ThreeSpaceDesignProposal

"3Space" - a tentative proposal for next-generation DSpace architecture
(M. Simpson, 5/7/2004)

INTODUCTION

Here's some (admittedly sketchy) architectural suggestions, as hashed out by Mike Simpson and Jim+Downing across a series of IC and email communications.
Some of the concepts that this design tries to embody include:

- "programming to interfaces, instead of implementations", as defined in *Design Patterns* (Erich Gamma et al., Addison-Wesley 1995) p.17-18, to simplify development and integration of new components.
- modular architecture, using either the "dependency injection" or "service locator" patterns to simplify interaction between components, as discussed in "Inversion of Control Containers and the Dependency Injection Pattern" (Martin Fowler, http://www.martinfowler.com/articles/injection.html).
- (possible) a "chain of responsibility" design pattern (*Design Patterns*, p.223-232) for certain modules, to increase flexibility in how a given DSpace instance could be configured to respond to client requests.
  The above ideas hopefully contribute to a design that will enable/encourage lots of independent development of functionality by folks in the general DSpace user community, and make sure that independent code contributions can be easily integrated into other DSpace instances.

LAYE ELATIONSHIPS

The "3" in "3Space" comes from the three layers of the architecture.
Each layer is defined by a Manager object, a set of APIs (implemented as Java interfaces), and one or more modules (implemented as Java classes each implementing one or more of the Java interfaces).
There's also (probably) an instance-level object that ties the Managers together, which I've been calling the "scaffold", for want of a better term.
The three layers in the architecture also implicitly define two "boundaries" between layers. The boundaries represent implied APIs as well: the set of objects that may be passed across them as parameters to and returned values from the API calls made by the implementing modules and their Manager objects.
The working terms for the three layers are: "protocol", "service", and "support". The terms for the boundaries are: "protocol/service" and "service/support".
The whole point of the layered architecture is to isolate logically separate chunks of functionality, so that, i.e. someone implementing a module for the service layer doesn't need to know or care about any of the details of the protocol layer above, or the support layer below.
Each module exists entirely within its layer, and sees only its own layer's APIs, and the objects that are allowed to pass the boundaries of the adjoining layers.

THE POTOCOL LAYE

The purpose of the protocol layer is to translate arbitrarily formatted requests from clients of the DSpace system into a neutral internal request format, which is then passed to the service layer for further action; and, upon receipt of the corresponding neutral internal response object from the service layer, translate it back into the appropriate client format.
As a concrete example, DSpace should probably ship with a protocol module to handle HTTP/HTML requests and responses ("HttpHtmlModule").
The Protocol Manager's job would be to instantiate this class, listen on a configurable hostname/port socket for client requests, and hand those requests to the HttpHtmlModule. The HttpHtmlModule would receive those HTTP requests, build a DSpaceequestObject, and then give it to the Protocol Manager for dispatch to the service layer, i.e. something like:
{{{
public interface DSpaceModule { }
public interface ProtocolModule extends DSpaceModule { }
public class HttpHtmlModule implements ProtocolModule

Unknown macro: {private ConfigurationManager config_mgr = getConfigManager();private ProtocolManager proto_mgr =
getProtocolManager();public void initialize_module(){proto_mgr.listen_on(config_mgr.getHostname() config_mgr.getPort());}public String
process request(String request){DSpaceequestObject ds req = new DSpaceequestObject();// ... code to turn the request string received from

```
the// client (i.e. an HTTP request) into the// DSpaceequestObject... DSpaceesponseObject ds_rsp = proto_mgr.service_request( ds_req );
Stringuffer sb = new Stringuffer();// ... code to turn the response object returned from the// service layer back into a String in whatever format// the
client is expecting, i.e. HTML ...return sb.toString();}}}}
}
```

The .getXXXManager() methods above are left deliberately blank - they could be Singletons bound by JNDI, ClassLoader (traditional singleton pattern implementation) or ServletContext. This is the Service Locator pattern - in the dependency injection pattern the component would specify exactly which services and configuration parameters it required for operation, rather than obtaining them through Locators (Managers).
JimDowning

A developer, core or otherwise, might choose to implement a web services interface to their DSpace instance, writing a new module ("HttpSoapModule") to receive SOAP-format requests from clients, and return SOAP-formatted responses to them. The HttpSoapModule could be implemented without having to touch any code, or in fact have any real knowledge of, below the protocol layer. Later, another developer might decide that SOAP-over-SMTP might be useful for certain functions (like bitstream retrieval) and might implement another module ("SmtpSoapModule") based on the original HttpSoapModule, again without having to touch anything below the protocol layer.

THE SEVICE LAYE

The purpose of the service layer is to receive requests (in a neutral internal format) from the protocol layer, make the necessary calls into the support layer to resolve the request and build a response object, and then return that object to the protocol layer.
As a concrete example, DSpace will need to ship with a service module to handle simple object retrieval ("ObjectetrievalModule"). The Service Manager instantiates the object, which then awaits the appropriate request event from the protocol layer (I'm actually not sure who should broker this, the Protocol Manager or the Service Manager, or both? The Service Manager is going to need to have a configurable mechanism for dispatching the various kinds of requests to the appropriate service module – maybe some kind of regex-based system, configured in an XML file?). Upon receipt of a request object, the ObjectetrievalModule calls down into the support layer to service the request – authenticating and/or authorizing as necessary, retrieving the requested asset object, and formatting it into a DSpaceesponseObject which is then passed back to the protocol layer.
More extremely rough pseudocode:

```
{{{
public interface DSpaceModule { }
public interface ServiceModule extends DSpaceModule { }
public class ObjectetrievalModule implements ServiceModule
  Unknown macro: {private ConfigurationManager config_mgr = getConfigManager();private ServiceManager srv_mgr = getServiceManager();
public DSpaceesponseObject service_request( DSpaceequestObject ds_req ){DSpaceesponseObject ds_rsp = new DSpaceesponseObject();//
code calling into the support layer to do, i.e.// session authentication, principal authorization // and object retrieval ... int sid = DSpaceequestObject.
getSessionId();DSpaceSession ds_sess = srv_mgr.getSession( sid );boolean authenticated = srv_mgr.authenticate_session( ds_sess, username,
passphrase );if( ! authenticated ){throw new DSpaceAuthenticationFailureException();}boolean authorized = srv_mgr.authorize_for_retrieval(
ds_sess.getPrincipal(), ds_req.getetrievalPath() );if( ! authorized ){throw new DSpaceAuthorizationFailureException();}DSpaceId did = ds_req.
getObjectetrievalId();DSpaceObject dso = srv_mgr.retrieve_object( did );// ... code to translate the DSpaceObject into the// DSpaceesponseObject
...return ds_rsp;}}}}
}
```

New service modules can be implemented and added to either the core code, or as optional plugin modules, as needs for new services arise. The service module developers can implement these new modules without any knowledge of the layers above or below, i.e. any new service immediately becomes available through any/all of the deployed protocol modules, without extra work on the part of the developer; and changes to the support layer implementations won't affect the existing service modules.

THE SUPPOT LAYE

The purpose of the support layer is to respond to a variety of calls filtering down from the service layer – i.e. for authentication, authorization, session maintenance, asset store maintenance, etc.
Each module in the support layer implements one or more of the Support Layer APIs, and then indicates to the Support Manager upon initialization which methods it will be responsible for supporting.
I.e. some DSpace instances may choose to configure a KerberosAuthNZModule support module that handles both authentication and authorization to a Kerberos-based back end; other instances may only want a KerberosAuthenticationModule (for authentication), but a locally-developed SomeLocalSystemAuthorizatioNModule (for authorization). The idea is to maximize pluggability into arbitrary backend systems, and hide all of specific configuration and implementation details behind the Support Manager, so that other

support modules, or the services layer above, don't need to worry about them.

More pseudocode:

```
{{{
public interface DSpaceModule { }
public interface SupportModule extends DSpaceModule { }
public interface AuthorizationModule extends SupportModule { }
public interface AuthenticationModule extends SupportModule { }
public interface AssetStoreModule extends SupportModule { }
public class KerberosAuthenticationModule implements AuthenticationModule
  Unknown macro: {private ConfigurationManager config_mgr = getConfigManager()\private SupportManager sup_mgr =
getSupportManager();// methods to implement the AuthenticationModule interface ... public boolean authenticate_session()
  Unknown macro: { ... authenticate against a local Kerberosbackend datasource ... }
  // ... }public class LdapAuthorizedAssetStoreModule implements AuthorizationModule, AssetStoreModule{// okay, having a single module deal with
authorization and asset// store management is a contrived example to prove a point. }
}
```

New support modules can be developed either by local sites, to tie into local legacy backends, or more general modules (Kerberos, LDAP) could be developed as part of the core codebase, with configurable options to hook them into local instances of well-known protocols.

Each support module is isolated from changes in the protocol and service layers above it, and independent of what other support modules are deployed by a particular instance's configuration.

CHAIN OF ESPONSIILITY PATTEN IN THE SUPPOT LAYE

The support layer is one place where it might be useful to allow for multiple individual modules to declare their interest in handling API calls. I.e. you might try to authenticate first against a remote Kerberos service, but fall through to a small filesystem-based directory if the Kerberos authentication is unsuccessful; or you might want to try object retrieval from the local asset store, but fall through to a remote object retrieval module for certain object identifiers.

The Apache webserver has a similar setup, where individual Apache modules can register their interest in various phases of the request transaction process, and indicate (via return values from a funtion dispatch mechanism) that they decline to process this particular request (passing it on to the next module registered for that particular phase); that they have handled the phase, and the request should move on to the next phase; or that they have rejected the request, and an error code should be returned to the client; etc.

Under this module, the Support Manager becomes responsible for managing "chains" of modules that have registered their support for various APIs; call priority for individual modules could either be based on position in a configuration file (modules listed first get a chance to respond to an API call first) or on explicitly configured priority levels (i.e. the KerberosAuthenticationModule answers Authentication API calls at a priority of 50, and LocalAuthenticationModule answers calls to the same API, but at a priority of 25).

The "Chain of esponsibility" design pattern, where a manager passes a method call down a chain of object, giving each a chance to respond in turn, is not hard to implement, and would add a lot of flexibility to the support layer.

I don't know if doing something similar at the higher layers is worthwhile – are their times where, for instance, multiple distinct modules might want to implement and respond to a single Service API? Something to think about further.

CONCLUSION

In considering the above, I want to emphasize that individual details like which methods wind up in which APIs, what we call particular objects, etc., isn't at all what I'm trying to describe. I went back and forth on whether or not to leave in the chunks of pseudocode precisely because I didn't want to let implementation details muddy up the overall architectural design. I wound up leaving them in, mainly to emphasize the "writing to interfaces" and "service locator" (i.e. the Manager objects) paradigms.

And of course, this is just One Way To Do It. Maybe we only need the service and support layers, and the protocol layer is more trouble than it's worth. Or maybe the "chain of responsibility" stuff needs to be dropped from the support layer, if it makes the configuration file syntax too horrible.

I do think that the two things we should be striving for are maximum flexibility of deployment (individual instances should be able to easily mix-and-match modules to serve their specific needs, and the core code should ship with a sufficient number of base modules to immediately answer the needs of most of the community); and minimum difficulty of code development and integration (individual module writers should be insulated from side-effects introduced by other module writers – the framework should make all other modules transparent, from the point of view of a single module).

I hope that the design I've described moves us towards those two goals, or at least suggests some directions for further discussion.