ISSN: 2754-6705

Journal of Mathematical & Computer Applications

SCIENTIFIC Research and Community

Review Article

Open Access

Plugin Architecture in Java and Python

Nilesh Jagnik

Mountain View, USA

ABSTRACT

Supporting plugins in software projects can drastically improve its usability, while also making the code easier to manage and test. The plugin architecture can be implemented with or without the help of dependency injection frameworks. In this paper we discuss providing plugin support with the help of a dependency management framework called Guice in Java. We also discuss how this can be done in Python only using language features.

***Corresponding author**

Nilesh Jagnik, Mountain View, USA.

Received: January 06, 2022; **Accepted:** January 13, 2022, **Published:** January 20, 2022

Keywords: Usability, Extensibility, Plugins, Dependency Injection, Reflective Programming

Introduction

Many software tools and products allow extension of their functionality by supporting third party contributions that add to the functionality of the tool. This is commonly observed in web browsers and programming IDEs. However just about any software can be built to support pluggable functionality. This includes both locally running applications and cloud services.

Plugin architecture refers to designing systems that can support plugins. There are special considerations required when designing a system that supports plugins. Certain design patterns should be followed to ensure that plugins are easy to build, test and deploy. In this paper we discuss these design patterns and considerations.

We then discuss ways to achieve the plugin architecture in Java and Python. In Java, we can utilize popular dependency injection frameworks to support plugins easily. In Python, although dependency injection frameworks exist, we discuss a method to support plugins that does not use an external framework.

Benefits of Plugin Architecture

Extensibility

Supporting plugins in your application or service significantly boosts its usability. It allows software to evolve as user needs change. New features can be added easily. Similarly, new integrations with other software can also be provided with ease. This allows for a lot of flexibility in the usage of software. Plugins can be utilized in this manner not only by third party developers but also by the software owners too.

Clear Ownership Model

Software owners can clearly document plugins which are developed and maintained by third parties. This allows feature requests and bugs to be directed accordingly. Statistics like usage and endorsement counts can also be displayed to allow users to make an informed decision while choosing plugins.

Additionally, software owners may choose to provide some official plugins which they would develop and maintain themselves. This separation allows for easy project management for all plugin owners.

Separation from Framework Code

Software owners can clearly define a contract related to plugin behavior. They can then develop the core functionality of their software assuming this contractual agreement. Failure to uphold these requirements could lead to errors which could be attributed as a plugin failure, rather than a framework failure. This allows framework owners to focus on the functionality of the core framework without worrying about details specific to plugin implementations. Similarly, it allows plugin developers to focus on development of implementation of plugins without knowing details about framework architecture.

Code separation also allows plugins to be tested independently of framework code and vice versa. This leads to healthier code.

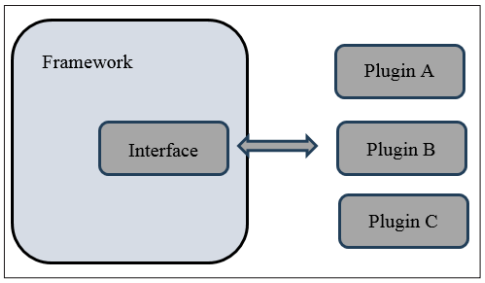

Figure 1: Plugin Architecture

PluginArchitecture

Let us discuss the general setup in a framework that supports plugins.

Framework and Plugin Interface

The framework owner defines a contract for plugins by the means of an interface. The framework code should not make any assumptions about the plugin beyond what is specified in the plugin interface. It is good practice for the framework owner to write several different plugins to provide default functionality and also to test that the framework works currently with different plugins.

The careful structuring above allows development of plugins using the defined plugin interface.

Registering Plugins

After developing plugins, the framework must allow a way to register these plugins. This makes the framework aware of the presence of the plugin. All plugins must be registered with a unique id that allows the users of the framework to specify the plugins they want to use.

Dependency Injection

Dependency injection is a technique that can be used to satisfy a larger set of requirements than what is needed for the plugin architecture. However, dependency injection frameworks are readily available in many languages. Leveraging these frameworks makes it easier to implement a plugin architecture.

What is Dependency Injection?

In Object Oriented Programming, the programmer is required to write code that builds objects. This is normally done by calling the constructor of an object. This constructor could have some dependencies on other objects and thus it might need to create these objects by calling their constructors. As the system grows in functionality, the construction logic may get more and more complex. The construction logic gets even more complex when different objects need to be constructed in different ways based on flags or other state.

Dependency injection solves this problem by decoupling object creation logic from the rest of the code. The constructor of an object only specifies the dependencies it needs and does not actually construct these dependencies itself. This separates the concerns of object creation from object behavior.

```
/** Plugins are operators on two integers. */
public interface Operator {
  int operate(int left, int right);
}
```

Dependency Injection for Plugins

Dependency injection frameworks separate object creation logic from object functionality. We can utilize this to develop core framework logic without the concern of how the dependencies are created. We can take this one step further by also removing the concern of what type of object is created. This is done by specifying a plugin dependency as an interface or abstract class. Then based on flags or user specified configuration, the object creation logic can decide which concrete object to inject. The plugins should obviously implement the interface to qualify as a valid dependency. This allows development of framework code by specifying only a plugin interface as a dependency. At runtime, the real plugin injected can vary based on user specifications.

```
public class Add implements Operator {
  public int operate(int left, int right) {
    return left + right;
  }
}

public class Multiply implements Operator {
  public int operate(int left, int right) {
    return left + right;
  }
}
```

How Dependency Injection is Implemented

Although dependency injection frameworks can used without needing to know how they work, in languages like Python, we have fewer options for dependency injection frameworks. Knowing about the inner workings would help us develop a lightweight version for supporting plugins.

There are two language features that are utilized by dependency injection frameworks. These are reflection and annotations.

```
public class PluginModule extends AbstractModule {
  public void configure() {
    Multibinder<Operator> opBinder =
        Multibinder.newSetBinder(binder(),
            Operator.class);
    opBinder.addBinding().to(Add.class);
    opBinder.addBinding().to(Multiply.class);

    // bind other plugins here
  }
}
```

Annotations

Annotations can be added to classes and objects attaching metadata to them. This metadata could be of many forms, but commonly contains information that could allow making runtime decisions about the usage of these classes and objects. Decisions about which dependency is appropriate to inject can also be made with the help of these annotations. In the later sections, we will see how this works in Java and Python.

Reflection

Reflection or Reflective programming allows a program to introspect and manipulate the internal properties of a program. For example, in Java it is possible to obtain the name of an objects class and annotations. Using the names of object types along with extra supporting information provided by annotations allows dependency inject frameworks to make decisions regarding which dependency is the right one to inject in various situations. In practice, a dependency injection framework could use user specified input or configuration to make these decisions.

```
public class Calculator {
  @Inject
  Calculator(Set<Operator> operators) {
    // Framework code can execute one or more
    // plugins based on runtime inputs
  }
}
```

Plugin Architecture in Java Using Guice

In Java, plugin support can be implemented with the use Guice (pronounced Juice), which is a popular dependency injection framework.

Plugin Interface

To start, an interface should be defined for the plugin. The interface sets expectations that the core framework has from each plugin. In this case, the plugin should be an operator that operates on two

integers and returns an integer.

Plugin Implementations
Let us consider a couple of implementations of the interface above. These will implement the operate() method in different ways.

Registering Plugins
The next step is to let the dependency injection framework know about existence of these plugin implementations. In Guice, this is done via creating a Guice module and specify bindings.

Framework Code
The core framework code can specify plugins as a dependency. It can then decide which plugins to execute depending on flags and other runtime inputs. The @Inject annotation tells the dependency injection framework that a dependency must be provided by it.

Python
There are many data science and Machine Learning applications where Python is the language of choice. Developing extensible software for these applications requires building a plugin architecture in Python.

Dependency Injection in Python
There are several available dependency injection frameworks for Python. However dependency injection is not as popular in Python as it is in other languages. Since migration to a framework may not always be an option, we discuss a way to achieve the plugin architecture using Python language features only instead of relying on a dependency injection framework.

Registry
Registry is the container for all plugin implementations. The framework code will find all plugins inside the registry and can make decisions about what plugins to execute. This is similar to binding plugin implementations in Java using Guice.

Decorators
Python has syntactic sugar that allows calling methods on class and method definitions. We can utilize this feature in addition to abstract classes to trigger registration of plugins automatically.

Abstract Plugin Class
Python does not have direct support for interfaces like Java. But it does have abstract classes which behave similar to interfaces. We will use abstract classes to define the plugin contract.

```
class Registry:
  # Framework code can find all plugin implementations
  # inside this container.
  REGISTRY = []

  def __call__(self, plugin):
    Registry.REGISTRY.append(plugin)
```

Real Implementations
Real implementations work the same as in Java and provide actual functionality of each plugin. However, we decorate the class implementations with the @Registry tag. This will automatically trigger the code that adds plugins to the registry.

Framework Code
Framework code can directly access the Registry to get all registered plugins. It can then make decisions about which plugins to actually execute based on runtime inputs.

```
class Operator(abc.ABC):
  """ Abstract class for plugins."""
  @abc.abstractmethod
  def operate(left, right):
    pass

@Registry()
class Add(Operator):
  def operate(left, right):
    return a + b

@Registry()
class Multiply(Operator):
  def operate(left, right):
    return a * b
```

Conclusion
The plugin architecture can add a lot of value to any software application by making it more extensible, testable and easy to manage. We discussed how the plugin architecture can be implemented using dependency injection frameworks in Java. We also showed by we can use dependency injection fundamentals to build support for Python using only language features.

References
1. Maxwell Mapako (2021) "Building a plugin architecture with Python" https://mwax911.medium.com/building-a-plugin-architecture-with-python-7b4ab39ad4fc
2. Roman Mogylatov (2021) "Dependency injection and inversion of control in Python" https://python-dependency-injector.ets-labs.org/introduction/di_in_python.html
3. Charles White (2020) "Plugin Architecture in Python" https://dev.to/charlesw001/plugin-architecture-in-python-jla
4. Glen McCluskey (2019) "Using Java Reflection" https://www.oracle.com/technical-resources/articles/java/javareflection.html
5. Guice Multibindings (2021) https://github.com/google/guice/wiki/Multibindings
6. Abstract Base Classes (Dec 2021) https://docs.python.org/3/library/abc.html
7. Kevin D Smith, Jim J Jewett, Skip Montanaro, Anthony Baxter (2003) "PEP 318 – Decorators for Functions and Methods" https://peps.python.org/pep-0318