Feature-Oriented Object Construction

Tom Woodfin
6.898 Final Project

Abstract

The creation of objects in a class-based polymorphic language like Java is a common source of both nominal and functional dependencies. Although patterns such as the Abstract Factory and Factory Methods have been introduced to reduce these dependencies, implementing these patterns often requires extensive infrastructure. I propose and implement a more general solution which eliminates these dependencies while requiring less effort on the part of both the library and application programmer and providing greater potential for extensibility.

Background and Motivation

The common idiom for object creation in Java involves invoking a constructor belonging to a concrete class and assigning the resulting object reference to a variable of an appropriate interface type that is satisfied by the class. This prevents future uses of the object from relying on the instantiating class and thus reduces potential dependencies. For example, when a Map object is required, it will be instantiated as follows:

```
Map myMap = new HashMap();
```

Future uses of myMap will be required by the type system to only use features of the Map type, rather than that of HashMap. As a result, a future class implementing the Map class more efficiently (say FastHashMap) can substitute for the original simply by changing the instantiation:

```
Map myMap = new FastHashMap();
```

Of course, this problem does not eliminate all dependencies between the application and the HashMap() class. There remains the nominal dependency at the point of instantiation. Multiple instantiations of the same class will require additional dependencies—and thus additional updates if it is desirable to replace the HashMap class.

Further, these dependencies are not limited to the name of the instantiating class. Java's construction operation is parameterized, and multiple constructors can exist for each class. To return to our Map example:

```
Map bigMap = new HashMap(4096);
```

This creation instance invokes a constructor of HashMap which takes an integer parameter specifying the initial number of entries to allocate for the mapping. The result is a functional, as well as nominal, dependency on the HashMap class.

One popular means of addressing these dependencies is the use of the AbstractFactory and Factory Method patterns, discussed extensively in the Gang of Four's *Design Patterns*. The essential "trick" of these patterns is substituting method invocation for constructor invocation in object creation. The result is a dependency on the Factory object rather than the ultimately instantiating class. This allows library providers, who have access to the internal workings of the Factory object, to control which class and constructor actually performs the instantiation.

Unfortunately, the Factory patterns suffer from several drawbacks. Most notably, it requires substantial infrastructure on the part of the library provider. For example, the DocumentBuilderFactory in the latest Java XML library provides the following documentation for its factory method:

```
public static DocumentBuilderFactory newInstance()
                                      throws FactoryConfigurationError
```

Obtain a new instance of a `DocumentBuilderFactory`. This static method creates a new factory instance. This method uses the following ordered lookup procedure to determine the `DocumentBuilderFactory` implementation class to load:

- Use the `javax.xml.parsers.DocumentBuilderFactory` system property.
- Use the properties file "lib/jaxp.properties" in the JRE directory. This configuration file is in standard `java.util.Properties` format and contains the fully qualified name of the implementation class with the key being the system property defined above.
- Use the Services API (as detailed in the JAR specification), if available, to determine the classname. The Services API will look for a classname in the file `META-INF/services/javax.xml.parsers.DocumentBuilderFactory` in jars available to the runtime.
- Platform default `DocumentBuilderFactory` instance.

Once an application has obtained a reference to a `DocumentBuilderFactory` it can use the factory to configure and obtain parser instances.

Throws:

`FactoryConfigurationError` - if the implementation is not available or cannot be instantiated.

Note the involved process by which the factory locates the instantiating class. Even if much of this procedure could be modularized, it still represents a substantial investment by the library implementer to eliminate creation dependencies. In addition, it relies heavily on somewhat esoteric features of the Java class library to locate the implementing class. In light of this complexity, it is unsurprising that Factory patterns have yet to permeate the entire Java API.

In addition, use of the Factory patterns merely substitutes one dependency for another. Although a hypothetical MapFactory would obviate the need of the programmer to be aware of, and create a dependency on, the HashMap class, it would substitute a required knowledge of and dependency on the MapFactory. While this may seem a minor concern, it indicates that the burden for the programmer is not nonexistent when using Factory patterns. The problem gets worse when the user requires an object that polymorphically satisfies two disparate types. The HashMap class, for instance, implements both the Map interface and the Serializable interface. The latter has nothing to do with maps, but rather is a part of the Java I/O package which specifies that HashMap objects can be converted into byte streams. How can the programmer requiring a Map that can be serialized across the network request such an object from the MapFactory? Perhaps there is a SerializableMapFactory in the documentation. Or is that a MapSerializableFactory? What about a Map that can be accessed concurrently as well as be serialized? The combinatorial possibilities boggle.

Finally, the Factory patterns fail to address the functional dependency introduced by parameterized constructors. The functional dependency is merely transferred from the instantiating class to the factory.

Summary and Evaluation

I propose a solution to these problems which is both simpler and more general than the Factory patterns. Rather than an individual factory class for each type, this solution provides a single “factory,” the Object Manager, with the responsibility of mapping user creation requests to constructors on concrete classes.

Much of the simplicity of the approach is facilitated by Java’s reflection capabilities. Methods are available to query a class’ superclass as well as the interfaces it implements. By recursively reflecting on the results of these queries, it is straightforward to determine the entire set of types supported by an instance of the class. The Java language specification even insures that this process can be performed naïvely without checking for cyclic implementation, as this is prevented at compile-time. Similarly, reflection allows a class’ constructors to be queried and referenced.

Constructor objects can be used to instantiate objects of their containing class, and specify the types of their required parameters, though not their names.

Thus this scheme requires that the library provider submit to the Object Manager only a Class object, representing the class to be provided, and a mapping between names and its constructor parameters. This process is called "registration." This would be simplest to do in a class static initializer, although it could be performed by anything with access to an instance of the class. Thus the scheme allows libraries designed without the Object Manager in mind (such as the standard Java classes!) to be registered with the Object Manager just as any other class.

In fact, the registration process could be performed quite dynamically, with classes being made available and replaced at any execution time. Identical creation requests at different points in the program could produce different results. More advanced factory implementations make this possible, but not in a standard or simple way (again refer to the documentation displayed above). This is also a substantial advantage in many situations over the approach of static tools such as Jiazzi, which only allow the instantiation dependency to be resolved at "link time."

Reflecting on the information available from all registered classes, the Object Manager constructs the following internal relations:

TypeImplementations:	type →* class
AvailableConstructors:	class →* constructor
ConstructorParameters:	constructor →* parameter
ParameterInfo:	parameter →! <name, type>

The **TypeImplementations** relation maps types (that is, classes or interfaces) to a set of registered classes which implement that type (formally, classes whose instances can be cast to that type without a run-time exception being generated). The **AvailableConstructors** relation maps each registered class to its public constructors. The **ConstructorParameters** relation maps these constructors to their parameters and **ParameterInfo** maps each parameter to a tuple of its name and type.

Each request for object creation consists of a set of requested types and a mapping of parameter names to typed values.[1] The Object Manager attempts to satisfy the request by performing an intersection operation on the sets of classes mapped by the **TypeImplementations** relation for each type. The resulting classes are those whose instances implement all the requested types. In this fashion it is straightforward to request, as discussed earlier, a serializable Map. This feature provides much of the flexibility of the scheme. Arbitrary characteristics of concrete classes, such as performance or concurrency constraints, could be noted by empty interfaces. Non-empty interfaces could provide generic mixin-like functionality.

With the set of type-appropriate classes in hand, the Object Manager uses the **AvailableConstructors** relation to derive a set of constructors from the classes. The requester-provided map from parameters to typed values is used to determine which constructors can actually be invoked. These candidate constructors are those whose parameters are a subset of those available in the map. An ideal implementation would select the candidate constructor whose parameters most closely matched (by type and number) those available in the map, probably using Java's usual overloading rules. As these rules are somewhat complex, the initial implementation will select an arbitrary constructor amongst the candidate constructors with the greatest number of parameters. In the trivial case where an empty map is provided in the creation request, only a nullary constructor can be invoked. Because constructors are chosen dynamically based on the parameters present in the requester-passed map, there is not a direct dependency on a particular constructor.

Implementing the Object Manager was surprisingly simple. The implementation essentially follows the outline here, using the Java Collection classes to model the required relations. One concern with this approach is obviously performance: in the initial implementation each object creation request requires potentially expensive set intersections. However, I believe this overhead could be substantially reduced with a small amount of caching. Unless the state of the Object Manager relations is changed, identical requests will result in identical constructors

[1] For the curious, Java's reflection capabilities nicely handle the awkward distinction between primitives and their object-type equivalents, allowing users to pass exclusively object types.

being invoked. Most applications generate only a (relatively) small variety of objects, and even a cache of a few thousand recent requests would require negligible lookup time. Dynamic modification of the registered classes, as discussed above, would reduce the efficacy of caches, but only applications which required this dynamism would pay a performance penalty.

Another difficulty with this approach is the syntax for object creation requests. Java does not provide a straightforward syntax for creating sets or mappings, requiring the creation of Set and Map objects and awkward repeated use of add() and put() methods. One alternative is to use the string and array types for which Java provides more syntactic support. In fact, this implementation uses an array of Strings to represent the set of types requested. Without operator overloading or a general composition operator, however, the syntax of object creation requests remains awkward.

Finally, using language features to implement this mechanism necessitates breaking the type system. It is impossible for the Object Manager to return anything more specific than an Object in response to a creation request. It is then the responsibility of the requester to cast the Object into an appropriate type. This is, again, somewhat awkward. However, there is great potential for elimination of unnecessary run-time checks if the Object Manager can be trusted to return objects that support all the requested types, as any future cast of this object to one of the requested types would then be guaranteed to be valid.

Lessons Learned

It is clearly possible to implement an object creation mechanism more general and powerful than the Factory patterns in Java. Several features of Java work to our advantage here:

- Reflection
  Without reflection, the onus is on the library programmer to determine every potential type which his class supports. For deep subclasses, this is not a trivial exercise. In addition, reflection provides the required access to constructors that Java's lack of function pointers would normally preclude.
- The Collections API
  Java's Collections classes proved invaluable in implementing this mechanism. Although missing some desirable features (notably here an intersection operation on sets), the straightforward interfaces and high-performance provided by Collections made writing much of the Object Manager straightforward.
- A well-structured type hierarchy
  As noted in the summary, the Java Language Specification's rules on inheritance and interface implementation ensure that a class implements a well-defined set of types. It is not clear whether such definition is maintained by languages like ML, which support functional operations on packages.

However, some "features" of Java affect our implementation in less desirable ways:

- Rigid syntax
  Without operators supporting set or map construction or a means to override existing operators, our implementation is forced to either use an awkward creation request syntax, perform computationally-intensive parsing of strings or arrays which can "stand-in" for more appropriate syntactic constructs, or require a preprocessing of code with a custom tool to make appropriate syntactic substitutions.
- Lack of rich type conversion semantics
  Many of the type casting operations required by the Object Manager should be "guaranteed" to be safe, and thus not incur a run-time cost. However there is no way to express this in Java, and we are left at the mercy of the compiler or run-time optimizer.

To me, the most surprising lesson learned while implementing this system is how straightforward it is to create a general object creation mechanism that relies exclusively on specification rather than implementation. Java types can fill the role of specifications quite nicely, particularly when interfaces are used to denote features that might not be apparent from method signatures alone, and objects having multiple interfaces can be created easily.