The Cascading Bridge Design Pattern

Brendan McCarthy
Sun Java Center
brendan.mccarthy@sun.com

Intent

Separate implementation dimensions into independent classes so they can be varied and evolved independently. The benefits of this approach apply in designing within sub-systems as well as architecting across sub-systems. There are some particularly unique benefits for the latter.

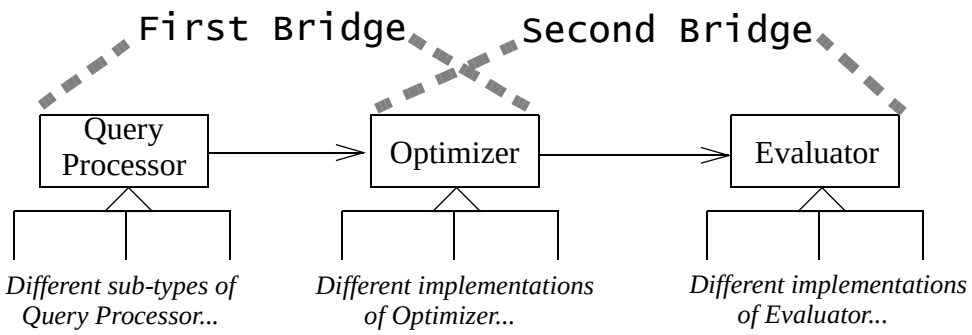
Motivation

The Bridge design pattern [Gamma+95] describes the benefits of decoupling an abstraction from its implementation. In a broader context, there are a number of situations in which it is useful to treat the "implementation" as multiple independent components (implementation dimensions) to which additional decoupling benefits apply. For example, the implementation of a query processing component might consist of an optimizer and an evaluator. While the abstraction of the query processing component remains constant, either or both of the optimizer or evaluator might be swapped out for reasons such as:

- The component is broken or does not perform sufficiently well and must be replaced
- A newer and better implementation becomes available, perhaps from a 3rd-party source
- Parallel development teams are incrementally and independently upgrading each component
- Competing implementations must be tested for comparative robustness or performance

If the implementation dimensions are completely orthogonal they can be referenced directly from the abstraction. However, a well-defined abstraction usually has a high degree of cohesiveness which necessitates that its implementation dimensions interact. As is the case for the query processor component, usually this interaction occurs in tandem -- part of the job of one (the optimizer) is handed off to the other (the evaluator). Put another way, two linked Bridges interact such that the middle component (the optimizer) plays the implementation role in the first Bridge and the abstraction role in the second Bridge:

Figure A) Multiple Bridges

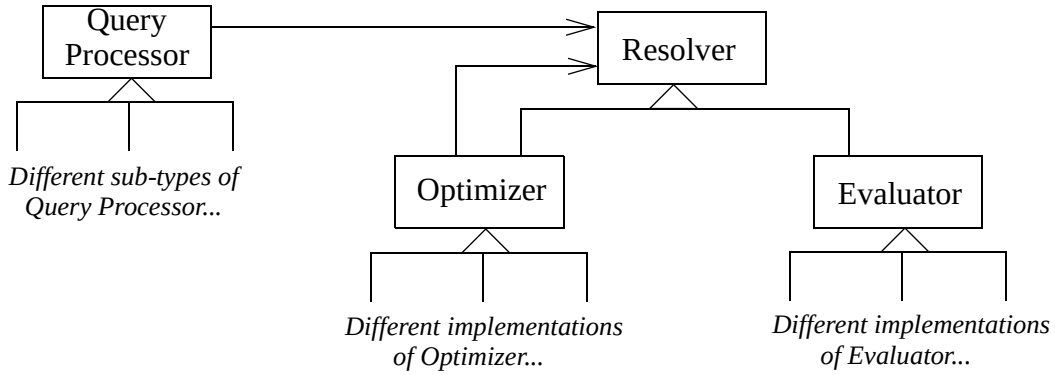


This situation can be described by the *Cascading Bridge* design pattern. Specifically, a Cascading Bridge is the linking of Bridges in a chain such that each component has responsibility for a specific set of transformations, and forwards any remaining work to the next component in the chain. Each component only interacts with its immediate siblings, and has no knowledge of the manner in which those siblings may in turn cascade.

Sometimes it is useful to allow the length and/or ordering of the sequence to vary at runtime. This can be achieved by the introduction of a common abstraction shared among one or components, which can refer back to the common abstraction as their next link in the chain. It is conceptually akin to allowing implementation components to be folded on top of one another, hence we describe this variant as a *Folded* Cascading Bridge as compared to the standard Cascading Bridge (which can also be described as "unfolded" for comparison purposes).

For the example above, we might want to make the use of the optimizer optional. The common abstraction could be defined in base class Resolver, with a recursive link from Optimizer to allow its optional insertion at runtime:

Figure B) Common Base Class



In this model, any number of Optimizer components can be inserted into the chain. Further specializations of Optimizer might split out common optimization techniques such as dynamic indexing, table ordering, or partial evaluation. The same cascading technique can be used to introduce components which transport the request to a different thread or process, perhaps driven by another processor or another machine altogether across a network. At runtime, the required combination of transformations & load could assigned and modified as needed.

The motivation for these patterns is most compelling at an architectural level, in which the implementation dimensions are focused on issues like distributed operation and persistence across whole graphs of interacting objects. The GoF Bridge can be applied here as well, but alone does not address the multiple separation of concerns which must be effectively managed at this level. In effect the domain model(s) of an application becomes the abstraction and the design model(s) becomes the implementation. Whereas all to often these models become entwined to the point where the domain model is difficult to independently recognize and grow, proper structuring with Cascading Bridges enables each to retain its integrity over the life-cycle of a project. The result enhances robustness, re-use, and maintainability.

Applicability

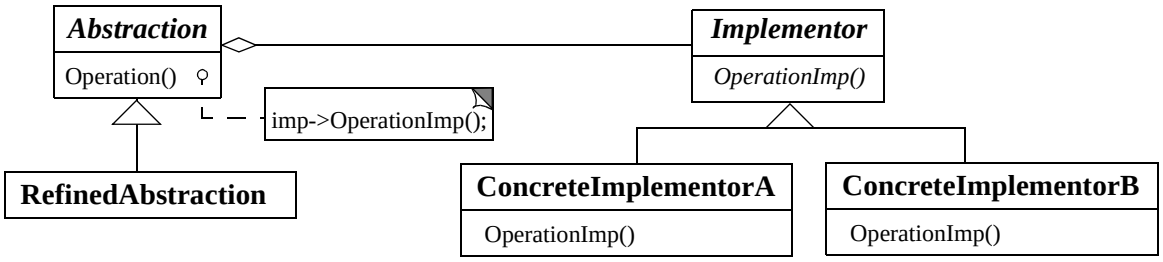
Use a Cascading Bridge pattern like the GoF Bridge pattern but in addition you want to:

- Preserve the modularity of *multiple* implementation dimensions at runtime, maximizing flexibility and opportunities for runtime re-configuration. For example, a persistence component might first be built and run locally. Later, a transport component is built which connects to a remote machine; the persistence component is moved to the remote machine with few or no code changes. Depending on runtime requirements, the persistence component can be run locally or remotely to support off-line or on-line operation, respectively. If folded, the Cascading Bridge can in addition allow for the length and ordering of the components to be varied at runtime.
- Isolate components which may later be replaced, rebuilt, or ported to different 3rd-party libraries. At the architectural level, isolating 3rd-party libraries for key implementation concerns such as persistence (choice of RDBMS or ODBMS) or distributed operation (e.g. CORBA, DCOM, RMI) precludes absolute commitments to technologies or tools before enough experience has been gained to make the best choice. This is often a significant concern in the early stages of a project when teams are facing a quickly-evolving set of technology options.
- Support piece-wise or parallel development of functionality. This is most critical for more complex implementations, especially at the architectural level when for example an initial implementation of persistence might provide limited capability to a flat file while a separate team implements true a DBMS implementation.
- Isolate implementation components to enable functional or performance testing or trialing of different configurations. For example, to isolate a bug in a complex multi-user distributed system, a testing team might proceed by (a) first testing for the bug in single-user single-system mode, then (b) if it persists successively swap in simpler pre-tested implementations of components, and (c) repeat testing until the bug goes away. When it does, there is a strong possibility that the bug exists in the swapped-out component.
- Preserve the integrity of an abstraction even with multiple independent dimensions, especially at the architectural level. This is particularly beneficial when the abstraction is at the level of a logical domain model. The problem avoided is one such as follows: take a clean domain model and make it conform to CORBA specifications; the changes will be extensive, and constrained to available CORBA capabilities. In addition, the selected CORBA library will be exposed to the users of the domain model (typically GUIs), and therefore difficult to later swap out if the need later arises. Finally, extra work will be required to optimize certain methods (especially get/set mutator methods) which don’t have to go over the network each time they are invoked (which is the default way that CORBA client stubs are generated). Even switching among CORBA vendors will be difficult in this scenario.

Structure

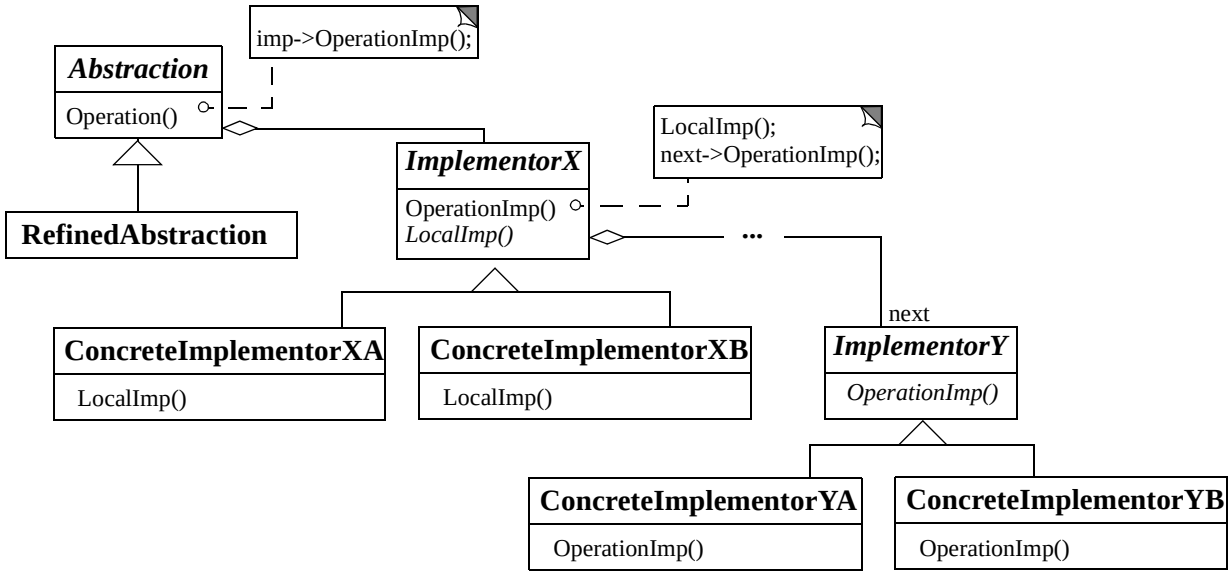
The following diagrams illustrate the patterns described in this paper. First the GoF Bridge is illustrated for reference purposes:

Figure C) GoF Bridge



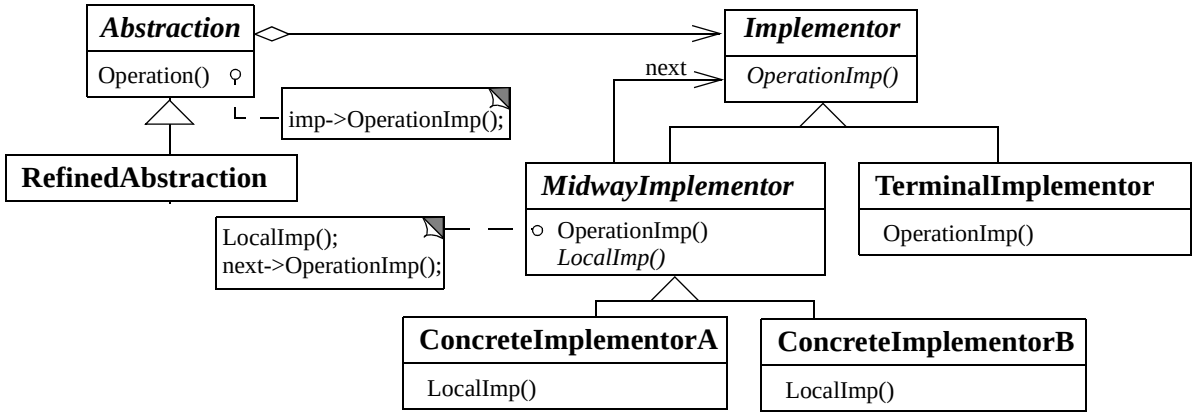
The Cascading Bridge is a series of linked Bridges:

Figure D) Cascading Bridge



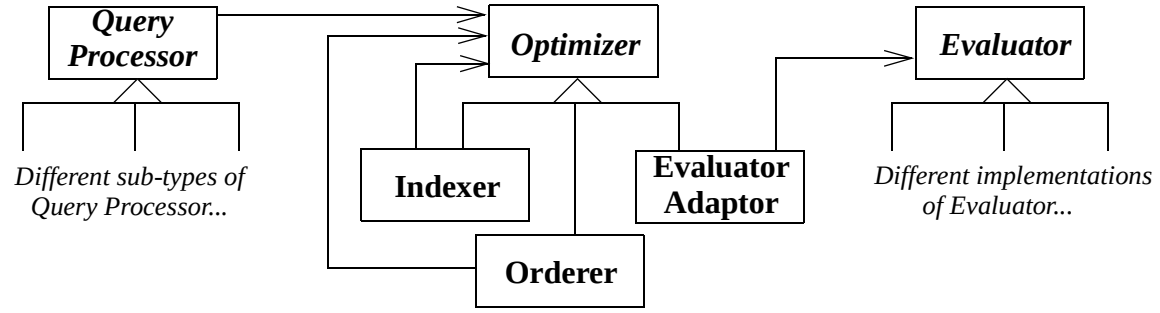
The Cascading Bridge has a recursive link to a common base class (Implementor) which allows its length to vary at runtime. MidwayImplementors perform partial processing than pass on the request, while TerminalImplementors do no further forwarding of the request:

Figure E) Folded Cascading Bridge



The Cascading Bridge might also have a folded sequence embedded in an unfolded sequence, in which case the TerminalImplementor from the diagram above must become and Adapter [Gamma+95]. The following illustrates this in terms of the earlier Query Processor example, in which Indexer and Orderer become the ConcreteImplementors. Additionally, this diagram illustrates a small variation in which the intermediate MidwayImplementor class does not exist and the ConcreteImplementors themselves hold the reference back to the base class:

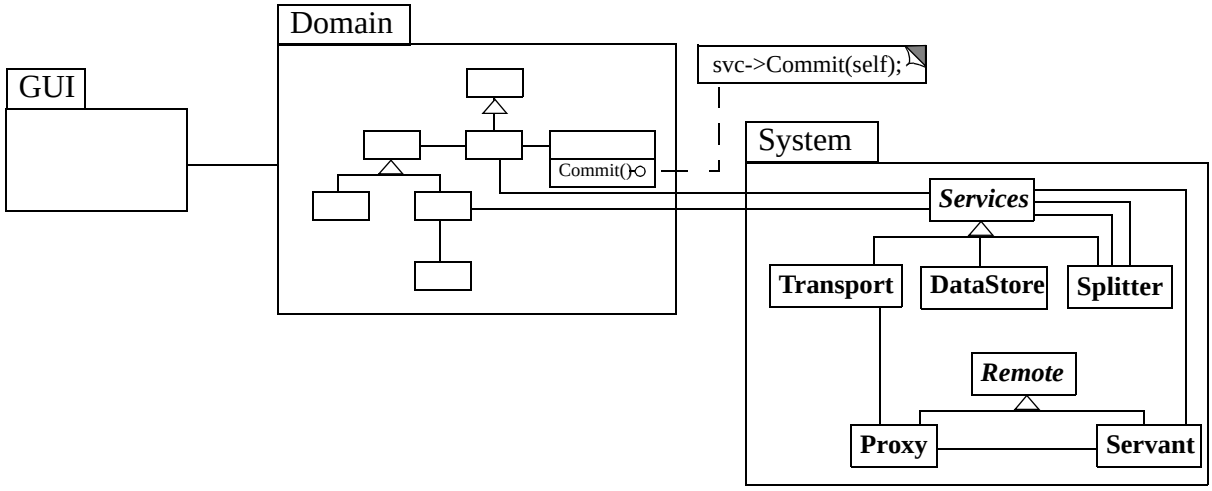
Figure F) Partially-Folded Cascading Bridge Example



At an architectural level, we think in terms of groups of collaborating objects organized in packages [UML97]. At this level the canonical pattern is the GUI package interacting with a Domain

model, such that only key methods on that domain model interact directly with the back-end System package which essentially defines a services layer for the domain model:

Figure G) Architectural Cascading Bridge



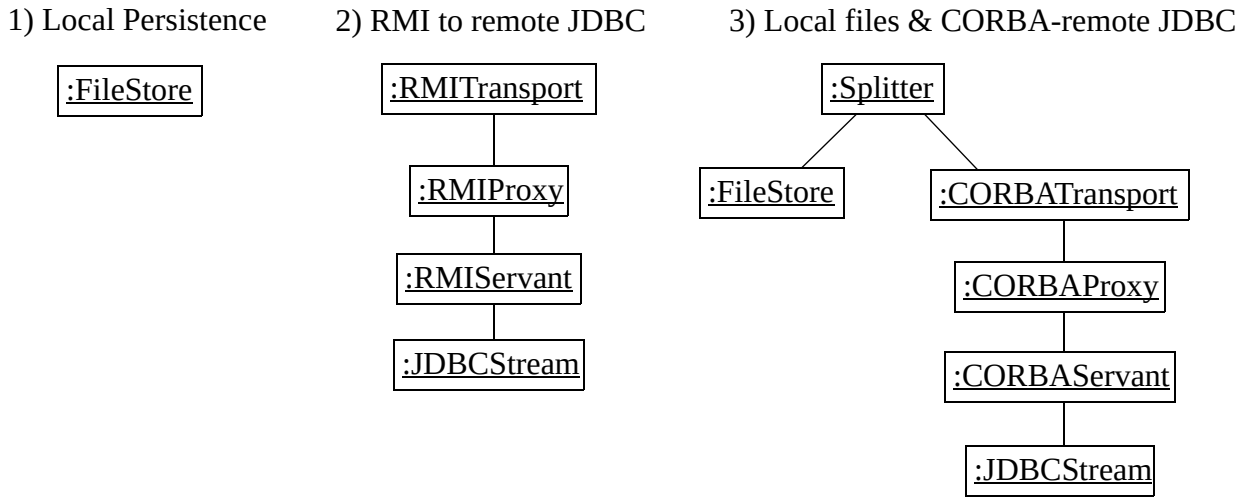
The Domain model is provided only as an example, to illustrate that only some of its methods actually interact with the System package (whose methods should be regarded as computationally expensive -- a good design will have the Domain object model doing as much local work as possible). The Services interface (although it need not be a single interface) defines all the services available to the Domain model. Example Services implementation subclasses are outlined. The DataStore class stores and retrieves its results in some way, while the Transport class interacts with a Proxy [Gamma+95], probably generated from an abstract remote interface [CORBA92, Horstmann98]. Regardless, all Proxy request eventually find their way to the Servant (CORBA terminology) which in this case simply forwards them back to another Services object. A third example Services implementation is a Splitter, which splits the set of Services among more than one (two in this example) target Services objects based on some system need. For example, a Splitter can be used for load balancing, for re-routing requests locally when the remote connection is not operational, or simply for routing incomplete or invalid requests to a local file store where they can later be completed

The Services subclasses are template classes, and should be replaced with versions which provide a specific implementation strategy. Here are some examples:

- FileStore is an example DataStore which reads and writes files
- JDBCStore is an example DataStore which reads and writes into an RDBMS
- RMITransport is an example Transport based on Java's RMI (similarly RMIProxy and RMIServant would be defined)
- CORBATransport is an example Transport based on CORBA (similarly CORBAProxy and CORBAServant would be defined)

The following are some runtime instance examples using these classes:

Figure H) Architectural Cascading Bridge Instance Example



Participants

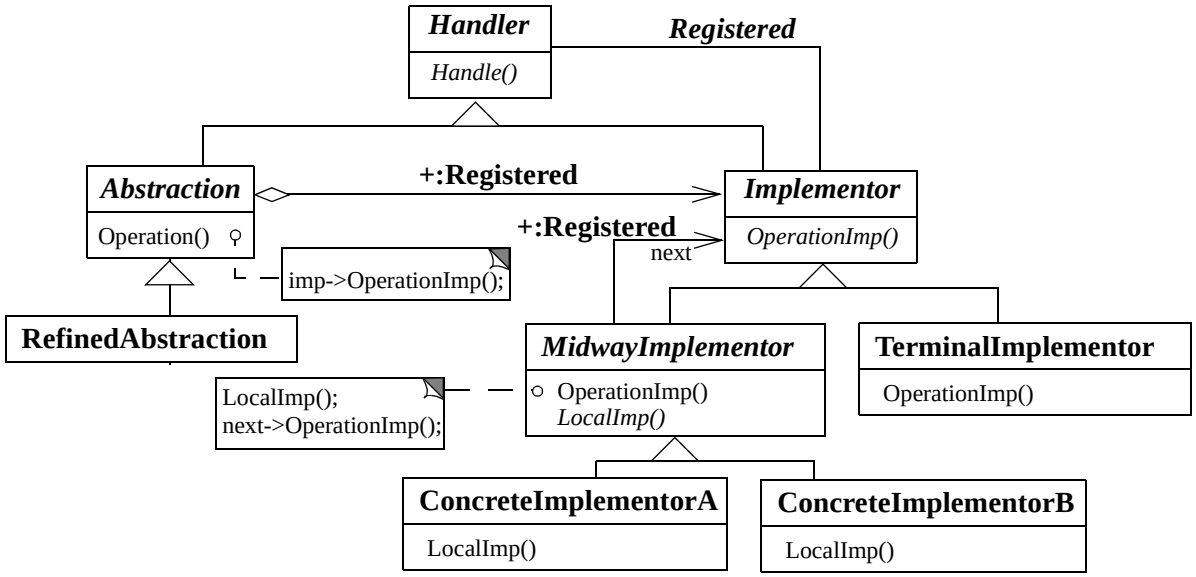
- **Abstraction/Domain Package** presents the interface(s) to a client, just like the GoF Bridge except that here we consider that a an entire Package of collaborating objects can substitute for the abstraction.
- **Refined Abstraction** extends the abstraction. This may or may not exist in the context of a Cascading Bridge.
- **Implementor** presents the generic implementation methods to the abstractions(s). This is just like GoF, except that at an architectural level the methods represent high-level services to the abstraction.
- **ConcreteImplementor** provides all or part of an implementation for the Implementor.
- **MidwayImplementor** provides an implementation dimension then forwards requests to another Services object.
- **TerminalImplementor** implements the Implementor with no further forwarding.
- **Services** defines all services used by the Domain package layer.
- **DataStore** reads and writes to a persistent store.
- **Transport** forwards all operations through a remote communication mechanism.
- **Splitter** forwards all operations among more than one target based on some criterion of interest.
- **Remote** provides the interface for a servant which can also be implemented by a Proxy.
- **Proxy** implements a Remote interface by forwarding all requests over a network.
- **Servant** is the server-side implementation of a Remote interface. In this context, it simply forwards all requests to another Services object.
- **GUI package** is a client to the Domain package layer.
- **Domain package** encloses classes which comprise the business object layer which collectively define clients such as GUI subsystems.
- **Services package** provides the Services interface(s) and all its implementations.

Collaborations

Interactions in both the GoF Bridge and Cascading Bridge are funneled through abstract base classes, forcing the designer to carefully strive for the essential minimum interactions that are required. This forces a strong focus on decoupling which enhances runtime flexibility (among other things). Specific implementation details are contained within implementation subclasses.

The interactions can of course be bi-directional. An example is change notification, such that a mutable data item(s) sits behind the implementors but the Abstraction (or its clients) want to know when that data item(s) changes. In this case, connections among Abstractions and Implementors would be bi-directional, and perhaps many-to-one. Each Implementor would propagate change notification events to every connected (or 'registered') sibling. This introduces the requirement for a common base class among Abstractions and Implementors, which would define abstract methods for receiving change propagation events:

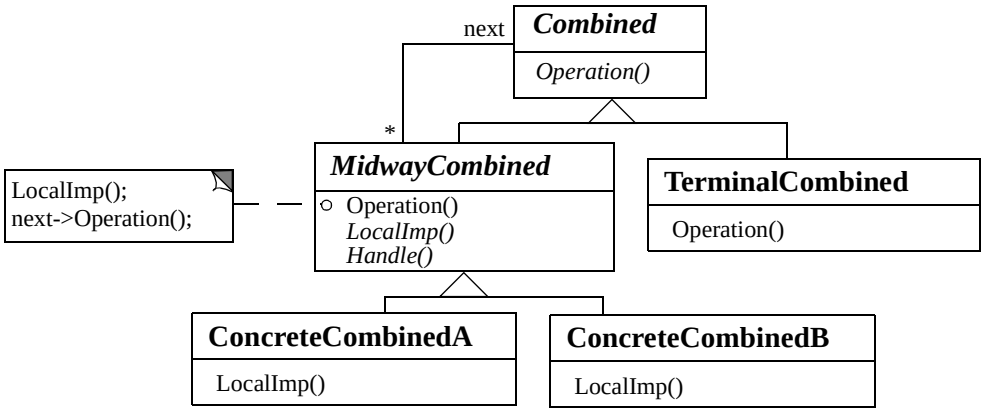
Figure I) Bi-directional Cascading Bridge



This diagram makes use of association inheritance [McCarthy96] to indicate that the association between Implementor and Handler is refined differently when the actual participants are Abstraction/Implementor vs. Implementor/Implementor. The important point is that an Implementor only sees a Handler, and does not need to know whether that handler is an Abstraction or another Handler.

This belies the fact that the simplest structure of all occurs when there is no distinct abstraction. In this case the protocol between Implementors is the same as that presented by the abstraction. Handling bi-directionality in this case is fairly simple:

Figure J) Single-Protocol Cascading Bridge



Consequences

The Cascading Bridge pattern offers the following consequences:

1. *Componentizes implementation dimensions*. Applying Cascading Bridges at multiple levels of granularity enables implementation dimensions to be encapsulated as components which can be swapped at compile time or runtime with minimal impact on the rest of the system.
2. *Preserves independence of models*. At an architectural level, the logical and physical models are maintained independently with well-defined interactions between them. The same goal is the domain of CASE tools, but these have no runtime role. Also those tools still require sustained manual effort to keep both the logical and physical models up-to-date as the system evolves; it is common that only the physical model gets updated because it is the only one from which code gets generated.
3. *Exposes implementation dimensions at a configuration level.* Once implementation dimensions are broken out into different components, application code must assemble them correctly. For some purposes this might be seen as a burden which did not exist with monolithic implementations. If this is the case then Abstract Factories [Gamma+95] should be introduced to hide the details of configuration.
4. *Introduces runtime overhead*. Moving the interaction between components to the generic level of abstract base classes may hinder opportunities for optimizations at a low level. To mitigate this cost, Cascading Bridge should only be considered when (a) the granularity of the computation for each component is considerably larger then the hand-off between components, or (b) the components can be run on different processors which offsets the hand-off overhead, or (c) runtime performance of the chain of components is not critical (e.g., in the context of systems which spend most of their time accessing relatively slow networks and/or secondary datastores).

Implementation

Consider the following issues when implementing a Cascading Bridge:

1. *Protocol transformation*. For the Cascading Bridge of Figure E, it is assumed that every Implementor sees the same object input. This may be difficult when objects are transferred to another process space, since most languages do not natively support shipping objects by value across process spaces. Java is an exception, and natively supports serialization of any object graph to and from a text stream; CORBA 3 will provide a similar capability. Even for those cases, the expense of applying a general-purpose serialization algorithm may not be acceptable. An optimized serialization may perform much better.
   One solution is to insert an Adaptor between the abstraction and initial implementor. The purpose of the Adapter is to transform the object representation to and from a compact stream. An Adaptor pair may also be nested within a larger Cascading Bridge, allowing an optimized data format to exist over a segment of the pathway.
   Regardless, from a speed-of-development perspective the right choice may to initially use standard serialization where available. After (and only after) it is determined through performance testing that the standard serialization is indeed a bottleneck, work on an optimized protocol can be undertaken.

2. *Local computation*. The logical model for the Cascading Bridge of Figure E may perform extensive computation without consulting the Services package. At the simplest level, this means that get and set methods are handled locally. At more complex levels, various derived calculations or data validations may be independently performed. It should be assumed that passing a request to the Services level is relatively expensive, since some form of I/O is typically performed by the Services implementation.

Sample Code

The architectural-level Cascading Bridge of Figure G presents the most interesting variations. We start with a simple model of just a single Application class with only a name and id attributes (the latter is the primary key). Its Java definition is:

```
public class Application implements java.io.Serializable {
  ... // define the name and id attributes, as well as get and set methods for these

  void commit() {
    AppStore.current().commit(this);
  }

  static Application retrieve(int id) {
    return AppStore.current().retrieve(id);
  }
}
```

The Application class implements java.io.Serializable so that it may be serialized (so that it may later be passed by value using RMI). Its methods which commit and retrieve applications simply

forward the operation onto the current AppStore handler object. AppStore is an abstract class defined as follows:

```
abstract public class AppStore {
  public static AppStore current() {return current;}
  public static void install(AppStore as) {current=as;}
  private static AppStore current;

  abstract public void commit(Application app);
  abstract Application retrieve(int id);
}
```

It has methods for maintaining the current handler (current, install), as well as abstract methods provided for the benefit of the domain model. A simple implementation of AppStore stores Applications in memory:

```
public class AppMemStore extends AppStore {
  private Hashtable apps = new Hashtable();

  public void commit(Application app) { apps.put(new Integer(app.getId()),app); }

  Application retrieve(int id) { return (Application)apps.get(new Integer(id)); }
}
```

A simple application would use an AppMemStore directly:

```
public class Client {
  static void init() { AppStore.install(new AppMemStore()); }

  static void test() {
    Application a1 = new Application(1,"barnie");
    a1.commit();
    Application a2 = Application.retrieve(1);
    ... // compare a1 and a2
  }

  public static final void main(String[] argv) {
      init();
      test();
  }
}
```

If instead a remote connection is desired, the Client.init routine simply installs an appropriate subclass of AppStore:

```
public class Client {
  static void init() { AppStore.install(new AppRMI()); }
  ... // same as before
}
```

AppRMI communicates to the server using RMI. Here we assume class AppRemote has been defined and had a client stub generated with standard RMI development tools. AppRMI then forwards requests to an AppRemote instance (potentially having to deal with network failures which are ignored here):

```
public class AppRMI extends AppStore {
```

```
  AppRemote remote;

  public AppRMI() {
    ... // RMI setup details and exception handling omitted
    try {
      remote = (AppRemote)Naming.lookup("theAppServer");
    } catch (Exception e) {}
  }

  public void commit(Application app) {
    try {
      remote.commit(app);
    } catch (Exception e) {}
  }

  Application retrieve(int id) {
    try {
      return remote.retrieve(id);
    } catch (Exception e) {
      return null;
    }
  }
}
```

On the server side, the AppRemoteServer class (not otherwise shown) provides the actual implementation of AppRemote. AppRemoteServer accepts an AppStore argument to which it will delegate requests; we'll pass it an AppMemStore:

```
public class Server {
  public static final void main(String[] argv) {
    try {
      ... // RMI setup omitted

      AppRemoteServer ars =
        new AppRemoteServer(new AppMemStore());

      Naming.rebind("theAppServer",ars);

      System.out.println("server is ready...");
    } catch (Exception e) {
      ... // handle server failure
    }
}
```

In effect, we have shown two variants. First we stored applications in memory on the client. Then we stored the applications in memory on the server, using RMI to connect the client and server. With the Cascading Bridge structure in place, switching between these variants is a simple matter.

Related Patterns

Cascading Bridge builds on the GoF Bridge concept by focusing on multiple rather than a single implementation dimension. In so doing the level of focus is drawn from smaller-scale design to large-scale application architecture.

Like Cascading Bridge, the Pipeline [Vermuelen+95] and Pipes and Filters [Buschmann+96, Shaw+96] design patterns suggest a partitioning of transformations into separate components which can be dynamically recombined. However, their focus is on unidirectional transformations of stream-based data. In Pipes and Filters, a filter is a component which accepts a data stream input from a pipe(s), transforms it, and passes to another pipe(s) the transformed result. There is little allowance for an extended bi-directional verb set of type-safe collaborations. Correspondingly, there is no thought to abstraction vs. implementation as in Bridge and Cascading Bridge. Pipelines are essentially the same as Pipes and Filters, although different authors describe the distinction in different ways.

Another pattern which describes an 'abstraction' at the level of a graph of interacting objects rather than a single object is Facade [Gamma+95]. The 'real' model is hidden by the Facade gateway. With Cascading Bridge, the logical model presented to clients can in fact be a Facade to another model. The intermediate layer the Facade speaks to could be a conduit for transmitting the Facade to the real model where it can be processed. This variant combination can be described by the Facade 'floating' between the client world and the back-end (which might have multiple facades defined for it), and is thus referred to as a 'Floating Facade'.

Bibliography

[Buschmann+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. "Pipes and Filters." *A System of Patterns*, Wiley 1996.

[CORBA92] *The Common Object Request Broker: Architecture and Specification*, Wiley 1992.

[Gamma+95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995.

[Horstmann+98] Cay Horstmann, Gary Cornell. *Core Java Volume II*, Sun Microsystems Press 1998.

[McCarthy97] Brendan McCarthy. "Association Inheritance and Composition." *Journal of Object-Oriented Programming* 10,4 (July/August 1997).

[Meunier95] Regine Meunier. "The Pipes and Filters Architecture." *Pattern Languages of Program Design*, J. Coplien and D. C. Schmidt, Eds., Addison-Wesley 1995.

[Shaw+96] Mary Shaw, David Garlan. *Software Architecture, Perspectives on an Emerging Discipline*, Prentice Hall 1996.

[UML97] The Unified Modeling Language, Rational Software Corp, www.rational.com.

[Vermeulen+95] Allan Vermeulen, Gabe Beged-Dov, Patrick Thompson. The Pipeline Design Pattern, OOPSLA '95 Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems.