

Reflective Class Search

Brett Daniel

An application uses reflection to search for classes that satisfy certain criteria without initializing explicit references to instances of the classes.

1. Motivation

Consider a batch processing module in a hypothetical business system. It contains a large number of batch process objects that are triggered by a batch process manager object. Processes are independent of each other and the order in which they are executed does not matter. At scheduled times throughout the day, the batch process executes all batch processes that are due to run. A possible implementation of the batch process manager is illustrated in the following C#.NET class:

```
public class BatchProcessManager {  
  
    /// <summary>  
    /// Runs all batch processes that are due  
    /// </summary>  
    public void runDueProcesses() {  
        List<BatchProcess> processes = new List<BatchProcess>();  
  
        processes.Add(new MailClientAccountReport());  
        processes.Add(new MailInvestorReport());  
        processes.Add(new RemoveInactiveClients());  
        // many, many more...  
        processes.Add(new UpdateClientFees());  
  
        foreach (BatchProcess process in processes) {  
            if (process.IsDue()) {  
                process.RunBatch();  
            }  
        }  
    }  
}
```

This naïve implementation is less than ideal for several reasons. First, `BatchProcessManager` should be loosely coupled to the batch processes. The developer should not have to change `BatchProcessManager` when he or she adds or removes batch processes. Ideally, `BatchProcessManager` would be unaware at compile time which batch processes are implemented. Also, the list is so long (imagine about a hundred batch processes) that it overshadows the important aspects of `BatchProcessManager`'s code and probably uses a great deal of memory.

2. Applicability

The batch processing module described in the motivation exhibits several general characteristics of many business applications.

- The application has a large number of classes that are used in a same way and have similar (but not necessarily identical) interfaces.
- A core processor needs to reference a large portion of the available classes during each of its executions.
- The target classes are independent (that is, one's execution does not adversely affect any others').

3. Solution

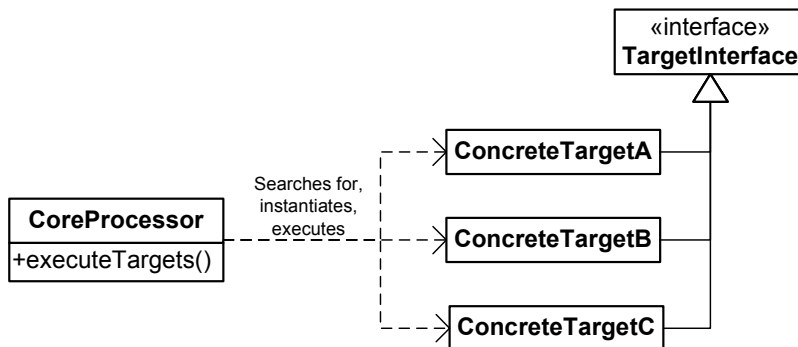
To solve the problems described in the motivation, one can create a core processor that performs the following steps:

1. Loop through the classes implemented in the application or one of its modules.
2. For each of the classes found, test the class' interface against some predefined criteria.
3. If the class meets the criteria, create an instance of the class.
4. Execute some predefined methods on the instance.

The core processor uses reflection for each of these steps (on the last, reflection is optional and depends on the class testing method). Reflection is the programming technique in which an application can examine and in some cases modify its structure at runtime. Here, reflection is merely used to query and execute existing code; no modification takes place.

3.1. Testing Classes

There are three ways in which the core processor can test the classes that it finds. The most straightforward is a **formal interface test** in which the core processor tests whether a class implements a particular formal interface specification or inherits from a particular base class.



The core processor must use reflection to find and instantiate the object, but method execution proceeds normally.

Classes found using an **informal interface test** need not implement a declared interface as in a **formal interface test**. Instead, the core processor uses reflection to test against a class' name, method names, method signatures, or other characteristics of the class. It only accepts those classes that, say, have at least one method whose name begins with "batch". After finding an acceptable class, the core processor must then use reflection to both instantiate the object and call the appropriate methods.

Finally, Java annotations and .NET attributes offer a way for developers to "mark" objects, methods, and other program elements with custom metadata. When performing a search using a **metadata test**, the core processor accepts only those classes marked with the appropriate annotations or attributes. Then, as with an **informal interface test**, it must use reflection to instantiate the object and call the appropriate methods.

4. Considerations

The given solution provides two major benefits over the implementation described in the Motivation section.

- The core processor is loosely coupled to the target classes. A developer need not modify an initialization list when adding or removing a target class. He or she merely creates the class in an expected location and constructs it such that it meets the test criteria
- Memory usage will be lower than explicit construction because target objects are instantiated one at a time, then abandoned to the garbage collector.

However, the solution poses some inherent dangers.

- Not all languages support reflection on the module, class, and method levels.
- It becomes difficult to enforce the object search criteria explicitly since reflection uses run-time rather than compile-time characteristics.
- Instantiating objects using reflection is much more computationally expensive than explicit instantiation. Similarly, calling methods reflectively is much more expensive than an explicit call.
- The core processor's code does not explicitly reference the classes being used.
- Developers must be aware of the search criteria when creating new target classes or modifying existing ones.
- Object instantiation can become especially difficult if the target classes have different, non-empty constructors. Often, target classes must have some predefined constructor signature that necessarily becomes part of the criteria used to test classes.

If any of these considerations conflict with the application's requirements, then some form of explicit instantiation is needed.

The core processor can search for classes every time it is executed, or it can search once and cache the result. The choice depends on the core processor's lifespan, how frequently the core processor is executed, the search criteria used when testing classes, and how often the list of target classes changes.

A designer must take care when choosing the testing method. If the target classes can implement a common interface or inherit from a common base class, then searching using a **formal interface test** is a good choice. Changes to the interface are enforced at compile time, preventing classes that implement the interface from being silently excluded from a search and ensuring that method calls remain valid. Also, the core processor invokes methods directly on the objects, so there is no cost associated with reflectively invoking methods.

Searching using a **formal interface test** is not a good choice when target classes can have a varying number of similar methods or when it does not make sense in the context of the domain for all of the target classes to implement a common interface. Both an **informal interface test** and a **metadata test** address these issues. An **informal interface test** allows for multiple methods because the core processor can execute any method whose name matches the specification. Similarly, in a **metadata test**, the core processor can execute any method marked with the correct

metadata. A search using either an **informal interface test** or a **metadata test** is generally slower than one using a **formal interface test** since method invocation relies on reflection.

Metadata attached to classes and methods are checked at compile time to ensure that the metadata types have been defined; they are attached to the correct programmatic elements; and they are well-formed. The naming conventions required for an **informal interface test** have none of these assurances. In addition, metadata can offer a much richer description of a program element than method names alone. For these reasons, many applications that used an **informal interface test** in the past are updating to use a **metadata test**. However, not all languages support this feature.

Finally, it must be noted that the three testing methods are not mutually exclusive, nor is one inherently better than another. As the sample code shows, a core processor usually combines testing methods based on the application's characteristics.

5. Sample Code

The following C# .NET example illustrates how the batch processing module described in the Motivation can implement this solution.

All batch processes implement the following interface:

```
public interface BatchProcess {
    bool IsDue();
    void RunBatch();
}
```

The IsDue method depends on some state such as the current date or a flag in the database. RunBatch executes the batch process.

A **formal interface test** is the obvious choice, but the designer does not want the core processor to execute every class that implements BatchProcess. Instead, he or she defines the following attribute that will be attached to batch processes that should be executed automatically.

```
/// <summary>
/// Attribute for batch processes that should be run automatically
/// </summary>
[AttributeUsage(AttributeTargets.Class)]
public class AutomaticBatchAttribute : Attribute { /* ... */ }
```

The AttributeUsage attribute ensures that the custom AutomaticBatch attribute can only be attached to classes.

The developers then use the interface and attribute to define their batch processes.

```
[AutomaticBatch]
public class SyncGovernmentRecords : BatchProcess {
    bool BatchProcess.IsDue() {
        //...
    }
    void BatchProcess.RunBatch() {
        //...
    }
}
```

The BatchProcessManager acts as the core processor. It performs a combination **formal interface test** and **metadata test** that retrieves only those classes that implement BatchProcess and are marked with AutomaticBatch. Then, it instantiates an instance of the batch process and calls the appropriate methods.

```

public class BatchProcessManager {

    /// <summary>
    /// Searches for and executes all due batch processes
    /// </summary>
    public void RunDueProcesses() {
        // Get types to test for BatchProcess-hood
        System.Type[] declaredTypes =
            System.Reflection.Assembly.GetExecutingAssembly().GetTypes();
        BatchProcess curBatch;
        System.Reflection.ConstructorInfo constructor;
        foreach (System.Type foundType in declaredTypes) {
            if (IsBatchProcess(foundType)) {
                // get empty constructor
                constructor = foundType.GetConstructor(System.Type.EmptyTypes);
                // create instance
                curBatch = (BatchProcess)constructor.Invoke(null);
                // execute the batch process
                if (curBatch.IsDue()) {
                    curBatch.RunBatch();
                }
            }
        }
    }

    /// <summary>
    /// Checks if the given type is a concrete class marked with
    /// the BatchProcess attribute that implements the BatchProcess
    /// interface
    /// </summary>
    /// <param name="foundType">The type to check</param>
    private static bool IsBatchProcess(System.Type foundType) {
        // Retrieve only concrete types
        if (foundType.IsAbstract || foundType.IsEnum || foundType.IsInterface) {
            return false;
        }
        // Check if the type is marked with the expected attribute
        if (foundType.GetCustomAttributes(typeof(BatchProcessAttribute), true).Length > 0) {
            // Check that the found type implements the desired interface
            foreach (Type interfaceType in foundType.GetInterfaces()) {
                if (interfaceType == typeof(BatchProcess)) {
                    return true;
                }
            }
        }
        return false;
    }
}
}

```

6. Known Implementations

Several widely-used frameworks implement a solution similar to the one described above.

- .NET's web services framework [<http://msdn.microsoft.com/webservices/>] uses a **metadata test** to search for all classes marked with the `WebService` attribute containing methods marked with the `WebMethod` attribute. These classes and methods are then made available for SOAP requests over the web.
- JUnit 3 [<http://junit.sourceforge.net/junit3.8.1/>] used an **informal interface test** to find and execute unit test methods that began with the word "test". However, it did not perform any search for classes that implemented such methods.
- JUnit 4 [<http://www.junit.org/>] and NUnit [<http://www.nunit.org/>] both use a **metadata test** to find unit test methods. NUnit also uses a **metadata test** to search for classes that implement unit tests.