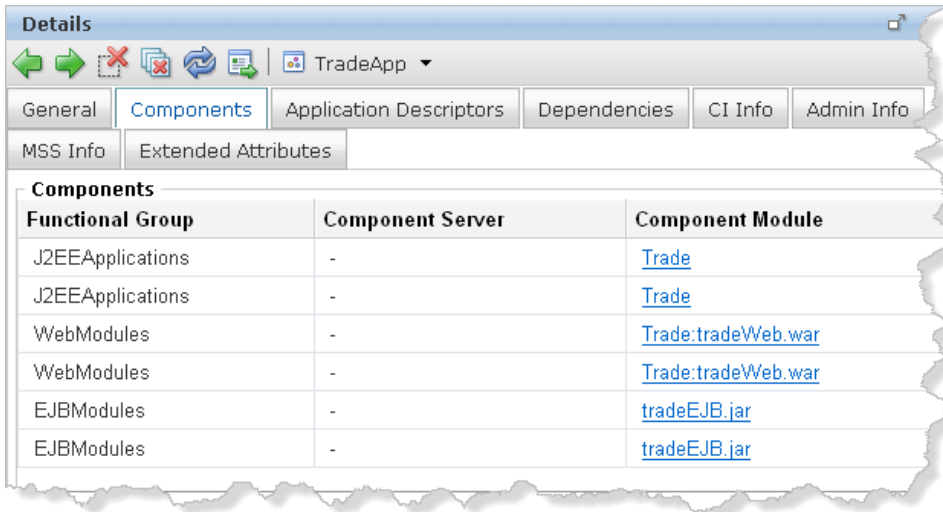
```
Select * from
com.collation.platform.model.topology.app.j2ee.WebModule where
com.collation.platform.model.topology.app.j2ee.WebModule.name ends-
with 'tradeWeb.war' and
com.collation.platform.model.topology.app.j2ee.WebModule.guid ==
J2EEDeployedObject.guid and
AdminInfo.objGuid==J2EEDeployedObject.parent.guid and
AdminInfo.site=='TEST'
```

In this query the WebModule class is used to identify WebSphereWebModule resources. If you want to be more generic you could have used the J2EEDeployedObject resource type, and if you need to be more specific, for example to filter on attributes that are only available at the WebSphereWebModule level, you could have used the WebSphereWebModule class instead of WebModule. If you refer to the CDM documentation, you will find, that you could have used any of the following classes to identify the WebSphere based WebModule instances:

SoftwareModule, J2EEDeployedModule, J2EEModule, WebModule and WebSphereWebModule.

As previously mentioned, to generalize the query, you should strive to use the top-most super class that supports the attributes you need.

Once again, after you apply this rule you will not notice any changes in the application topology in the example environment. Both the already identified WebSphere servers contain the tradeWeb.war module so not new resources are added to the topology. However, if you look at the Components tab of the Details view for the TADDM business application, you see that each of the three rules you have applied return two component nodules - one for each WebSphere server.



Functional Group	Component Server	Component Module
J2EEApplications	-	Trade
J2EEApplications	-	Trade
WebModules	-	Trade:tradeWeb.war
WebModules	-	Trade:tradeWeb.war
EJBModules	-	tradeEJB.jar
EJBModules	-	tradeEJB.jar

As you can see, the functional group that includes WebModules is called: WebModules.

The query used to find application server instances hosting the WebModules could be coded like this:

```
Select * from AppServer where SoftwareModule.displayName ends-with
'tradeWeb.war' and SoftwareModule.parent.guid==AppServer.guid and
AppServer.guid==AdminInfo.objGuid and AdminInfo.site=='TEST'
```

Even though it may not seem like it, this rule produces almost the same results as the J2EEJBMModules rule. In this rule, the SoftwareModule super class is used to identify the marker module. SoftwareModule is the super class of J2EEModule, so in this case, the two can be used interchangeably. In addition, this rule returns an AppServer object. AppServer happens to be the super class of J2EEServer, and the attributes used in this query is supported by both object types.

5.1.4 The Databases marker rule

At this point you have seen how you can use different marker module types and filtering attributes to identify the specific J2EE components or servers you want to include in the application. Now it is time to expand your horizon, and include the back-end database that supports the Trade business application.

As already described the J2EE platform hides the run-time links between the EJBs and the database, so there is no logical path you can follow to locate the database based on the discovered information about the J2EEApplication. From the architecture you know that you are looking for a database named TRADEDB, so you are faced with several challenges:

1. Locate the J2EEServers that host a J2EEResources (EJBs or servlets) that are part of the Trade application.
2. Identify the DBMS systems that host a database named TRADEDB.
3. Find the J2EEServers from step 1 that communicate with the DBMS systems identified in step 2.

To identify the instances that support the Trade application you must rely on a combination of the marker module information (as you just did in the previous marker rules) and the discovered relationships. If you recall, TADDM discovered that both WebSphere servers are communicating with a Db2Instance, but are you sure which WebSphere servers access the TRADEDB database? Unless you use a transaction monitoring and decomposition tool, there is not guarantee, but it would be a fair to assume that if the Db2Instances host a database named TRADEDB, and receives traffic from the J2EEServers, it supports the Trade application.

By using the available information you can piece together a rule that identifies the Db2Instances on which the WebSphere servers depend.

First, to identify the J2EEServers that host the application marker modules you can replicate and combine the relevant criteria used in the previous AppServer rules, or use the J2EEDeployedComponent or SoftwareModule super classes to be able to reference all three types of resource as the same type:

```
(J2EEDeployedObject.displayName IN ('Trade', 'tradeEJB.jar') or
J2EEDeployedObject.displayName ends-with 'tradeWeb.war') and
J2EEDeployedObject.parent.guid==J2EEServer.guid and
J2EEServer.locationTag=='TEST'
```

Next, to identify the DatabaseServer (Db2Instance) in the TEST location that hosts a database named TRADEDB, you can use these criteria:

```
DatabaseServer.locationTag=='TEST' and  
Db2Database.parent.guid==DatabaseServer.guid and  
Db2Database.displayName=='TRADEDB'
```

Notice how the `DatabaseServer.guid==Db2Instance.guid` clause is used to typecast the results.

Finally, to ensure that you only include only DatabaseServers that are communicating with the J2EEServers that host the marker module resources, the following criteria should be added:

```
J2EEServer.guid==Relationship.source.guid and  
Relationship.target.guid==DatabaseServer.guid and RelationshipType  
ends with 'TransactionalDependency'
```

To combine all of this, the final rule used to associate DatabaseServers with the Trade application looks like this:

```
Select * from DatabaseServer where (J2EEDeployedObject.displayName  
IN ('Trade', 'tradeEJB.jar') or J2EEDeployedObject.displayName ends-  
with 'tradeWeb.war') and  
J2EEDeployedObject.parent.locationTag=='TEST' and  
J2EEDeployedObject.parent.guid==Relationship.source.guid and  
Relationship.type ends-with 'TransactionalDependency' and  
Relationship.target.guid==DatabaseServer.guid and  
DatabaseServer.locationTag=='TEST' and  
DatabaseServer.guid==Db2Database.parent.guid and  
Db2Database.name=='TRADEDB'
```

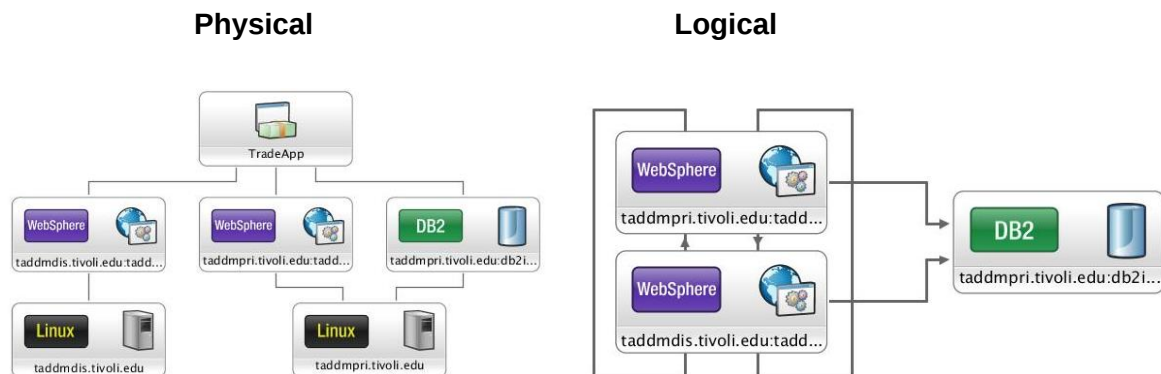
As you can see, the query basically combines the three fragments. Having identified all the relevant marker modules, you can return any resource type that is represented in the query. This means that you could also have used this query to associate the J2EE specific marker modules as J2EEDeployedObjects, but for performance reasons this is not recommended. In addition, you see that the locationTag attribute of the server resources is used to include only resources from the TEST location.

Notice that the query returns a server resource and not a component resource as the other marker module queries. By changing the *from* clause you could have forced the query to return Db2Database resource types to provide consistency among the rules.

Note: During development of this paper, a *feature* of TADDM V7.2.2 has been observed. If you associate Db2Database components to an application using a n MQL rule, the Grouping Composer will not load, and you can no longer edit the application definition from the Data Management Portal. The only way to recover from this is to manually delete the offending MQL rule using the command-line API.

You may wonder why the three J2EE marker modules are referenced in the marker rule for the database. These resources, and their parents have already been associated with the application, so wouldn't it be easier to use reference the already associated resources. The answer is yes – and no. As you will see later (if you keep on reading) you can reference members of a functional group from a rule that populates another functional group, and this will make your rules more dynamic and flexible. However, in this example it was deliberately decided to 'hardcode' the referenced to the J2EE marker modules in order to demonstrate how it can be done, and the use of the IN clause.

If you wait until the BizAppsAgent has executed as part of the bizApps background agent group – or invoke it using the `runtopobuild.sh/bat` script – you will be able to see the updated software topology for the Trade business application. In the test environment, it looks like this:



As you might have expected, both WebSphere servers connect to a single database instance that hosts the TRADEDB database.

5.1.5 The WebServer marker rule

As already discussed already, you can identify the web servers that support the Trade application without having to specify a marker rule. However, if multiple web servers support different sets of applications that are hosted in the same cluster, you may want only associate specific web servers with the application. In order to do so, you must apply a marker rule for the web servers you want to include, and for example use the virtual host names and/or port numbers to identify the specific servers.

The rule needed to include the *frontend.pulse.com* virtual host server that listens to port *11190* and is part of the *192.168.81* subnet (another way to differentiate between environments) can be constructed like this:

```
Select * from WebSVirtualHost where WebVirtualHost.contextIp starts-with '19.168.81.' and
WebVirtualHost.serverName=='frontend.pulse.com' and
exists(WebVirtualHost.bindAddresses.portNumber==11190)
```

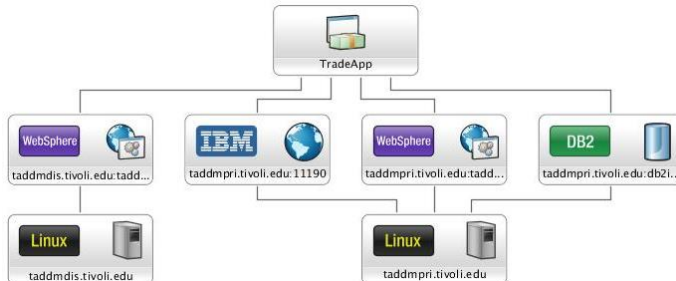
In this case, the virtual host may bind to several ports, so to ensure that you find the correct one, you must use the *exists* clause to select values from multiple members of an array.

If you want to associate the WebServer with the Trade application, as an AppServer instead of the virtual host, the rule should be modified to look like this:

```
Select * from AppServer where WebServer.contextIp starts-with
'192.168.81.' and WebVirtualHost.serverName=='frontend.pulse.com'
and exists(WebVirtualHost.bindAddresses.portNumber==11190) and
WebVirtualHost.parent.guid==WebContainer.guid and
WebContainer.parent.guid==AppServer.guid
```

When you have added the WebServer marker rules, you will see that the topologies have been updated:

Physical



Logical



Note: Be advised that if you use the WebVirtualHost rule to assign the component to the Trade business application, the business application inventory will reflect that the hosting WebServer was added, but neither the WebServer nor the WebVirtualHost will show up in the topologies.

5.1.6 Marker rule summary

As you hopefully can appreciate, the marker rules described in the previous demonstrates how you can use creative rule-writing to associate specific server or component resources with an application, without having to deploy, and manage application descriptor files.

In all the examples, the signature of the marker module (name and type) is used to associate the hosting application server with the application. However, if you want to associate a specific resource, simply change the *from* clause in the MQL Query to indicate the object type of the resource you want to include, and add a clause that links the resource type to the object type that is used in the query to qualify the resource. If, for example, want to associate specific instances of WebSphereJ2EEModules with an application, the MQL query would look similar to this:

```
Select * from WebSphereEJBModule where
WebSphereEJBModule==J2EEModule and
J2EEModule.displayName=='TradeEJB.ear' and
J2EEModule.parent.guid==J2EEServer.guid and
J2EEServer.locationTag=='TEST'
```

If you recall, low-level resources are not shown in the TADDM application topologies or inventories, so from a strict TADDM perspective it does not make sense to associate low-level resources with business applications. If you consider how the information in the TADDM database is leveraged by Tivoli Business Services Manager, Jazz for Service Management, or IBM SmartCloud Control Desk, you will realize that in these situations associating low-level resources with applications does not provide any immediate advantages – unless special business requirements call for it.

5.2 Dependency rules

Dependency rules are rules that use the dependencies discovered by TADDM to identify resources that are related to marker modules, or the application servers hosting marker modules.

For the Trade application you have already seen one example of using relationships to identify related resources. The marker rule used to identify the database uses the discovered relationship to find only the database servers that are actively communicating with the WebSphere servers in the Trade application.

Using relationships you can easily identify resources that consume services from the resources associated with the Trade application, or additional resources that the individual Trade application resources rely on. These may be IT services such as LDAP, NFS, DNS and similar general services, or specific services that apply uniquely to the application environment.

TADDM supports the use of many different types of relationships, which all are unidirectional. Some of the most common relationships you see is the TransactionalDependency, and Uses. The TransactionalDependency relationship type is used when TADDM discovers an active session between the service access points of two application servers. In the Trade application, these types of relationships were discovered between the web servers and the J2EE servers, and between the J2EE servers and the database instance because the application was active at the time of the discovery. You can see the transactional dependencies discovered by TADDM, on page 10.

Because of the way you construct the MQL queries that are used by the rules in the business application definition, you cannot mix and match resource super classes in the query. This means that you must create one dependency rule that applies to server resources that are associated with the application and another rule for component resources. To provide the greatest flexibility, it is also advised that you create specific rules for resources that consume the services of the business application resources, and another rule which identifies resources that provides services to the application resources. In total, four rules are required to include all the resources that communicate with the members of the business application.

5.2.1 The ComponentProviders dependency rule

To associate components in the infrastructure which provide services that are consumed by the components that are associated with the Trade application, you must create a rule that identifies the resources which are the targets of a relationship for which the source must be a component associated with the Trade application. Your challenge is to - once again - identify the application servers hosting marker modules, and reference them as sources of the relationship. Since the marker modules, or the application servers that host them, are already members of the functional groups you defined in the marker module rules, there must be a way to reuse the work that has already been accomplished. And indeed there is.

Referencing members of existing functional groups from rules that populate other functional groups within the same application is pretty trivial, and if you use standardized names for the functional groups within an application, the query can be applied to any application. In addition, since you are working with super class resource types such as AppServers, SoftwareModules, and Relationships, there are no specific situations you need to take into consideration. This implies that once you have created a rule to identify providers to your application resources, you can copy the rule to another application definition and use it without any major modifications.

Here is how it works:

First you have to identify the application you are working with, and the functional groups from which you will identify members that are part of the relationship. The following criteria can be used to identify the groups that are populated with resources that are sub-types of the SoftwareModule type:

```
Application.name=='Trade' and
Application.guid==FunctionalGroup.app.guid and
FunctionalGroup.groupName IN
('J2EEApplications', 'EJBModules', 'WebModules')
```

The criteria above can be used to identify all the members of the named functional groups that you have already populated. You will use this later to identify the members, and these will in turn represent one end of the relationship.

Note: In this rule you should only reference functional groups that contain components, or modules. If you identified appserver resources in the marker module rules that populate the groups, you should can rely on the *server consumes* dependency rule.

Next you must define criteria that detect the relationships you will use to identify the other end of the relationship. In this case, where you are looking for resources that provide services to the application components, the source of the relationships are the members identified by the marker modules. The target of the relationship represents the application server resources you want to associate with the application. So in this case, the following criteria will apply:

```
SoftwareModule.parent.guid==Relationship.source.guid and
AppServer.guid==Relationship.target.guid
```

Now, to treat all the members of the functional groups as SoftwareModules you must use an exists clause. The following criteria specifies just that:

```
exists(FunctionalGroup.members.guid==SoftwareModule.guid)
```

The final query that associates all application servers which are at the receiving end of communications from the Trade application components looks like this:

```
Select * from AppServer where Application.name=='Trade' and
Application.guid==FunctionalGroup.app.guid and
FunctionalGroup.groupName IN
('J2EEApplications','EJBModules','WebModules') and
AppServer.guid==Relationship.target.guid and
Relationship.source.guid==SoftwareModule.parent.guid and
exists(FunctionalGroup.members.guid==SoftwareModule.guid)
```

When you define the rule, an appropriate name for the rule would be *ComponentProviders*, and the name of the functional groups would be the same.

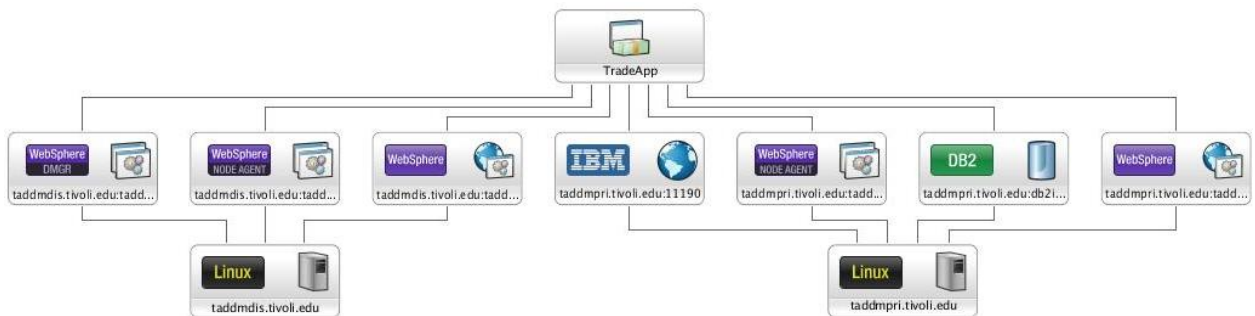
Naturally you can specify special resource types instead of the general SoftwareModule type if you only want to include certain types of resources. You can also add filtering criteria for the relationship itself, for example type or locationTag.

When ComponentProviders rule is applied to the business application definition, you will see that a number of WebSphere related components have been added to the application inventory.

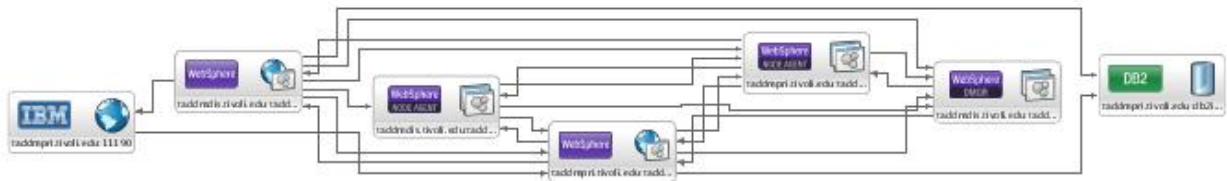
Details		
TradeApp		
General	Components	Application Descriptors
		Dependencies
		CI Info
		Admin Info
		MSS Info
		Extended Attributes
Components		
Functional Group	Component Server	Component Module
WebModules	-	Trade:tradeWeb_war
WebModules	-	Trade:tradeWeb_war
EJBModules	-	tradeEJB.jar
EJBModules	-	tradeEJB.jar
Databases	taddmpri.tivoli.edu:db2inst1	-
WebServers	taddmpri.tivoli.edu:11190	-
J2EEApplications	-	Trade
J2EEApplications	-	Trade
ComponentProviders	taddmdis.tivoli.edu:taddmDISCellManager01:dmgr	-
ComponentProviders	taddmdis.tivoli.edu:taddmDISNode01:nodeagent	-
ComponentProviders	taddmdis.tivoli.edu:taddmDISNode01:TradeServer1	-
ComponentProviders	taddmpri.tivoli.edu:11190	-
ComponentProviders	taddmpri.tivoli.edu:db2inst1	-
ComponentProviders	taddmpri.tivoli.edu:taddmPRINode01:nodeagent	-
ComponentProviders	taddmpri.tivoli.edu:taddmPRINode01:TradeServer2	-

Notice that some of the resources that were already associated with the application are included again. Because the Web Server receives data from the WebSphere serves, it is included by the ComponentProviders rule.

If you take a look at the physical topology, you see that three WebSphere resources that have been identified are the WebSphere Deployment Manager and the two WebSphere Node agents. These are WebSphere infrastructure components that are used to manage the environment, so they do not provide any application specific functionality, but are only used to manage and maintain the environment.



In the software topology (or logical topology) you see that all the relationships between the WebSphere management resources and the WebSphere servers are not only pointing towards the management resources.



You might have expected that since the ComponentProviders only included resources that are targets of the relationships, you would not see any relationships pointing in the direction of the application resources. However, the TADDM topology viewer visualizes all the relationships that have been discovered between two resources, and as you can see, TADDM has discovered that the WebSphere Deployment Manager as well as the WebSphere Node Agents have initiated sessions with the WebSphere Servers.

5.2.2 The ComponentConsumers dependency rule

The rule that includes additional resources which consumes the services of the components that are associated with the business application is very similar to the ComponentProviders rule. All you have to do is to switch the source and target definitions for the relationship so that the application components represent the target side of the relationships, and the resources you want to add to the functional group are the source of the relationship.

The ComponentConsumers rule looks like this:

```
Select * from AppServer where Application.name=='Trade' and
Application.guid==FunctionalGroup.app.guid and
FunctionalGroup.groupName IN
('J2EEApplications','EJBModules','WebModules') and
AppServer.guid==Relationship.source.guid and
Relationship.target.guid==SoftwareModule.parent.guid and
exists(FunctionalGroup.members.guid==SoftwareModule.guid)
```

When this rule is applied, you see that it identifies the same set of resources as the ComponentProvider rule – except for the DBMS system. The reason for this is that in the WebSphere cluster in which load balancing is enabled, the WebSphere Servers report status and performance metrics back to the Deployment manager through the Node agents.

5.2.3 The ServerProviders dependency rule

The two component dependency rules are designed to identify the resources that are related to the components you have associated with the business application. The server dependency rules you are about to define will support the same purpose for the AppServer components that were added through the server marker rules. It should go without saying, that the server dependency rules are very similar to the component dependency rules.

The purpose of the ServerProvider dependency rule is to identify resources to which the server marker modules provide services. Whether these dependent resources are critical to the operation of the application is a matter of architecture, best practices, and design, so you may decide not to include this rule in the business application definition - or apply filtering to include only the resources that are relevant to the actual business application.

If you decide to include it, it should look like this:

```
Select * from AppServer where Application.name=='Trade' and
Application.guid==FunctionalGroup.app.guid and
FunctionalGroup.groupName IN ('Databases','WebServers') and
AppServer.guid==Relationship.target.guid and
exists(FunctionalGroup.members.guid== Relationship.source.guid)
```

Notice that only the two functional groups that contain server components are included, and that the source of the relationship this time is determined directly from the members of the functional groups.

If you apply this rule to the sample environment, the only resources that are identified are the WebSphere Servers that send results back to the Web Server - and these resources are already included in the application.

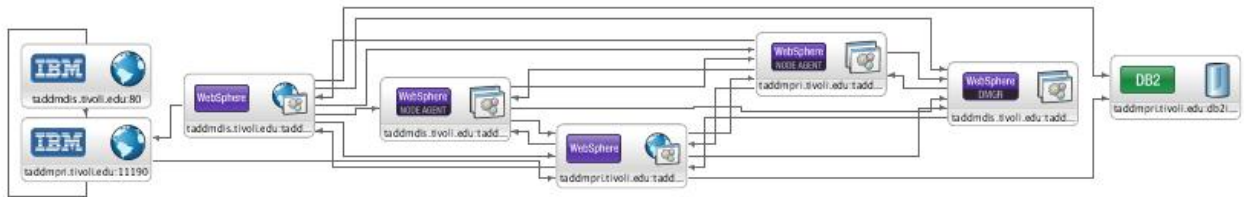
5.2.4 The ServerConsumers dependency rule

Before defining the ServerConsumers rule you need to consider how your environment is architected. As you probably have already anticipated, the purpose of the ServiceConsumers rule is to identify the resources that use the services of the server components you identified through the marker modules. In the case of servers, they may provide services to multiple resources and applications. For example, in the example environment the DatabaseServer used by the Trade application also hosts the TADDM database. If the TADDM processes have been discovered, the ServerConsumers rule will associate the TADDM processes with the Trade application because they use a database on the same Db2Instance. That was not really the plan.

When excluding the Databases functional group from the ServerConsumers rule it will look like this:

```
Select * from AppServer where Application.name=='Trade' and
Application.guid==FunctionalGroup.app.guid and
FunctionalGroup.groupName IN ('WebServers') and
AppServer.guid==Relationship.source.guid and
exists(FunctionalGroup.members.guid== Relationship.target.guid)
```

When this rule is applied to the application definition, and the background agents have had a chance to process the results, the software topology for the Trade application will look like this:



Notice at the far left, how the reverse proxy web server through which external users access the system has been added to the application because it receives data (consumes) from the application web server.

5.2.5 Dependency rule summary

Hopefully you appreciate the use of discovered relationships that allow you to incorporate resources in the business application which are somehow related to the marker modules. Naturally, using dependency rules require that you have given TADDM the opportunity to discover the relationships. This means that you need to discover your resources during production hours so you capture all the relationships that are in use when the application is operational.

Remember that there are no requirements for you to use dependency rules. These are optional and you may have to tailor the rules so that you do not include resources that are related to shared application servers, but not necessarily part of the application.

You will soon realize that your organization best practices most likely prescribe a limited number of patterns or deployment architectures. These patterns can be mapped in business application definitions which can be used as templates when you need to map new applications.

5.3 Service dependencies

In addition to associating marker modules and their related server resources, it is possible that one or more of the resources use common IT Services such as DNS, LDAP, NFS or similar enterprise-wide services.

Some organizations decide to ignore these services when creating business application definitions while others, especially in the ISP and ASP businesses, include services in order to provide the complete picture.

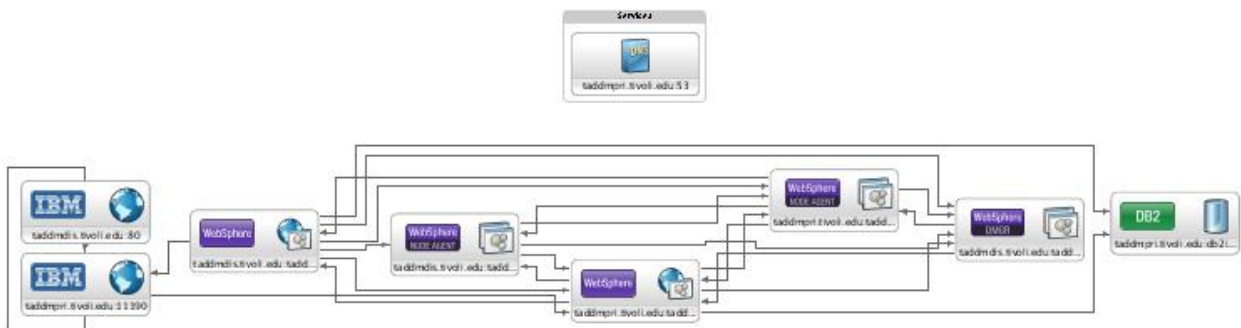
To include services in the application definition, you can once again use the relationships. When TADDM discovers sessions between a resource and a service, explicit ServiceDependency relationships are created. To leverage these, the following rule can be used to identify the services that are used by the resources in the functional groups of your business application:

```
Select * from Service where Application.name=='Trade' and
Application.guid==FunctionalGroup.app.guid and
FunctionalGroup.groupName IN
('Databases','WebServers','ComponentProviders','ComponentConsumers',
'ServerProviders','ServerConsumers') and
Relationship.target.guid==Service.guid and
(Relationship.source.guid==AppServer.guid or
Relationship.source.guid==AppServer.host.guid) and
exists(FunctionalGroup.members.guid==Relationship.source.guid)
```

An appropriate name for this rule - and the functional group it populates - is *Services*.

Notice that only the functional groups that has members that are AppServer are included. As mentioned, TADDM registers relationship based on the communications between Application servers and/or services, so it would not make sense to include the functional groups that contain SoftwareModule members.

When the Services rule has been applied, you will see the services as non-connected nodes in the topologies. It is assumed that they are used by so many components that it will disturb the topology view if all the connections are displayed.



In the topology example above you see only a single DNS system. The example environment includes both LDAP and NFS, but the services provided by these components are not used by the Trade application.

6 Creating application patterns

If you inspect the rules you have created so far, you will quickly realize that only the marker rules contain application specific information. The marker rules are the ones in which you determine which J2EEApplication, EJBModules, WebModules, Databases, or WebVirtualHosts define your application.

Using the 80-20 rule, it is highly likely that most of your business applications adhere to a small number of patterns. Now, imagine that you can replace the hard-coded references to specific marker modules with relationships. If you can do this, all you need to do to bring in a new application is to create the definitions that relate the application to specific marker modules - and clone the appropriate application pattern.

If you use this approach, you can create the application definition in advance of discovery (based on the information you obtain from the application owner and developers). In addition, you can run a scheduled script that creates the necessary relationships as the marker modules are discovered. This way, the work that goes into defining applications is minimized, and automated to the greatest extent possible.

6.1.1 Marker module relationships

The relationships you need to create should adhere to the TADDM standards for allowable relationships between model objects. The names of the standard relationship types used by TADDM are all prefixed by the string `com.collation.platform.model.topology.relation.` and the first character of the relationship name is always in uppercase. Even though there is nothing that enforces this standard, you are encouraged to adhere to it in order for TADDM to be able to correctly interpret the relationship, and cleanup the relationships if either the source or target are deleted.

In the following it is assumed that you create relationships in which the application is the source, and the marker module is the target. However, you can reverse that if you want, but remember that it will affect the type you assign to the relationship. In the documentation for the Common Data Model, you can find the definitions of all valid relationships – and guidance on how to interpret them. For these examples, the *Requires* relationship will be used to relate a marker module to an application. Other valid type are *BasedOn*, *Contains*, *DependsOn*, *are Uses*. If the marker module is the source of the relationship, consider using one of the following relationship types: *AllocatedTo*, *Implements*, *MemberOf*, or *Supports*

Creating relationships

Associated with this paper, a script named `relationshipManager.py` is provided. This script can be used to manage the relationships needed to associate marker modules with applications. For installation and use instructions, please refer to the documentation that accompanies the script.

Note: The `relationshipManager` script uses MQL queries or GUIDs to specify both the source and the targets of the relationships you want to create, so if you are not careful, you can inadvertently create a lot of relationships in a single operation. You are advised to test your queries using the command-line API before you start.

When defining the relationships for the Trade application, you can basically reuse the queries in the marker rules you already defined to populate the functional groups. However, for performance reasons you should only select the `guid` attribute in the queries instead of all attributes (*).

J2EEApplication marker relationships:

To create the Requires relationships needed to relate the two J2EEApplication marker modules to the Trade application, you would invoke the `relationshipManager` script with these arguments:

```
./relationshipManager.py -u administrator -p collation
-a create
-s "select guid from Application where displayName=='Trade'"
-t "select guid from J2EEApplication where
J2EEApplication.displayName=='Trade' and
J2EEApplication.parent.locationTag=='TEST'"
-r Requires
-f
```

EJBModules marker relationships:

In a similar fashion, the following invocation of the relationshipManager script will create the relationships needed to associate the EJB modules with the Trade application.

```
./relationshipManager.py -u administrator -p collation
-a create
-s "select guid from Application where displayName=='Trade'"
-t "select guid from EJBModule where
EJBModule.displayName=='tradeEJB.jar' and EJBModule.parent is-not-
null and EJBModule.parent.locationTag=='TEST'"
-r Requires
-f
```

WebModule marker relationships:

```
./relationshipManager.py -u administrator -p collation
-a create
-s "select guid from Application where displayName=='Trade'"
-t "select guid from J2EEDeployedObject where
J2EEDeployedObject.name ends-with 'tradeWeb.war' and
AdminInfo.objGuid==J2EEDeployedObject.parent.guid and
AdminInfo.site=='TEST'"
-r Requires
-f
```

Databases marker relationships:

To create the relationship that defines the TRADEDB database as part of the Trade application, you can start the relationshipManager script with the arguments shown below:

```
./relationshipManager.py -u administrator -p collation
-a create
-s "select guid from Application where displayName=='Trade'"
-t "select guid from DatabaseServer where
J2EEDeployedObject.parent.locationTag=='TEST' and
(J2EEDeployedObject.displayName IN ('Trade', 'tradeEJB.jar') or
J2EEDeployedObject.displayName ends-with 'tradeWeb.war') and
J2EEDeployedObject.parent.guid==Relationship.source.guid and
Relationship.type ends-with 'TransactionalDependency' and
Relationship.target.guid==DatabaseServer.guid and
DatabaseServer.locationTag=='TEST' and
Db2Database.parent.guid==DatabaseServer.guid and
Db2Database.name=='TRADEDB'"
-r Requires
-f
```

6.1.2 Relationship based marker rules

Now that you have created the relationships that associate the marker modules with the application - in a similar fashion to what is achieved by deploying application descriptor files - you must modify the marker rules so they use relationships rather than hardcoded names. All you need to do is to replace the marker module queries similar to this:

```
Select * from J2EEApplication where
J2EEApplication.displayName=='Trade'
J2EEApplication.parent.locationTag=='TEST' "
```

with:

```
Select * from J2EEApplication where Application.name=='Trade' and
Relationship.source.guid==Application.guid and
J2EEApplication.guid==Relationship.target.guid and
Relationship.type=='com.collation.platform.model.topology.relation.Requires'
```

Notice that you only need to change the where clause, and that the only attribute that is hardcoded is the application name. Naturally you can use any relationship, but you specifically created the application>module/server relationships to associate the marker modules with the application. Using only the application name as the only hardcoded attribute allows you to easily clone or reuse the rule to define another application.

When you have modified all the marker rules in the application definition, and ensured that the BizAgentsAgent background agent has executed, you will see that the business application topologies and inventory are identical to the previous definitions. As a matter of fact, all you have done is to use the original marker module rule to identify a relationship, and replace the rule with a pointer to the relationship.

7 Automating application definition

By now it should be obvious that when you have specified the application patterns that meet the requirements of your organization, you can automate the whole application mapping process using these three steps:

1. Define the application using the appropriate pattern.
2. Identify marker modules, and create marker relationships.
3. Discover resources

You see that you can perform all the preparation work in advance of discovering the application resources. However, this is only possible if you use a mechanism to automatically create marker relationships as new resources are discovered. Such a mechanism can easily be coded, and scheduled to run for example every 60 minutes.

7.1.1 Define the application using the appropriate pattern

After you have defined an application using relationships to identify the marker modules, and making the dependency and service rules generic, you can use it as a pattern to create new applications.

Agreed, TADDM 7.2.2 and prior versions does not include a facility that allows you to clone applications, so unless you find a programmatic way easily clone an application, you need to apply a lot of copying, pasting, and manual updates.

Associated with this paper you will find a script named `groupManager.py`. This script can be used to manage TADDM groups from a command-line, and one of the available functions is the *clone* function. As part of the cloning, you can specify if you want to replace all instances of the old application name to the new application name. Using this option, along with naming standards for your functional groups, and relationship-based association of marker modules to applications, you can create a new application definition in a matter of minutes.

To use the `groupManager` script to clone an existing application named `J2EE_Pattern1` to a new application instance named `PlantsByWebSphere`, you would simply run this command:

```
groupManager -u administrator -p collation -t Application -g  
J2EE_Pattern1 -n PlantsByWebSphere -eR -m -o
```

This command will create a copy of the J2EE_Pattern1 application definition and rules, as well as any related AdminInfo, and UserData (extended attributes), while replacing all instances of the old application name (J2EE_Pattern1) with the new name (PlantsByWebSphere).

For installation and usage of the groupManager tool, please refer to the separate documentation that accompanies the script.

7.1.2 Identify marker modules, and create marker relationships

As discussed previously, you need the input of the application owner, developer or architect to understand which resources constitute the marker modules for a particular application. Even if the application applies to one of your standard, best-practice application patterns, you will need to understand which marker modules support the application in order to create the relationships that links them to the application.

Neither the current (7.2.2) or previous versions of TADDM provide facilities in the Data Management Portal that allow you to create relationships (or dependencies) that include low-level resources. Once again, you need a scripted solution.

You are in luck. Associated with this paper you will find a script named *relationshipManger* that can be used to create your maker module relationships. This script has already been discussed in 6.1.1 *Marker module relationships* on page 37.

7.1.3 Automating marker relationship creation

The relationshipManager script can be used to create the marker relationships you need when it is executed. However, if you want to automate the creation of marker module relationships as new resources are discovered, you need a way to control which marker module relationships to create, and the frequency with which they are created.

To facilitate this you should create your own script that calls the relationshipManager script a number of times to create the marker module relationships you need. Unfortunately TADDM does not provide the means to add custom-agents to the background agent groups, so you have to use the operating system scheduling facility (cron) to schedule the invocation of your relationship creation script for example every 15 minutes.

If you are using TADDM 7.2.2, you should consider using the new dependencies function instead of the relationshipManager script to manage the relationships. This is based on SQL, so you have all the capabilities of the SQL language at your fingertips. Agreed, coding SQL statements require even better understanding of the Common Data Model, and the implementation of the TADDM database, but the dependency management feature in TADDM 7.2.2 can be used to create even the most intricate non-discoverable relationships such as those between a J2EEDataSource and a database. For more details regarding the TADDM 7.2.2 dependencies function, please refer to

http://pic.dhe.ibm.com/infocenter/tivihelp/v46r1/index.jsp?topic=%2Fcom.ibm.taddm.doc_7.2.2%2FUserGuide%2Ft_dmp_manual_create_cidependencies.html?

8 Summary

Most business applications are defined by certain unique marker modules, EJBs, Servlets, virtual hosts, databases, queues and the like. With TADDM, application topologies can be mapped by identifying these application specific marker modules and leveraging the relationships TADDM has discovered between the marker modules, or the components that host them. The application topologies will reveal which marker modules and application server resources take part in the provisioning of the application.

Even though it may seem enticing, it is not recommended to include computer systems, network devices, and storage components in the application definitions. Under the covers, TADDM already keeps track of which computer systems host the application servers, and the dependencies between computer systems and networking and storage devices are used by the computer systems.

Traditionally marker modules are identified through the deployment of so-called application descriptor files to each of the systems in the IT infrastructure. Deploying these application descriptor files to the correct systems, and ensuring that they are discovered by TADDM is a time consuming and error-prone process, which is why the use of application descriptor files has not been widely accepted.

With the dynamic, rule-based application definition functions introduced in TADDM 7.2.1, TADDM has the capabilities to identify marker modules directly from the discovered information and therefore it is no longer required that the files are distributed throughout the IT infrastructure in order to discover the application infrastructure. Every time the application definition rules are re-validated (by default every 4 hours) the application topology is updated based on the information in the TADDM database.

The approach described above requires that identification attributes and qualifying criteria are hardcoded directly in the application definition rules. These attributes and criteria can just as well be used to create an explicit relationship in the TADDM database, and the application definition rules can examine relationships instead of marker modules. This way, the rules become generic, and can be copied from one application to the next almost without any modification. Using marker module relationships, you can define the small number of business application patterns that support 80% of your business application architectures, and have the applications automatically defined and created based on the discovery of the marker modules, and the relevant business application pattern.

Appendix A. Installation

The helper scripts that accompany this paper can be installed either on the TADDM server or on another system where the TADDM SDK is installed. The scripts have been designed to use the jython interpreter that is delivered with TADDM, and must be located in the `bin` subdirectory of any child directory of the TADDMM installation directory. For example, if TADDM is installed in `/opt/IBM/taddm/dist`, you can place the helper scripts in any directory that conforms to the following expression: `/opt/IBM/taddm/dist/*/bin`.

You can place the files in `/opt/IBM/taddm/dist/external/bin`, but since this directory may be replaced if you upgrade your TADDM instance, it is recommended that you create a new directory that contains your own tools and additional packages such as this. An appropriate name for this directory would be `cust`, or `custom`.

If you decide to use a new directory, remember to authorize the users to read and execute the files in the directory and all its subdirectories.

To install the `relationshipManager.py` and `groupManager.py` utilities, simply download the files along with the `taddmJythonApiHelper.py` helper file, and place them in the desired directory. If you are using a Windows platform, you should also download the `groupManager.bat` and `relationshipManager.bat` invocation scripts.

If you are using the linux platform, you must convert the files to unix format, using the `dos2unix` utility. For example, to convert `relationshipManager.py` to linux format issue the following commands:

```
cd /opt/IBM/taddm/dist/cust
dos2unix relationshipManager.py
```

The scripts have been designed to be owned and run as the TADDM instance user (`taddmusr`). So you should ensure that the `taddmusr` group and `taddmusr` group owns the files. To change the ownership, issue these commands:

```
chown taddmusr:taddmusr *
```

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement might not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
224A/101
11400 Burnet Road
Austin, TX 78758 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information in softcopy form, the photographs and color illustrations might not be displayed.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked