software-patterns

Release 2.0.0

Konstantinos Lampridis

Nov 19, 2022

CONTENTS:

1 Introduction 3

2 Software Patterns 5

3 Development 11

4 Finalle 17

5 Software Patterns Full API reference 19

6 Indices and tables 25

Python Module Index 27

Index 29

These are the documentation pages of the Software Patterns libre software.

Software Patterns' open source code is hosted on the boromir674/software-patterns repository on Github and its respective software-patterns Python package (both source and wheel distributions) on Pypi.

It also features a public **`CI workflow`_** hosted on CircleCI.

What are Software Design Patterns?

Software Engineers are employing various designs and solutions to solve their problems. The emerging (software) patterns, among the code solutions, targeting reoccuring problems have been studied and formalized in terms of how they are used, what problem they solve and why they are a fit candidate to solve it. These code designs, which are frequently found in various code bases, are known as Software Design Patterns.

Example code to use the *factory* pattern in the form of a *(sub) class registry*:

```
from software_patterns import SubclassRegistry

class MyClassRegistry(metaclass=SubclassRegistry):
    pass

@MyClassRegistry.register_as_subclass('a')
class ClassA:
    def __init__(self, number):
        self.attr = number

@MyClassRegistry.register_as_subclass('b')
class ClassB:
    def __init__(self, number):
```

(continues on next page)

(continued from previous page)

```
        self.attr = number - 1

assert MyClassRegistry.subclasses == {'a': ClassA, 'b': ClassB}

instance_a = MyClassRegistry.create('a', 10)
assert type(instance_a) == ClassA
assert instance_a.attr == 10

assert isinstance(instance_a, ClassA)

instance_b = MyClassRegistry.create('b', 10)
assert type(instance_b) == ClassB
assert instance_b.attr == 9

assert isinstance(instance_b, ClassB)
```

CHAPTER

ONE

INTRODUCTION

In these documentation pages we present the *software_patterns* Python package and the motivation for publishing it. The package currently host only a handful of software patterns, but they come with a test suite and full CI for local and remote builds.

1.1 Why would this project be useful to you?

1.1.1 To develop your code, while abstracting common software patterns away and focus on important business logic.

- To promote the DRY prinicipal in your codebase.
- To promote the principal of Single Responsibility in your codebase.
- To promote having Simple Units of Code in your codebase.
- To keep technical debt low in your codebase.

1.1.2 To Learn:

- You may study the source code to find out more about software design patterns.
- You may wanna learn how to make an open source code contribution and find this codebase a good place to start.

1.2 Why use this Python Patterns library?

It is user-friendly with a clean API designed to reduce boilerplate code needed to use a pattern in your client code.
It comes with a 100% code coverage test suite (automated tests).
It is statically type checked with mypy.
It features (these) documentation pages with examples on how to use the package, doctests (see [here]) and complete API reference (automated) documentation.
It is developer-friendly featuring a public CI-workflow with jobs automating running tests, deploying the python package to a staging server, hosting the test coverage results and visualizing the code dependencies and uml diagrams.

CHAPTER

TWO

SOFTWARE PATTERNS

In this section we dive into the package architecture and the various implementations of Software Design Patterns.

2.1 Architecture Overview

Each implemented pattern corresponds to a python module residing in the *software_patterns* package.

In this section you can see visualizations of the code base capturing the architecture in the form of *dependency graphs* and class *uml diagrams*.

2.1.1 Dependency Graphs

The Dependency Graphs are built by analysing the code and tracking *import* statements.

Inspecting the graph, (corresponding to the code), one can see that *software_patterns* is a "pure" python package. In other words, the code solely depends on modules belonging to the standard python distribution, namely *abc*, *types* and *typing* and thus does not require the installation of any external package.

Note: The .svg file showing the dependency graphs has been generated with the *pydeps* tool. Note: We have a dedicated tox environment (see tox.ini) for generating svg (or png) files depicting the dependency graphs.

2.1.2 Uml Diagrams

The Unified Modeling Language is used to depict the important interfaces of the *software_patterns* package's classes.

Note: The .svg file showing the uml class diagrams has been generated with the legacy *pyreverse* tool which is comes bundled in the *pylint*. Note: We have a dedicated tox environment (see tox.ini) for generating svg (or png) files depicting the uml class diagrams.

2.2 Patterns Implementations

We provide Python implementations of various Software Design Patterns.

2.2.1 Memoize

Implementation of the Memoize Software Design Pattern.

Memoize is implemented using an Object Pool which is queried by a key which is the result of computing a hash given runtime arguments.

class software_patterns.memoize.**ObjectsPool**(*callback: Callable[[...], ObjectType], hash_callback: Optional[Callable[[...], Union[int, str]]] = None*)

Cache objects and allow to query (the pool) using runtime arguments.

Instances of the ObjectsPool class implement the Object Pool Software Design Creational Pattern.

Whenever an object is requested, it is checked whether it exists in the pool by using the runtimetime arguments to query a python dictionary. If it exists, a reference is returned, otherwise a new object is constructed (given the provided callback) and its reference is returned.

Example

```
>>> from software_patterns import ObjectsPool
>>> class ClientClass:
...  def __init__(self, a: int, b: int):
...   pass
```

```
>>> object_pool = ObjectsPool[ClientClass](ClientClass)
```

```
>>> obj1 = object_pool.get_object(1, 2)
>>> obj2 = object_pool.get_object(1, 3)
>>> obj3 = object_pool.get_object(1, 2)
```

```
>>> id(obj1) == id(obj3)
True
```

```
>>> len(object_pool._objects)
2
```

Parameters

- **callback** (*Callable[..., ObjectType]*) – constructs objects given arguments
- **hash_callback** (*Optional[RuntimeBuildHashCallable], optional*) – option to overide the default hash key computer. Defaults to None.

Returns
[description]

Return type
[type]

get_object(**args: Any, **kwargs: Any*) → ObjectType

Request an object from the pool.

Get or create an object given the input arguments, which are used to create a unique hash key. The key is used to query a python dictionary and determine whether the object request refers to a cached object.

Returns
the reference to the object that corresponds to the input arguments, regardless of whether it was found in the pool or not

Return type
object (ObjectType)

2.2.2 Listener (aka Observer)

class software_patterns.notification.**Subject**(**args, **kwargs*)

Concrete Subject which owns an important state and notifies observers.

The subject can be used to build the data encapsulating the event being broadcasted.

Both the _state and _observers attributes have a simple implementation, but can be overrode to accommodate for more complex scenarios.

The observers/subscribers are implemented as a python list. In more complex scenarios, the list of subscribers can be stored more comprehensively (categorized by event type, etc.).

The subscription management methods provided are 'attach', 'detach' (as in the SubjectInterface) and 'add', which attached multiple observers at once.

Example

```
>>> from software_patterns import Subject, Observer
```

```
>>> broadcaster = Subject()
```

```
>>> class ObserverTypeA(Observer):
...  def update(self, *args, **kwargs):
...   event = args[0].state
...   print(f'observer-type-a reacts to event {event}')
```

```
>>> class ObserverTypeB(Observer):
...  def update(self, *args, **kwargs):
...   event = args[0].state
...   print(f'observer-type-b reacts to event {event}')
```

```
>>> subscriber_1 = ObserverTypeA()
>>> subscriber_2 = ObserverTypeB()
```

```
>>> broadcaster.add(subscriber_2, subscriber_1)
```

```
>>> broadcaster.state = 'event-object-A'
```

```
>>> broadcaster.notify()
observer-type-b reacts to event event-object-A
observer-type-a reacts to event event-object-A
```

```
>>> broadcaster.detach(subscriber_2)
```

```
>>> broadcaster.state = 'event-object-B'
>>> broadcaster.notify()
observer-type-a reacts to event event-object-B
```

add(**observers*)

Subscribe multiple observers at once.

attach(*observer: ObserverInterface*) → None

Attach an observer to the subject; subscribe the observer.

detach(*observer: ObserverInterface*) → None

Detach an observer from the subject; unsubscribe the observer.

notify() → None

Notify all observers about an event.

property state: Optional[StateType]

Get the state of the Subject.

Returns
the object representing the current state of the Subject

Return type
StateType

class software_patterns.notification.**Observer**

abstract update(**args*, ***kwargs*) → None

Receive an update (from a subject); handle an event notification.

2.2.3 Proxy

Proxy structural software pattern.

This module contains boiler-plate code to supply the Proxy structural software design pattern, to the client code.

class software_patterns.proxy.**Proxy**(*runtime_proxy: T*)

The Proxy has an interface identical to the ProxySubject.

Example

```
>>> from software_patterns import Proxy
>>> class PlusTen(Proxy):
...  def __call__(self, x: int):
...   result = self._proxy_subject(x + 10)
...   return result
```

```
>>> def plus_one(x: int):
...  return x + 1
>>> plus_one(2)
3
```

```
>>> proxy = PlusTen(plus_one)
>>> proxy(2)
13
```

2.2.4 Subclass Registry

class software_patterns.subclass_registry.**SubclassRegistry**(**args*)

Subclass Registry

A (parent) class using this class as metaclass gains the 'subclasses' class attribute as well as the 'create' and 'register_as_subclass' class methods.

The 'subclasses' attribute is a python dictionary having string identifiers as keys and subclasses of the (parent) class as values.

The 'register_as_subclass' class method can be used as a decorator to indicate that a (child) class should belong in the parent's class registry. An input string argument will be used as the unique key to register the subclass.

The 'create' class method can be invoked with a (string) key and suitable constructor arguments to later construct instances of the corresponding child class.

Example

```
>>> from software_patterns import SubclassRegistry
```

```
>>> class ClassRegistry(metaclass=SubclassRegistry):
...  pass
```

```
>>> ClassRegistry.subclasses
{}
```

```
>>> @ClassRegistry.register_as_subclass('child')
... class ChildClass:
...  def __init__(self, child_attribute):
...   self.attr = child_attribute
```

```
>>> child_instance = ClassRegistry.create('child', 'attribute-value')
>>> child_instance.attr
'attribute-value'
```

```
>>> type(child_instance).__name__
'ChildClass'
```

```
>>> isinstance(child_instance, ChildClass)
True
```

```
>>> {k: v.__name__ for k, v in ClassRegistry.subclasses.items()}
{'child': 'ChildClass'}
```

create(*subclass_identifier*, **args*, ***kwargs*) → T

Create an instance of a registered subclass, given its unique identifier and runtime (constructor) arguments.

Invokes the identified subclass constructor passing any supplied arguments. The user needs to know the arguments to supply depending on the resulting constructor signature.

Parameters
subclass_identifier (*str*) – the unique identifier under which to look for the corresponding subclass

Raises

- *UnknownClassError* – In case the given identifier is unknown to the parent class
- *InstantiationError* – In case the runtime args and kwargs do not match the constructor signature

Returns
the instance of the registered subclass

Return type
object

register_as_subclass(*subclass_identifier*)

Register a class as subclass of the parent class.

Adds the subclass' constructor in the registry (dict) under the given (str) identifier. Overrides the registry in case of "identifier collision". Can be used as a python decorator.

Parameters
subclass_identifier (*str*) – the user-defined identifier, under which to register the subclass

class software_patterns.subclass_registry.**InstantiationError**

CHAPTER

THREE

DEVELOPMENT

In this section, we discuss several matters related to how development is done through out the project.

3.1 Version Control

For versioning our code we use git and github.com for hosting the open-source code in the web (online).

We use two long-living branches, *master* and *dev*.

3.1.1 Master branch

We use the *master* branch to tag "releases" published in github as well as in pypi as python packages.

One should (only) branch of off master when there is the need to 'develop' a hotfix. A hotfix is a topical branch that directly remedies a problem found in production code. One example of such use case is when a bug is discovered that affects production code and the need for fix rises immediately.

3.1.2 Dev Branch

We use the 'dev' to facilitate development operations on the repository.

One should branch of off *dev* when developing for example a new feature, a ci improvement, or a documentation addition/re-write.

Having a dedicated dev branch helps collaborators to rapidly acquire critical patches of code despite working on separate branches. This is achieved with a sequence of merges, but more on this later.

3.2 Developer setup

In order to do development on your local machine you should first get the code. You should "clone" the code from github making sure you receive the dev branch from the remote server.

Example command to clone code using ssh and checkout dev branch

> git clone git@github.com:boromir674/software-patterns.git –branch dev cd software-patterns

3.2.1 How to prepare your dev environment?

Normally, a dev environment needs to facilitate some kind of write-code -> test-code loop. In terms of python, that involves creating a 'virtual environment', installing the package using the altered local source code, installing tooling (eg pytest) and running for example the test suite (unit tests).

However, in our development workflow there more activities involved than just running our test suite in an (isolated) virtual environemnt.

A developer may also need to engage with activities such as build the documentation pages (eg into html), generating visualizations of the codebase (ie generate *.svg files), running a static code analyzer tool, building the code into a wheel distribution, doing a code/package deployment, etc.

Most of the above activities/processes require some kind of python tool to be installed in an separate isolated virtual environment and subsequently invoked from a cli, with specific runtime arguments.

In order to automate running such python processes we use *tox* which manages python environments using a declarating config file.

This way we roughly translate all the development activities into simple cli commands!

In the following sub sections you will find an illustration on how to run all the development processes in an automated way and how to run the 'local testing' process in an alternative manual way.

Automated processes

To automate the various development processes we use the tox automation tool. This allows us to map each process into a single cli command, as well as use tox invokations as common front-end code between local and remote CI.

To use tox please install it with something like *pip install –user tox* (its up to you to do a "global", "user" or in-a-virtual-environment' installation of tox).

```
tox -av
```

Use tox -av for a complete enumeration and short description of all the tox environments that are available to the developer out of the box. All these environemnts, dependencies, commands, environment variables, etc can be found in file tox.ini.

tox –help is always available for an explanation of the cli parameters and flags.

** How to run local CI?

To run the default sequence of tox envs, (*mypy*, *clean* and *dev*) run:

tox

To run the test suite on the currently developed code (local git checkout) and measure code coverage run:

tox -e dev

To run mypy against the source code and do static type checking (similar to a compiled language), run:

tox -e mypy

To analyse the code and output information about errors, potential problems, convention violations and complexity you can run the prospector tool:

tox -e prospector

To generate local svg files depicting the dependency graphs, run:

tox -e graphs

Manually running tests

In this section, we present an alternative way to run the test suite against the developed code. It is roughly equivalent to executing *tox -e dev* from the cli.

We first create a python *virtual environemnt*, install the package in 'dev' mode (aka edit mode), install a test-runner and then we are ready to do the write-code -> test-code loop as many times as needed.

Sample environment setup, code and tooling installation commands:

> # Create a virtual environment in 'env-dev' directory pip install –user virtualenv && virtualenv env-dev –python=python3
>
> # Activate the virtual environment source env-dev/bin/Activate
>
> # Install the code in 'dev mode' (aka edit mode) pip install -e .
>
> # Install test-runner pip install pytest

The above commands should have set you up for starting to writing code.

Now you can loop the below steps as needed.

1. Change the code
2. Run test suite using test-runner with command *pytest tests -vv*

3.3 Contributing

*** How to branch?

Pick a short but descriptive name for your branch.

If developing a missing feature we recommend starting with a verb and using imperative.

Examples:

- for a feature to implement the Mediator pattern one might name their feature branch

as 'implement-mediator-pattern' - for a feature to add a (class) factory method on the Observer class to facilitate creating simple instances of the Observer class from runtime 'update callback' one might name their branch as 'add-factory-method-in-observer'

Branch of off the dev branch.

git branch –track dev origin/dev git checkout dev

For example, create topical branch as follows:

git checkout -b 'add-factory-method-in-observer'

*** How to commit?

Of course if you make changes in the production code, make sure it is tested and fully covered with unit tests. Please strive for making atomic commits, meaning that each commit can act as a stand-alone bulk of changes. Of course each bulk of changes should also be in aggreement with the topic covered in the working branch.

In our opinion, atomic commits come with the following benefits:

- easier cherry-picking (less conflicts)
- easier merges (less conflicts)
- cleaner commits timeline visualizations

It is also a good idea to be consistent on how we format and write each commit's message. To achieve that we use the 'commitizen' (cz) tool, which provides an interactive "commit message building" wizard through the cli.

Specifically, we use the so called cz-conventional-adapter, which defines commit messages semantics and format.

In practice, each time one is about to commit changes, they just needs to invoke *git cz* in place of *git commit* and the wizard shall run on the terminal.

So, please install commitizen and the cz-conventional-adapter in your development environemnt. A good starting point would be the script we provide that automatically installs commitizen in "user space" and takes care of setting up the cz-conventional-adapter too.

You can read more about commitizen and the aforementioned cz-conventional-adapter [URLs].

**** How to do a pull-request? A Pull Reqest can be opened either from a github cli or the gitub web interface.

We open a pull request whenever we are confident and want to signal that our branch has been developed to completion. For any type of Pull Request we should adhere to the following principals: - all commits are more or less atomic

> As discussed we promote the idea of having atomic commits on working branches That does not mean that the developer should refrain from committing as frequently as they want, since one can always do "commit squashing" before opening a pull request.
>
> git rebase dev –interactive
>
> At the end is does not matter how many commits end up in the branch, as long as they are atomic.

3.3.1 Feature Branches

All feature branches should be branched off of the 'dev' branch. All Pull Requests should target the 'dev' branch.

For Feature Branch type of Pull Request we should adhere to the following principals: – all necessary business logic code is finished – all tests (old and new) are passing (both locally and in remote CI server) – all documentation sources have been updated — docstrings to build API ref in html — doctests written and passing — other documentation pages (eg section where we discuss what Software Desgin Patterns are included as modules in our package) — images embedded in docs pages that reflect the code architecture (dependency graphs and uml diagrams)

Apart from the above requirements we should pay attention to the evolution of the dev branch in the meantime.

If the dev branch has progressed from the commit that our branch's base started from, we need to make a decision.

Should our code immediately benefit from the changes incorporated in 'dev'?

If yes, then one has two options: to merge 'dev' into their branch or to rebase their branch on 'dev'

We recommend to merge the 'dev' code into our branch, because that way we clearly singal what are the branch's topical commits and what are the merged changes. Importantly, this is evident on the github web interface too, which is where peer-code-review, is done.

> git merge dev –no-ff

If no, then one can proceed with opening the Pull Request, which theoritically should still not produce any "conflicts", assuming that each dev is commited to staying on-topic on their branch.

If dev branch has not progressed further from the commit where we intially based our branch off, we simply proceed by opening a Pull Request (ie from the github web interface or cli).

3.3.2 Bugfixes (Hotfix Beanches)

All hotfix branches should be branched off of the 'master' branch. All Pull Requests should target the 'master' branch.

For Bugfix (Hotfix) Branch type of Pull Request we should adhere to the following principals:

– all necessary business logic fixes are finished – all tests (old and new) are passing (both locally and in remote CI server) – all documentation sources have been updated

*** How to run remote CI?

WE utilize a series of 3rd party web services to facilitate the various automated "actions" we undertake during various stages of the development.

CI/CD

CHAPTER

FOUR

FINALLE

These were the documentation pages of the *Software Patterns* project, featuring a open source python implementation of various Software Design Patterns, automation, public CI workflow, documentation with sphinx, doctests.

We hope this projects' features proove at least remotely useful to another software engineer who shall get familiar with it (the project).

Happy Libre Software!

CHAPTER

FIVE

SOFTWARE PATTERNS FULL API REFERENCE

5.1 software_patterns package

5.1.1 software_patterns.memoize module

Implementation of the Memoize Software Design Pattern.

Memoize is implemented using an Object Pool which is queried by a key which is the result of computing a hash given runtime arguments.

class software_patterns.memoize.**ObjectsPool**(*callback: Callable[[...], ObjectType], hash_callback: Optional[Callable[[...], Union[int, str]]] = None*)

Bases: Generic[ObjectType]

Cache objects and allow to query (the pool) using runtime arguments.

Instances of the ObjectsPool class implement the Object Pool Software Design Creational Pattern.

Whenever an object is requested, it is checked whether it exists in the pool by using the runtimetime arguments to query a python dictionary. If it exists, a reference is returned, otherwise a new object is constructed (given the provided callback) and its reference is returned.

Example

```
>>> from software_patterns import ObjectsPool
>>> class ClientClass:
...  def __init__(self, a: int, b: int):
...   pass
```

```
>>> object_pool = ObjectsPool[ClientClass](ClientClass)
```

```
>>> obj1 = object_pool.get_object(1, 2)
>>> obj2 = object_pool.get_object(1, 3)
>>> obj3 = object_pool.get_object(1, 2)
```

```
>>> id(obj1) == id(obj3)
True
```

```
>>> len(object_pool._objects)
2
```

Parameters

- **callback** (*Callable[..., ObjectType]*) – constructs objects given arguments
- **hash_callback** (*Optional[RuntimeBuildHashCallable], optional*) – option to overide the default hash key computer. Defaults to None.

Returns
[description]

Return type
[type]

get_object(**args: Any*, ***kwargs: Any*) → ObjectType

Request an object from the pool.

Get or create an object given the input arguments, which are used to create a unique hash key. The key is used to query a python dictionary and determine whether the object request refers to a cached object.

Returns
the reference to the object that corresponds to the input arguments, regardless of whether it was found in the pool or not

Return type
object (ObjectType)

user_supplied_callback: Dict[bool, Callable] = {False: <function ObjectsPool.<lambda>>, True: <function ObjectsPool.<lambda>>}

5.1.2 software_patterns.notification module

Notification-Listener (aka subject-observer) software design pattern.

Simple implementation of the subject/observers (broadcast/listeners) pattern, exposed as python classes.

This is a Behavioural Pattern which can be used when you want one or more components to be notified and react accordingly, when 'something happens'.

One entity, known as Subject, is responsible to send out a notification and each entity "subscribed" to the Subject receives it and reacts. Each subscribed entity is known as a Listener or Observer.

The idea is that the Subject is agnostic of its Observers implementation and the client code can "attach" or "detach" (subscribe/unsubscribe) as many of them at runtime.

This module provides the Observer class, serving as the interface that needs to be implemented by concrete classes; the update method needs to be overrode. Concrete Observers react to the notifications/updates issued by the Subject they had been attached/subscribed to.

This module also provides the concrete Subject class, serving with methods to subscribe/unsubscribe (attach/detach) observers and also with a method to "notify" all Observers.

class software_patterns.notification.**Observer**

Bases: ObserverInterface, ABC

class software_patterns.notification.**Subject**(**args*, ***kwargs*)

Bases: SubjectInterface, Generic[StateType]

Concrete Subject which owns an important state and notifies observers.

The subject can be used to build the data encapsulating the event being broadcasted.

Both the _state and _observers attributes have a simple implementation, but can be overrode to accommodate for more complex scenarios.

The observers/subscribers are implemented as a python list. In more complex scenarios, the list of subscribers can be stored more comprehensively (categorized by event type, etc.).

The subscription management methods provided are 'attach', 'detach' (as in the SubjectInterface) and 'add', which attached multiple observers at once.

Example

```
>>> from software_patterns import Subject, Observer
```

```
>>> broadcaster = Subject()
```

```
>>> class ObserverTypeA(Observer):
...  def update(self, *args, **kwargs):
...   event = args[0].state
...   print(f'observer-type-a reacts to event {event}')
```

```
>>> class ObserverTypeB(Observer):
...  def update(self, *args, **kwargs):
...   event = args[0].state
...   print(f'observer-type-b reacts to event {event}')
```

```
>>> subscriber_1 = ObserverTypeA()
>>> subscriber_2 = ObserverTypeB()
```

```
>>> broadcaster.add(subscriber_2, subscriber_1)
```

```
>>> broadcaster.state = 'event-object-A'
```

```
>>> broadcaster.notify()
observer-type-b reacts to event event-object-A
observer-type-a reacts to event event-object-A
```

```
>>> broadcaster.detach(subscriber_2)
```

```
>>> broadcaster.state = 'event-object-B'
>>> broadcaster.notify()
observer-type-a reacts to event event-object-B
```

add(**observers*)

Subscribe multiple observers at once.

attach(*observer: ObserverInterface*) → None

Attach an observer to the subject; subscribe the observer.

detach(*observer: ObserverInterface*) → None

Detach an observer from the subject; unsubscribe the observer.

notify() → None

Notify all observers about an event.

property state: Optional[StateType]

Get the state of the Subject.

Returns
the object representing the current state of the Subject

Return type
StateType

5.1.3 software_patterns.proxy module

Proxy structural software pattern.

This module contains boiler-plate code to supply the Proxy structural software design pattern, to the client code.

class software_patterns.proxy.**Proxy**(*runtime_proxy: T*)

Bases: ProxySubjectInterface, Generic[T]

The Proxy has an interface identical to the ProxySubject.

Example

```
>>> from software_patterns import Proxy
>>> class PlusTen(Proxy):
...  def __call__(self, x: int):
...   result = self._proxy_subject(x + 10)
...   return result
```

```
>>> def plus_one(x: int):
...  return x + 1
>>> plus_one(2)
3
```

```
>>> proxy = PlusTen(plus_one)
>>> proxy(2)
13
```

5.1.4 software_patterns.subclass_registry module

Exposes the SubclassRegistry that allows to define a single registration point of one or more subclasses of a (common parent) class.

exception software_patterns.subclass_registry.**InstantiationError**

Bases: Exception

class software_patterns.subclass_registry.**SubclassRegistry**(**args*)

Bases: type, Generic[T]

Subclass Registry

A (parent) class using this class as metaclass gains the 'subclasses' class attribute as well as the 'create' and 'register_as_subclass' class methods.

The 'subclasses' attribute is a python dictionary having string identifiers as keys and subclasses of the (parent) class as values.

The 'register_as_subclass' class method can be used as a decorator to indicate that a (child) class should belong in the parent's class registry. An input string argument will be used as the unique key to register the subclass.

The 'create' class method can be invoked with a (string) key and suitable constructor arguments to later construct instances of the corresponding child class.

Example

```
>>> from software_patterns import SubclassRegistry
```

```
>>> class ClassRegistry(metaclass=SubclassRegistry):
...  pass
```

```
>>> ClassRegistry.subclasses
{}
```

```
>>> @ClassRegistry.register_as_subclass('child')
... class ChildClass:
...  def __init__(self, child_attribute):
...   self.attr = child_attribute
```

```
>>> child_instance = ClassRegistry.create('child', 'attribute-value')
>>> child_instance.attr
'attribute-value'
```

```
>>> type(child_instance).__name__
'ChildClass'
```

```
>>> isinstance(child_instance, ChildClass)
True
```

```
>>> {k: v.__name__ for k, v in ClassRegistry.subclasses.items()}
{'child': 'ChildClass'}
```

create(*subclass_identifier*, **args*, ***kwargs*) → T

Create an instance of a registered subclass, given its unique identifier and runtime (constructor) arguments.

Invokes the identified subclass constructor passing any supplied arguments. The user needs to know the arguments to supply depending on the resulting constructor signature.

Parameters

subclass_identifier (*str*) – the unique identifier under which to look for the corresponding subclass

Raises

- *UnknownClassError* – In case the given identifier is unknown to the parent class

- *InstantiationError* – In case the runtime args and kwargs do not match the constructor signature

Returns
the instance of the registered subclass

Return type
object

register_as_subclass(*subclass_identifier*)

Register a class as subclass of the parent class.

Adds the subclass' constructor in the registry (dict) under the given (str) identifier. Overrides the registry in case of "identifier collision". Can be used as a python decorator.

Parameters
subclass_identifier (*str*) – the user-defined identifier, under which to register the subclass

subclasses: Dict[str, type]

exception software_patterns.subclass_registry.**UnknownClassError**

Bases: Exception

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

software_patterns.memoize, 19
software_patterns.notification, 20
software_patterns.proxy, 22
software_patterns.subclass_registry, 22

INDEX

A

add() (*software_patterns.notification.Subject method*), 8, 21
attach() (*software_patterns.notification.Subject method*), 8, 21

C

create() (*software_patterns.subclass_registry.SubclassRegistry method*), 10, 23

D

detach() (*software_patterns.notification.Subject method*), 8, 21

G

get_object() (*software_patterns.memoize.ObjectsPool method*), 6, 20

I

InstantiationError, 22
InstantiationError (*class in software_patterns.subclass_registry*), 10

M

module
    software_patterns.memoize, 6, 19
    software_patterns.notification, 20
    software_patterns.proxy, 8, 22
    software_patterns.subclass_registry, 22

N

notify() (*software_patterns.notification.Subject method*), 8, 21

O

ObjectsPool (*class in software_patterns.memoize*), 6, 19
Observer (*class in software_patterns.notification*), 8, 20

P

Proxy (*class in software_patterns.proxy*), 8, 22

R

register_as_subclass() (*software_patterns.subclass_registry.SubclassRegistry method*), 10, 24

S

software_patterns.memoize
    module, 6, 19
software_patterns.notification
    module, 20
software_patterns.proxy
    module, 8, 22
software_patterns.subclass_registry
    module, 22
state (*software_patterns.notification.Subject property*), 8, 22
subclasses (*software_patterns.subclass_registry.SubclassRegistry attribute*), 24
SubclassRegistry (*class in software_patterns.subclass_registry*), 9, 22
Subject (*class in software_patterns.notification*), 7, 20

U

UnknownClassError, 24
update() (*software_patterns.notification.Observer method*), 8
user_supplied_callback (*software_patterns.memoize.ObjectsPool attribute*), 20