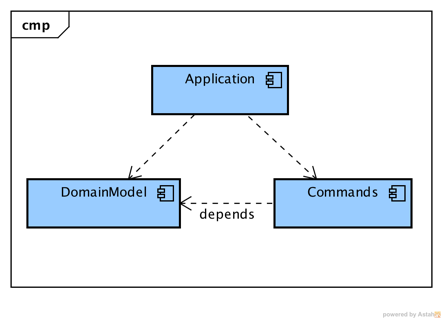
Semantic Server

Application Architecture

# Introduction

At its highest level, the application architecture can be described as three primary component types; facades, Application and data sources. The Application is a singleton. Multiple types and instances of facades and data sources may be deployed.

An Application is composed of a Domain Model and Commands. The Domain Model defines the business model of the Application. The Commands define the capabilities of the Application.

The purpose of a façade is to allow support of alternative clients. A façade is specific to the calling client requirements (capabilities used) and protocol. For example, the current state would include a REST façade. A messaging façade may be deployed when the infrastructure reliably supports messaging. The Application does not change, translation and message patterns are the responsibility of the façade.

A data source is specific to the back-end data store, either a System of Record or a database or another service. The Application invoke data sources as library components through a Service Provider Interface. A Service Provider Interface is a group of (domain) related data access operations.

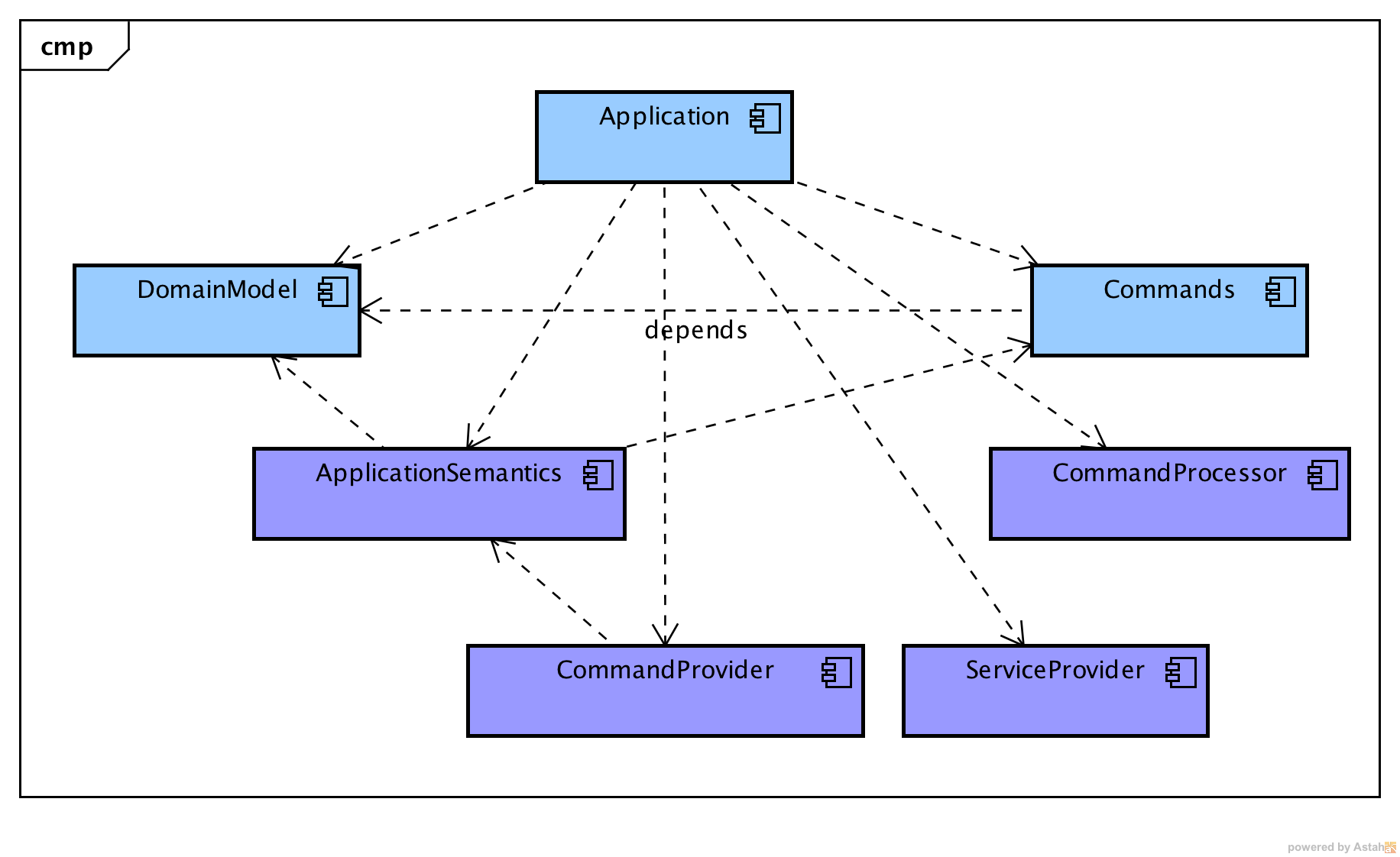
An application is deployed in a dedicated server. Multiple Application could be deployed to a single server, though that would be a move away from small and specific components.

The architecture strictly enforces encapsulation rules between the components:

1. The Application has no dependency on any of the other components.
2. The domain model defines the subject semantics of the application. For example:
   1. Account
   2. Customer
   3. Merchant
3. The commands define the action semantics of the application using a limited vocabulary. For example:
   1. PostAccountCommand – to create a new Account
   2. GetAccountCommand – to retrieve an Account
4. The façades visibility to the Application is strictly constrained to the semantics defined by the Application, specifically the commands and the Domain Model elements.
5. A Data Source conforms to an SPI defined in Application semantics. For example:
   1. AccountID createAccount(Account)
   2. Account retrieveAccount(AccountID)

The architecture, when described at this level, is simple but precepts are intrinsically enforced. For example: encapsulating the client access in a façade serves to isolate changes in interface protocols. Encapsulating business functionality as commands preserves the capabilities independent of client interfaces.

## Deployed Application Structure

The Application consists of the aforementioned Domain Model and Commands along with server defined components. These components are illustrated in the diagram to the right. The components are: Application Semantics, Command Provider, Service Provider and Command Processor.

The Application Semantics are an encapsulation of the naming defined by the Domain Model and by the action (and preposition) vocabulary. Application Semantics are divided into Processor Bridge Semantics and Command Semantics. The Processor Bridge Semantics define the naming of the methods within the Processor Bridge, effectively the naming exposed to Façade components. The Command Semantics define the naming of the Command classes, in essence the entirety of the Application capabilities. The Processor Bridge and Command semantics are constrained by common rules to restrict them to only the allowed vocabulary.

The Command Provider is a factory of available Commands. Each Application has a “Root” Command Provider that manages one or more Application Command Providers. Each Application Command Provider can create a subset of the Application Commands.

The Service Provider Factory is a factory of available Service Providers. Each Service Provider implements one or more Application-specific Service Provider Interfaces (SPI), which are defined in terms of the Application Domain Model.

Command Provider and Service Provider Factory are both implementations of a Service Provider model. If you are not familiar with SPIs or Service Loaders, see Service Providers later in this document for an explanation.

The Command Processor manages the lifecycle of Command instances, including execution of the Command. The lifecycle includes making an application context available to each Command. A Command may be executed synchronously or asynchronously by the Command Processor. Asynchronous execution is limited to Command implementations that are marked as eligible for asynchronous execution.

## Application Semantics

The purpose of defining the application semantics is to simplify mapping application capabilities to implementation. The capabilities of an Application are realized in its Commands. Invocation of Commands is made available to facades in an intuitively consistent semantic.

For example, an Application may implement capabilities such as creating, reading and updating an Account. In Semantic Server terms this would mean that the Application includes commands such as:

* CreateAccountCommand(Account account)
* ReadAccountCommand(AccountID accountId)
* UpdateAccount(Account account)

Command names are strictly constrained to the form:

<action><domain object>[<collection type>][<preposition><object>]Command

where:

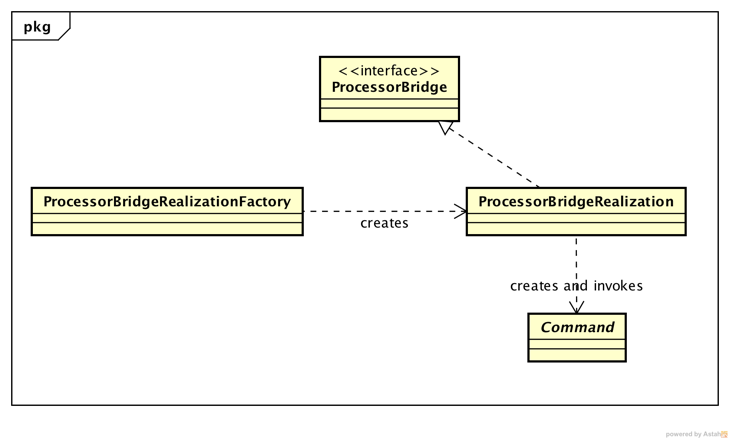
* <action> is one of an enumerated set of verbs defined by this framework
* <domain object> is a simple class name of a class within the Domain Model
* <collection type> (optional) is a simple name of a collection type
* <preposition> (optional, with object) is one of an enumerated set of prepositions
* <object> (optional, with preposition) is a simple name of a class within the Domain Model or within a few standard Java packages
* “Command” suffix on all Command class names

A façade could expect to find methods with signatures such as:

* AccountID createAccount(Account account),
* Account readAccount(AccountID accountId)
* void updateAccount(Account account)

The name, signature and result are constrained by the Application Semantics. For example, the “createAccount” method must resolve to a “CreateAccountCommand” command, which must have a constructor or factory that takes an Account parameter and whose “invoke” method returns an AccountID instance[[1]](#footnote-1).

## Façade to Application Interface

The interface between a façade and the Application is provided by a Processor Bridge. A Processor Bridge is defined by the façade as an interface whose method names and signatures are strictly constrained by the Application Semantics. A Processor Bridge is realized by a Processor Bridge Realization Factory. Each method call in the Processor Bridge is mapped to an available Command. A Processor Bridge realization must implement instantiation of the Command and then submit the Command instance to the Command Processor for execution. The relationships are illustrated in the diagram to the right.

The façade and the Processor Bridge are linked to the Application Domain Model at compile time. Neither the façade nor the Processor Bridge interface are linked to the Commands until execution.

### Processor Bridge Interface

A Processor Bridge Realization Factory orchestrates the resolution of a Processor Bridge interface into a realizing class. Initially this is implemented with a dynamic proxy, runtime class generation may be added later to enhance performance.

During initialization, the calling client calls the createFacade() method, passing the Processor Bridge type as the only parameter. For each method in the Processor Bridge the Realization Factory locates a Command implementation that maps to the method. Once all methods are validated the Realization Factory creates an instance of an invocation handler and a dynamic proxy, which it returns to the client. The invocation handler creates Command instances and submits the Command instances for processing.

While the internals are somewhat complex, from the perspective of a client façade, the code is straightforward:

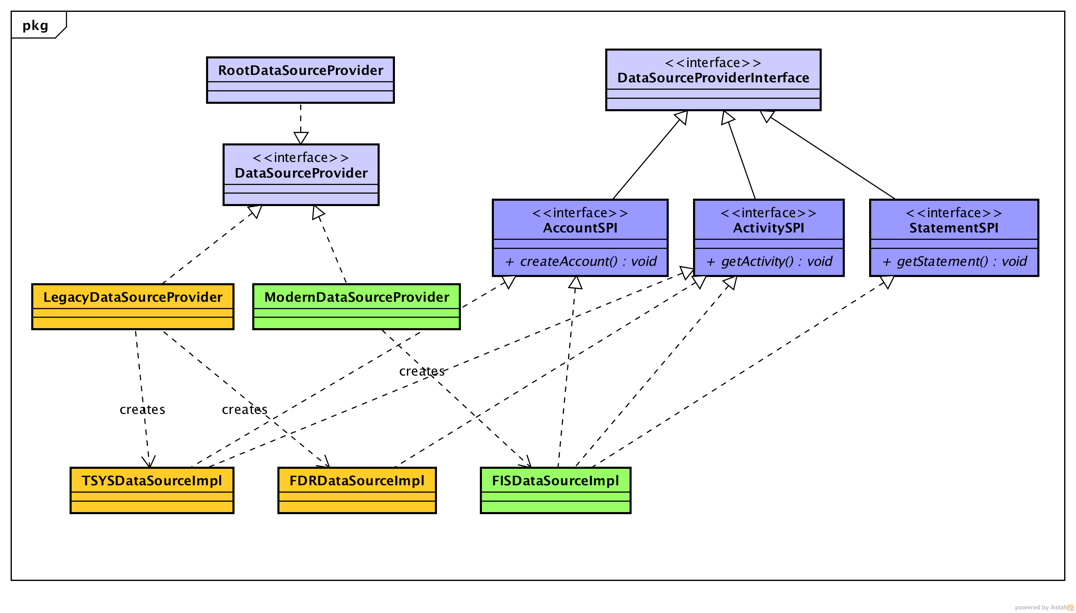
*// create the façade realization*

*ClientSpecificFacade façade = application.processorBridgeFactory.create(ClientSpecificFacade.class);*

*// call a façade method (invoke a Command)*

*AccountID accountId = façade.createAccount(account);*

## Data Source Provider

Data Source Providers are the connection from an application to a data repository. Data Source Providers may connect to databases, services, queue connections, remote RPC calls, etc. Service Providers are defined in terms of an interface and data source. The repository may be identified in business terms (product type, market, etc.) rather than a technical identification.

In the diagram above the packaging is indicated by the component color.

* The light purple are components of the Semantic Server.
* The dark purple are components of the Application.
* The orange are components of a legacy data source provider.
* The green are components of a “modern” data source provider.

Note that the packaging of the data sources is arbitrary, in that a single DataSourceProvider may make multiple DataSource available.

## General Use Command Implementations

Earlier in this document the capability to create Command implementations during execution was alluded to. This capability was developed specifically to address the large number of capabilities that are workflow based.

The WorkflowCommand, and its associated WorkflowCommandProvider, may be configured during execution from a workflow description. The workflow description is an XML file that describes the processors and their connections that comprise a workflow.

Describing the operation is simpler as a concrete example so …

1. The WorkflowCommandProvider is always installed as a CommandProvider.
2. The Root Command Provider findCommands() or createCommand() method is called which, in turn, calls the findComands() method of each of the installed CommandProvider instances.
3. findCommands() is given four parameters, command class semantics, result, parameter types and a routing token (which contains the product type).
4. The WorkflowCommandProvider applies an internal naming convention to the parameters and searches for a workflow definition with the expected name. If the workflow definition is found then it is read and stored locally as an object graph and the WorkflowCommandProvider returns a token, indicating that it can create a Command that satisfies the request. If the workflow definition cannot be found, then no token is returned.
5. The Root Command Provider calls the CommandProvider createCommand() method, one of the parameters being the token returned from the previous findCommand() method. The WorkflowCommandProvider uses the information in the token to create an instance of WorkflowCommand, using the workflow definition to initialize it and then returns the instance.

In effect, a façade may invoke a postAccount() method that, at deployment, is represented only by the Post Account workflow description in a configuration server.

# Service Providers

A Service Provider model separates the definition of a needed service from the implementation and defers the selection of the implementation until execution. A Service Provider model defines an interface, Service Provider Interface or SPI, and a set of selection criteria such as publisher, version or some transaction property.

An SPI encapsulates a group of methods related by the type of data, such as Account information. The selection criteria are defined by the service provider factory implementation in its method signatures, for example Product Type. To continue with this example, application code would then request “a service that returns Account information for Product Type ‘X’” … which is precisely what the “ServiceProvider” class does within the Semantic Server.

Another use of the Service Provider model is in Command Providers. In this case the criteria for selection is the semantics of the façade method along with the parameters, and results. In essence the façade is asking for “a Command that, given an AccountID, will return the account information as an Account instance”.

1. Formally, the createAccount method name, parameter types and result are evaluated by the Command Provider implementations. A Command Provider may create Command realizations at runtime. The Workflow Command, covered later in this document is an example. [↑](#footnote-ref-1)