A framework for component software in Python

Lazar Nikolić, Igor Dejanović, Bojana Zoranović, Gordana Milosavljević
Faculty of Technical Sciences, University in Novi Sad, Serbia
lazar.nikolic@uns.ac.rs, bojana.zoranovic@uns.ac.rs, igord@uns.ac.rs, grist@uns.ac.rs

***Abstract*— Component-based software development (CBSD) offers techniques for extending base functionalities of a software solution, usually through plug-ins. This approach positively affects software's flexibility and potentially its longevity through community contribution. Despite the merits of CBSD and its widespread use, there is no standard way of doing it. In this paper we present a framework for component software in Python, which strives to cover fundamental concepts of CBSD while maintaining the simplicity advocated by the Python community.**

1. Introduction

Extensible software is widespread in Python, which is evident from the fact that some of the most popular projects offer extensibility through plug-ins. This is made possible by Python's ecosystem rich support for extensibility, but there is no standard way of doing it. Large projects such as Django, Sphinx and Flask, [11][12][13] all offer extensibility. What is common among these projects though, is that their plug-in mechanisms are built and developed exclusively to support its parent project [10]. For this reason Python community is denied of a more general, yet pythonic [6], solution.

On the other side there are component software frameworks such as iPOPO and Cohorte [14][15], that are heavily influenced by Java and OSGi [17]. They promise production ready solutions and distributed plug-in support, but setting up new projects can be cumbersome, especially for smaller projects. Their design is closer to Java than to Python; component declaration is mainly done by decorators, that are highly similar to Java annotations in both syntax and semantics. This approach poorly handles separation of concerns, since component declaration is done in the same file as its implementation.

Our goal is to build a general purpose, pythonic solution for building component software in Python named *puzzle_box*. The framework must cover all core concepts of CBSD, while still being simple and intuitive. Python's ecosystem has good support for CBSD; Popular Python libraries, *Setuptools*, *virtualenv*, *pkg_resources* and *pip* [17][18][19], provide the necessary toolset to achieve this goal.

2. Related work

2.1. iPOPO

iPOPO iPOPO is a Python-based Service-Oriented Component Model (SOCM) based on Pelix platform. In iPOPO, components are declared as classes with certain decorators and are contained in bundles. These decorators include:

- *@ComponentFactory*: Mark this class as a component factory. This factory is used to instantiate objects of decorated class during runtime.
- *@Property*: Contextual properties which can serve as keywords for component discovery.
- *@Provides*: Mark this class as a service provider for the passed service.
- *@Requires*: Name of the service required by this component. The service will be injected in the class field with the passed name.
- *@Instantiate*: Instantiate the component as soon as the bundle is active.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__author__ = "Lazar Nikolić <lazar.nikolic AT uns DOT acr DOT rs>"
__version__ = "0.1.0"

from setuptools import setup, find_packages

NAME = 'pretty_visualiser'
VERSION = __version__
DESC = 'Displays data in a pretty way'
AUTHOR = __author__
AUTHOR_EMAIL = '<lazar.nikolic AT uns DOT acr DOT rs>'

setup(
    name=NAME,
    version=VERSION,
    description=DESC,
    author=AUTHOR,
    author_email=AUTHOR_EMAIL,
    maintainer=AUTHOR,
    maintainer_email=AUTHOR_EMAIL,
    py_modules=['pretty_visualiser'],
    entry_points={
        'visualiser': [
            'node_painter=pretty_visualiser:paint_node',
        ],
        'puzzle_box_events': [
            'on_start=pretty_visualiser:on_start',
            'on_install=pretty_visualiser:on_install'
        ]
    }
)
```

Figure 1. Example setup.py used in *puzzle_box* framework

Additionally, iPOPO offers method decorators, such as *@Validate* and *@Invalidate*, which lets the platform call the decorated method when the component is valid or invalid. Running an iPOPO application is done by starting the Pelix platform and then installing and starting all the required bundles.

2.2. Cohorte

COHORTE is a Python SOCM that relies on iPOPO for component declaration, meaning that the component declaration is identical. COHORTE first requires the user to download its runtime platform and add it to $COHORTE_HOME environment variable. Minimal configuration requires creating autorun_conf.js file, which contains JSON object with name of the application and name, factory and language of each component. Application is started by running the generated run script

by calling ./run –t –c command, and is hosted as a web application listening on port 38000.

3. Software component model

Software component is defined in different ways by many authors [1]. The following definition is given by by Heineman and Council [2]:

"A [component is a] software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."

Szyperski gave the most widespread definition of software components [3]:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties"

Definition gave by Szyperski helps us define what can be a component, and therefore allow us to decide how to define the component model. In the same book, characteristic properties of components are defined as follows:

1. component is a unit of independent deployment
2. component is a unit of third-party composition
3. component offers a contractually defined interface
4. component has no externally visible state
5. component defines explicit context dependencies

The perfect candidate for components are Python distributions built by Setuptools library. They are versioned archive files that contain Python packages, modules and other resource files [3]. Mechanisms that enable Python distributions to gain aforementioned properties will be discussed in this section. In this paper we use the term "package" to refer to Python distribution, not to Python package as a collection of modules.

Python package descriptor is declared in setup.py file in the project root. It is a regular python module that uses Setuptools function calls to declare the distribution's metadata. An example can be seen in Fig. 1.

Setuptools defines many keywords parameters of *setup* function, but we will only cover those relevant for our framework.

- Keywords *name*, *version* and *description* are self-explanatory and together they form package identifier.
- Keyword *entry_point* declares Python callables that are usable outside this package. This is the most crucial keyword for implementing interfaces and will be covered in detail in later sections.
- Keyword *requires* defines list of required package names as strings. Those packages must be available during runtime.
- Other keywords (*author*, *author_email*, *maintainer* etc.) are purely descriptive and do not contribute to any functionality.

3.1. Independent units of composition

In this section we will cover conditions 1-3 from section 2.

Python packages are designed to be distributed and used independently. Most of packages are readily available at PyPi package repository [4] and can be downloaded and installed into a Python environment. Packages can also use services from other packages, in which case *requires* argument of *setup* function must contain name of the required package as a string. In this case Python interpreter will abort execution once it encounters an unresolved dependency, due to its language's dynamic-typing. This directly conflicts with the principle that component assembly should fail if all criteria are not met [9]. Inherent problem with dependency checking can be resolved by writing custom handlers for package (component) installation. Package dependencies on this level are regular module imports, which only partially satisfy condition 5 from section 2.

3.2. Component definition and interface

In this section we will cover conditions 4 and 5. Each component exposes an interface which serves to communicate with the rest of the system. Interface traditionally consists of function pointers or objects that provide a functionality to the client. For this purpose we used *entry point* from Setuptools API, which are used to expose any Python object (object, function, class or variable) under a specified name. Entry points can also be

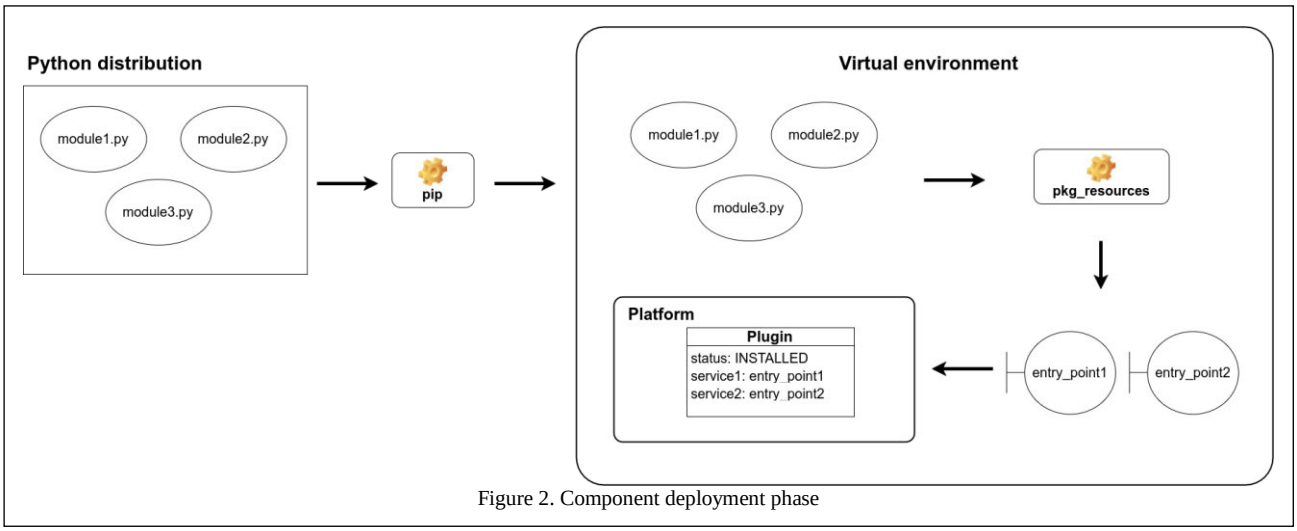

Figure 2. Component deployment phase

aggregated into groups to define component types. Single component can declare any number of entry points, even those in different groups. An example is shown in Fig. 1. The component in question declares two entry point groups: "visualiser" and "puzzle_box_events". The first contains "node_painter" endpoint which exposes *on_start* function from *pretty_visualiser* module, while the latter contains "on_start" and "on_install" endpoints in a similar manner.

4. Component platform

Component platform serves as component run time context and should provide the following:

1. component assembly and life cycle management
2. service registration and discovery
3. model of communication and infrastructure.

Component platform is represented by *Platform* class of our framework. It acts as a facade that uses virtualenv, pip, pkg_resources combined with custom code to create run time context for components. Starting the platform is done by instantiating *Platform* class and each instance represents a different context and will be isolated from others. Platform holds references to components so that they can register, discover services and pass messages between themselves through it. Reference to the platform is injected as a parameter into registered service callables. Platform relies on virtual environments created by *virtualenv* to ensure that only component packages and their dependencies are available during run time, and thus require a virtual environment to be activated before starting.

1.1. Component assembly and life cycle management

Component life cycle is split into two main phases [7][8]:

1. *Design phase* – In this phase, components are constructed and designed. They are stateless and cannot be executed, but can be stored in a repository to be retreived for later use.
2. *Deployment phase* - In this phase, components are created and initialized with a state and thus ready for execution. Before their instantiation, components must be retrieved from a repository.

Components in design phase are represented by Python packages. They carry component's design in the source code and component descriptor in the *setup.py* file. Packages can be stored in PiPy package repository or file system, both of which can serve as the component repository. *Pip* package manager is used for the purpose of retrieving packages (components) from a repository. By installing a package, component's source code is loaded into virtual environment so it can be deployed by the platform. Components in deployment phase are represented by *Plugin* class, which encapsulates component's metadata (name, type, version etc.), state and services. Services are meant to be used for direct communication, by invoking them as functions in the traditional sense. To make sure each component can only be installed once, *Plugin* objects are singletons. During run time, components go through states similar to OSGi bundle life cycle states:

1. *INSTALLED* – Component's package is installed and is available in virtual environment.
2. *STARTED* – Component is operational and its services are available.
3. *STARTED* – Component is operational and its services are available.
4. *STOPPED* – Component's package is available in virtual environment, but is not operational and its services are unavailable.
5. *UNINSTALLED* – Component's package is removed from the virtual environment.

State change handlers can be exposed in "puzzle_box_events" entry point group. Each entry point of this group corresponds to a single state: *on_init, on_start, on_stop and on_uninstall.* Component will then emit an event that notifies other components about the state change. Handlers for state change event coming from other components are registered to platform during runtime, by calling *register_event_handler* function.

Component deployment phases are shown in Fig. 2:

1. Python package (distribution) is retrieved from a repository.
2. Package's modules are loaded into virtual environment with pip install.
3. pkg_resources loads entry points for package modules
4. New Plugin object is created with status *INSTALLED* and added to Platform's context. Previously loaded entry points are registered as its services.

4.1. Service registration and discovery

Component registration is done once a package is loaded into virtual environment. Packages are installed in two ways:

1. By calling *install* command of *pip* package manager. Required package can be located in file system or in available in the PiPy package repository.
2. By instantiating the platform with *folder_name* argument. Platform will then periodically poll the file system for changes and automatically install or uninstall packages as they are added or removed, respectively. Packages are installed by internally calling *pip install* and dynamically loading their modules.

Upon registration, components are given *INSTALLED* status after their *on_init* functions are called, and are ready to be started.

Components can be discovered by calling the platform's *find_plugin* method that takes two optional parameters: component name and component type. If a component name is given, *find_plugin* returns reference to the component with the given name to the client, or a list of components of the required type if a component type is given.

4.2. Model of communication and infrastructure

Platform supports two communication models: direct and indirect. Direct communication involves one component retrieving reference to another component and directly calling its services. In this case the client component can communicate with exactly one component at a time (one-to-one communication). Indirect communication involves one component emitting an event that carries a payload, by calling the platform's *broadcast* function. Custom events can also be created alongside framework's built in events. Other components can

subscribe to an event type by passing a function reference to *register_event_handler* function of the platform.

5. Case Study

In this section we present how *puzzle_box* framework can be used to build a simple pluggable application. The application consists of extensible core and two plugin types: source, used to load data, and visualizer, used to represent the loaded data. Both component types provide their services through an interface. Component diagram of this application is shown in Fig. 3.

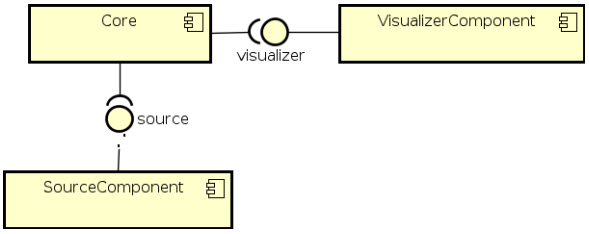

Figure 3. Component diagram of a pluggable Python application with visualizer and source plugins

For the purpose of presentation we created two example visualizers: *pretty_visualizer* and *cool_visualizer*, and two sources: *db_source* and *http_source*. Plugin are stored in "plugin" folder as a *Python wheel* files. Component descriptor for *pretty_visualizer* is shown in Fig. 1. To keep the case study simple and clear, sources are built to return data loaded from a json file, while visualizers only print data to console in different formats. The following use case will be discussed:

1. Core instantiates and starts the platform. Starting the platform involves instantiating *Platform* class with arguments in the following order: 1) list of supported component types, 2) plug-in folder location (Fig. 4.a). If the second argument is omitted, working directory will be used instead.
2. Each plug-in from "plugin" folder is installed and their *on_init* function is called, if registered in "puzzle_box_events" entry point group. In this case, *on_install* function will be called from *pretty_visualizer*, that as a result registers event handler for "load_source" event (Fig. 5.).
3. Core gets references to each plug-in of type "source" then calls their *load_source* service, causing them to load data from a json file and emit a "load_source" event (Fig. 4.b).
4. Event handler *paint_node* is called and the event's payload is printed to the console (Fig. 5.).

Notice that platform reference is injected into every callable that registered with entry points. Besides core, which uses *Platform* class, none of the plug-ins need to import any dependencies from the framework. Component interfaces are declared in *setup.py* file and its implementation is done in the source code. Thanks to this approach, components are developed and distributed as regular Python packages.

6. Conclusion

In this paper we presented *puzzle_box*, a framework for building component software in Python. The goal of this framework is to offer an approach for CBSD in Python

```
if __name__ == "__main__":
    platform = Platform(["source", "visualiser"], "plugins")
    plugins = platform.get_plugin(group="source")
    for plugin in plugins:
        plugin.services.load_source()
```

```
def load_source(config):
    print("db_source: Loading source from database")
    with open('db_source.json') as data_file:.
        data = json.load(data_file)
        platform.broadcast("load_source", "db_source", data)


def on_start(platform):
    print("db_source: Starting db_source")


def on_uninstall(platform):
    print("db_source: Uninstalling db_source")
```

Figure 4. a) Code for core (top) and b) *db_source* (bottom)

```
def paint_node(node, platform):
    col_width = 12
    print("pretty_visualiser:")
    print("| {0} | {1} | {2} |".format("First name".ljust(col_width),
          "Last name".ljust(col_width), "Email".ljust(col_width)))
    print("-----------------------------------")
    print("| {0} | {1} | {2} |".format(node.first_name.ljust(col_width),
          node.last_name.ljust(col_width), node.email.ljust(col_width)))


def on_start(platform):
    print("pretty_visualiser: Starting pretty visualiser")


def on_install(platform):
    print("pretty_visualiser: Installing pretty visualiser")
    platform.register_event_handler("load_source", on_load_source)


def on_load_source(event, platform):
    paint_node(event.data, platform)
```

Figure 5. Code for *pretty_visualizer*

that covers all core aspects of component software, while striving for simplicity and intuitiveness advocated by the Python community.

Available component-based frameworks are iPOPO and Cohorte, with latter being built on former. Their design is closer to Java than to Python; component declaration is mainly done by decorators, that are highly similar to Java annotations in both syntax and semantics. Cohorte promises production-ready distributed system, but its configuration can be cumbersome, especially for small projects. Extensibility of Python applications is also possible by using non-standard its popular libraries, such as Setuptools. Large projects (Django, Flask, Sphinx etc.) use this approach to build a specialized solution that supports only their parent project, so there is no general purpose solution that is also pythonic.

Resulting framework allows development of components that are used and developed as regular Python distributions. Thanks to Python's dynamic typing, only core component needs to import dependencies from the framework itself. Main benefit is that the source code is written and looks like that of a regular, non-extensible Python application, increasing readability and potentially maintainability and productiveness. Comparison of existing frameworks is shown in Table 1.

Future work includes:

1. Using *pip* for run time assembly and platform control.
2. Listening to file system for changes to install new components or update existing ones during run time.
3. Creating a special virtual environment separate from the user's, that is available only to platform.

	configuration	execution	component declaration	component distribution	component discovery
iPOPO	Pelix platform with installed bundles	Run Pelix platform and install all the bundles	decorated class	source code	install with Pelix shell
COHORTE	app_config.js with JSON	Run generated *run.sh* script	decorated class	source code	from config file
puzzle_box	setup.py	Instantiate *Platform* class and run as Python application	any callable exposed with entry point	source code, wheel, egg	pip install, add packages to plugin folder

Table 1. Comparison of CBSD frameworks

4. Contract pre and post conditions support and contract breach detection.
5. Service level dependencies, in which a component can ask for a specific service, not component type, to be available.
6. Interface implementation enforcement option, in which platform requires certain interfaces to be implemented in a specific way.

REFERENCES

[1] M. Broy et al. What characterizes a software component, Software – Concepts and Tools, 19(1):49–56, 1998. G. Heineman and W. Councill, editors.

[2] Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley, 2001.

[3] C. Szyperski, D. Gruntz, and S. Murer. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, second edition, 2002.

[4] https://packaging.python.org/glossary/#term-distribution-package, accessed in April 2017.

[5] https://pypi.python.org/pypi, accessed in April 2017

[6] https://www.python.org/dev/peps/pep-0020/, accessed in April 2017.

[7] B. Christiansson, L. Jakobsson, and I. Crnkovic. CBD process. In I. Crnkovic and M. Larsson, editors, Building Reliable Component-Based Software Systems, pages 89–113. Artech House, 2002.

[8] Kung-Kiu Lau and Zheng Wang, A Taxonomy of Software Component Models, Proceeding EUROMICRO '05 Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications, pages 88-95, 2005.

[9] Kung-Kiu Lau, Vladyslav Uki, Defining and Checking Deployment Contracts for Software Components, CBSE 2006 proceedings, 2006.

[10] D. Hellmann, Dynamic Code Patterns: Extending Your Applications with Plugins, Pycon 2013.

[11] https://www.djangoproject.com/, accessed in April 2017.

[12] http://flask.pocoo.org/, accessed in April 2017.

[13] http://www.sphinx-doc.org/en/stable/, accessed in April 2017.

[14] https://ipopo.readthedocs.io/en/latest/, accessed in April 2017.

[15] https://cohorte.github.io/, accessed in April 2017.

[16] https://www.osgi.org/, accessed in April 2017.

[17] https://setuptools.readthedocs.io/en/latest/, accessed in April 2017.

[18] https://pip.pypa.io/en/stable/, accessed in April 2017.

[19] http://setuptools.readthedocs.io/en/latest/pkg_resources.html, accessed in April 2017.