# Implementation Details

This section delves into the core classes that make up the architecture of our VV system. Each class has a specific role and set of responsibilities, designed to facilitate modular and extensible software development. It describes the purpose of these classes, their primary methods, and example use cases that highlight their functionality. The intent is to provide a comprehensive understanding of how these components interact to support the system's goals, including adaptability and the ability to accommodate future plugin development by our team or other contributors.

## Core Classes and their Responsibilities

In the following subsections, we will examine each core class to detail its role and responsibilities in the system architecture.

### The ‘dataset\_builders’ Module

The dataset\_builders module serves as the core for data acquisition in the Visual Viper Framework. This module offers an abstract class, AbstractDatasetBuilder, designed to be extended for specific data sourcing implementations. Its design promotes low coupling, making it easier to integrate new data sources.

#### The ‘AbstractDatasetBuilder’ Class

The first class in the architecture is AbstractDatasetBuilder, which is an abstract class acting as a blueprint for all dataset builders. The class declares a method build\_dataset(params=None), which subclasses should implement to provide the actual dataset-building functionality (Listing X). This abstract class is crucial in achieving low coupling as it ensures that other components of the system need not know the specific dataset builder that will be used.

class AbstractDatasetBuilder:

@abc.abstractmethod

def build\_dataset(self, params=None):

raise NotImplementedError()

Listing X: Code snippet showing the AbstractDatasetBuilder class, which provides a method interface for building datasets.

#### The ‘Key’ Class

Within the dataset\_builders module, there's a simple but critical class named Key (Listing X). This class serves to encapsulate key-value pairs used for data retrieval. The Key class has an initializer that takes two arguments: key and an optional src parameter. Here, key represents the data attribute, while src can be used to specify the data source.

class Key():

def *\_init\_*(self, key, src=None) -> None:

self.key = key

self.src = src

Listing X: Code snippet showing the Key class used for encapsulating data retrieval attributes.

The utility of the Key class becomes more evident when used in conjunction with the notation\_builders module, where it plays an instrumental role in linking dataset attributes to visual elements in a chart.

#### The GoogleSpreadsheetDatasetBuilder Class

Extending the AbstractDatasetBuilder is the GoogleSpreadsheetDatasetBuilder class (Listing X). This concrete implementation utilizes the Google Sheets API to fetch data. The class uses the gspread library and OAuth 2.0 for secure and efficient data retrieval. One of the significant advantages of this class is its ability to handle multiple named ranges across multiple worksheets.

import gspread

from google.oauth2 import service\_account as sa

from googleapiclient.discovery import build

from .abstract\_dataset\_builder import \*

class GoogleSpreadsheetDatasetBuilder(AbstractDatasetBuilder):

DEFAULT\_SA\_PATH = "./service\_account.json"

DEFAULT\_SCOPES = ['<https://www.googleapis.com/auth/drive>']

def *\_init\_*(self, file\_id=None, sa\_path=None) -> None:

self.file\_id = file\_id

self.sa\_path = sa\_path or self.DEFAULT\_SA\_PATH

self.auth = sa.Credentials.from\_service\_account\_file(

self.sa\_path,

scopes=self.DEFAULT\_SCOPES

)

self.dataset = dict()

def build(self, params=None, ws\_index=0):

gs = gspread.service\_account(self.sa\_path)

range\_sets = dict()

for el in params["ranges"]:

if not isinstance(el, tuple):

el = (el, self.file\_id)

named\_range, file\_id = el

if not file\_id in range\_sets:

range\_sets[file\_id] = []

range\_sets[file\_id].append(named\_range)

for file\_id, ranges in range\_sets.items():

sheet = gs.open\_by\_key(file\_id)

worksheet = sheet.get\_worksheet(ws\_index)

response = worksheet.batch\_get(

ranges,

value\_render\_option="UNFORMATTED\_VALUE",

)

response = {

ranges[i]: response[i][0][0] for i in range(len(response))

}

self.dataset.update(response)

return self.dataset

Listing X: Code snippet showing the GoogleSpreadsheetDatasetBuilder class, responsible for building datasets from Google Sheets.

### The ‘notation\_builders’ Module

The notation\_builders module encapsulates the logic required for constructing the chart notations and solving data dependencies for the actual visualization. Two abstract classes form the core of this module: AbstractChartNotationBuilder and AbstractChartNotation.

#### The ‘AbstractChartNotationBuilder’ Class

AbstractChartNotationBuilder is an abstract class that acts as a blueprint for all chart notation builders (Listing X). It declares methods like build(params=None) that subclasses need to implement to provide the actual chart-building functionality. The class uses an internal property bindings, designed to be overridden in subclasses, that links the dataset keys to visual elements in a chart.

The AbstractChartNotationBuilder class also introduces a collect\_keys() method, which traverses all the bindings and collects the Key instances, serving as a bridge to the dataset\_builders module. This method ensures that all necessary data points can be fetched efficiently from the dataset.

class AbstractChartNotationBuilder:

# ...

def *\_init\_*(self, bindings=None, id=None, opts=None):

# ...

@property

def bindings(self):

raise NotImplementedError()

def collect\_keys(self, dataset):

# ...

@abc.abstractmethod

def build(self, params=None) -> dict:

raise NotImplementedError()

Listing X: Code snippet showing the AbstractChartNotationBuilder class, which serves as the framework for building chart notations.

#### The ‘AbstractChartNotation’ Class

The AbstractChartNotation class functions as a complementary element to the AbstractChartNotationBuilder class. This class registers the dataset and contains a solve() method. The solve() method uses instances of the Key class from the dataset\_builders module to fetch the necessary data points, thereby linking the chart notation to the actual data (Listing X).

class AbstractChartNotation:

def *\_init\_*(self):

self.dataset = {}

def register\_dataset(self, dataset):

# ...

def solve(self, el):

# ...

Listing X: Code snippet showing the AbstractChartNotation class, which registers the dataset and provides a method for solving notation elements.

#### The ‘ForestPlot’ Class

The ForestPlot class (Listing X) is a concrete implementation that inherits from AbstractChartNotationBuilder. It specializes in building Forest Plots, a type of chart that is commonly used to visualize grouped data points in a graphical format. The class provides the option to include labels for different measures (hr, lo, hi) and customizes them as needed.

from .abstract\_notation\_builder import AbstractChartNotationBuilder

from .forest\_plot\_binding\_notation import ForestPlotBinding

class ForestPlot(AbstractChartNotationBuilder):

OPTS = dict(

labels = dict(

hr="HR",

lo="CI Low",

hi="CI High",

)

)

@property

def bindings(self):

return [

ForestPlotBinding(

measure="",

hr=self.opts["labels"]["hr"],

lo=self.opts["labels"]["lo"],

hi=self.opts["labels"]["hi"],

),

\*self.\_bindings

]

def build(self, params=None) -> dict:

base\_schema = {

"$schema": "<https://vega.github.io/schema/vega-lite/v5.json>",

"data": {

"values": [

]

},

#...

}

notation = base\_schema.copy()

values = [binding.solved\_data for binding in self.bindings]

notation["data"]["values"] = values

return notation

Listing X: Code snippet showing the ForestPlot class, responsible for building the notation for Forest Plots.

The ForestPlot class overrides the bindings property, providing a default ForestPlotBinding instance that serves as a blueprint for all bindings related to this specific type of chart. It also defines the build(params=None) method to generate the notation for rendering the chart using the Vega-Lite schema.

#### The ForestPlotBinding Class

This class inherits from AbstractChartNotation and serves to hold and solve the data points necessary for a Forest Plot. Unlike the generic AbstractChartNotation, ForestPlotBinding has additional properties specific to Forest Plots, such as hr (Hazard Ratio), lo (Low Confidence Interval), and hi (High Confidence Interval), as can be seen in Listing X.

The ForestPlotBinding class introduces the data and solved\_data properties. The data property returns the initial (unsolved) key-value pairs, whereas the solved\_data property uses the inherited solve() method to get the actual data points from the dataset. These properties bridge the gap between data sourcing and data representation in the chart.

import json

from .abstract\_chart\_notation import AbstractChartNotation

class ForestPlotBinding(AbstractChartNotation):

def \_\_init\_\_(self, measure, hr, lo, hi) -> None:

super().\_\_init\_\_()

self.measure = measure

self.\_hr = hr

self.\_lo = lo

self.\_hi = hi

@property

def data(self) -> dict:

return dict(

measure=self.measure,

lo=self.\_lo,

hr=self.\_hr,

hi=self.\_hi,

)

@property

def solved\_data(self) -> dict:

return dict(

measure=self.measure,

lo=self.lo,

hr=self.hr,

hi=self.hi,

)

@property

def lo(self):

return self.solve(self.\_lo)

@property

def hr(self):

return self.solve(self.\_hr)

@property

def hi(self):

return self.solve(self.\_hi)

def items(self):

yield ("hr", self.\_hr)

yield ("lo", self.\_lo)

yield ("hi", self.\_hi)

def \_\_repr\_\_(self):

return f"hr:{self.hr}, lo:{self.lo}, hi:{self.hi}"

Listing X: Code snippet showing the ForestPlotBinding class, which encapsulates the logic for holding and solving data points specific to Forest Plots.

#### Summary Diagram for the ‘notation\_builders’ Module

To sum up the relationships between these classes, please refer to the following class diagram depicted in Figure X.

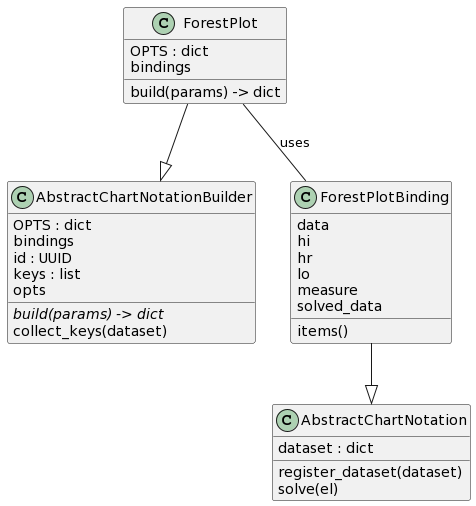


Figure X: Class diagram of the classes included in the ‘notation\_builders’ module.

The ForestPlot class inherits from AbstractChartNotationBuilder, while ForestPlotBinding inherits from AbstractChartNotation. The ForestPlot class uses instances of ForestPlotBinding to build the chart, leveraging the options and methods provided by the parent classes.

Again, this setup ensures low coupling and high cohesion, thus aligning well with the principles of clean architecture.

### The ‘chart\_renderers’ Module

The chart\_renderers module is a pivotal component in the VV Framework responsible for rendering visualizations. The module houses an abstract class, AbstractChartRenderer, which is designed to be extended by specific rendering engines.

#### The ‘AbstractChartRenderer’ Class

The backbone of the chart\_renderers module is the AbstractChartRenderer class (Listing X). It is an abstract class serving as a blueprint for all chart rendering implementations. It declares a method render(notation=None, params=None), which is expected to be implemented by subclasses to provide the actual chart rendering functionality. This design pattern ensures that other system components do not need to be aware of the specific renderer in use, thereby achieving low coupling.

import abc

class AbstractChartRenderer:

"""

Documentation TBD

"""

def *\_init\_*(self) -> None:

pass

def render(self, notation=None, params=None):

raise NotImplementedError

Listing X: Code snippet showing the AbstractChartRenderer class, which provides a method interface for rendering charts.

#### The ‘AltairChartRenderer’ Class

Extending the AbstractChartRenderer is the AltairChartRenderer class (Listing X). This specialized class serves as a wrapper for Vega-Altair, utilizing the Altair library to perform the rendering of visualizations. One of its key features is the flexibility of outputting the rendered chart through a file pointer (fp). This fp can be either a string representing a file path or an in-memory file-like object such as a StringIO object. This offers versatility for different use-cases, including real-time chart generation and embedding charts into web applications.

By overriding the render method, this class takes in a chart notation and a file pointer (fp) parameter. The chart is generated from the notation and saved in SVG format to the location pointed to by fp.

import altair

from .abstract\_chart\_renderer import AbstractChartRenderer

class AltairChartRenderer(AbstractChartRenderer):

"""

Documentation TBD

"""

def \_init\_(self) -> None:

super().\_\_init\_\_()

def render(self, fp, notation=None, params=None):

chart = altair.Chart.from\_dict(notation)

chart.save(fp, format="svg")

return fp

Listing X: Code snippet showing the AltairChartRenderer class, which acts as a wrapper for Vega-Altair and is responsible for rendering charts using the Altair library.

### The ‘chart\_deployers’ Module

The chart\_deployers module serves as the component in the VV Framework that specializes in the deployment of visualizations. This module introduces an abstract class, AbstractChartDeployer, which acts as a blueprint for various chart deployment strategies, including concrete implementations like GdriveChartDeployer and MiroChartDeployer. These implementations provide specialized mechanisms for deploying charts to Google Drive and Miro boards, respectively. The design of the module encourages low coupling, allowing easy integration of different deployment methods without altering the core framework.

#### The AbstractChartDeployer Class

The foundational class in this architecture is AbstractChartDeployer, an abstract class that defines the standard for all chart deployers (Listing X). It declares a method deploy\_chart(buffer, params=None), which is designed to be overridden by subclasses to offer the actual chart deployment functionality.

import io

import abc

class AbstractChartDeployer:

"""

Documentation TBD

"""

@abc.abstractmethod

def deploy\_chart(buffer: io.BytesIO, params=None) -> None:

"""

Documentation TBD

"""

raise NotImplementedError()

Listing X: Code snippet showing the AbstractChartDeployer class, which provides a method interface for deploying charts.

#### The ‘GdriveChartDeployer’ Class

Extending the AbstractChartDeployer is the GdriveChartDeployer class (Listing X). This concrete implementation leverages Google Drive's API for the deployment of visualizations. It uses the google-auth and google-api-python-client libraries for secure and authenticated communication with Google Drive.

class GdriveChartDeployer(AbstractChartDeployer):

DEFAULT\_SA\_PATH = "./service\_account.json"

DEFAULT\_SCOPES = ['<https://www.googleapis.com/auth/drive>']

DEFAULT\_FILE\_NAME = "filename.svg"

def *\_init\_*(self, folder\_id, mime\_type=None, sa\_path=None, params=None):

self.sa\_path = sa\_path or self.DEFAULT\_SA\_PATH

self.auth = sa.Credentials.from\_service\_account\_file(

self.sa\_path,

scopes=self.DEFAULT\_SCOPES

)

self.drive\_service = build('drive', 'v3', credentials=self.auth)

self.folder\_id = folder\_id

self.file\_name = params.get("filename") if params else self.DEFAULT\_FILE\_NAME

self.mime\_type = mime\_type

def deploy(self, fp):

files = []

file\_metadata = {

'name': self.file\_name,

'parents': [self.folder\_id],

}

if hasattr(fp, 'getvalue'):

content = BytesIO(fp.getvalue().encode("utf-8"))

elif isinstance(fp, (str, bytes, os.PathLike)):

with open(fp, 'rb') as file:

content = file.read()

else:

raise TypeError("fp must be a file-like object or a file path")

#...

response = request.execute()

return response.get('id')

Listing X: Code snippet showing the GdriveChartDeployer class, responsible for deploying charts to Google Drive.

#### The ‘MiroChartDeployer’ Class

Another subclass of AbstractChartDeployer is the MiroChartDeployer class (Listing X). This specialized class is designed for deploying charts to Miro boards. It uses Miro's REST API for communication with Miro boards.



Listing X: Code snippet showing the MiroChartDeployer class, specialized in deploying charts to Miro boards.

The MiroChartDeployer class encapsulates a set of attributes and methods designed to automate the deployment of charts onto a Miro board. Within the class, several attributes warrant particular attention for their role in shaping the class functionality:

* Default Constants: A suite of class-level constants prefixed with DEFAULT\_ is defined to establish fallback values for various properties.
* deployment\_counter: This attribute serves as a counter of the number of deployments executed through the deploy method.
* row\_elements\_height: This list-based attribute is specifically designed to capture the height of individual elements within each row on the Miro board. The data stored in this list informs the layout calculations, facilitating the arrangement of multiple widgets on the board.
* last\_widget\_id: After each successful deployment, the ID of the last deployed widget is stored in this attribute for later manipulation (namely getting the widget height for layout calculations).

The deploy(fp) method is responsible for actually uploading a chart as an image widget onto a Miro board. It accepts the parameter fp, which stands for file pointer.

class MiroChartDeployer (AbstractChartDeployer):

DEFAULT\_IMAGE\_WIDTH=2000

DEFAULT\_IMAGE\_X\_POSITION=0

DEFAULT\_IMAGE\_Y\_POSITION=0

DEFAULT\_IMAGE\_TITLE="Default Image Title"

DEFAULT\_LAYOUT\_COLUMNS=2

DEFAULT\_LAYOUT\_COLUMN\_SPACING=150

DEFAULT\_LAYOUT\_ROW\_SPACING=150

def *\_init\_*(self, board\_id, token, params=None):

self.board\_id = board\_id

self.oauth\_token = token

self.parent\_id = params.get("parent\_id") if params else None

self.image\_title = params.get("image\_title") if params else self.DEFAULT\_IMAGE\_TITLE

self.image\_width = params.get("image\_width") if params else self.DEFAULT\_IMAGE\_WIDTH

self.image\_x\_position = params.get("image\_x\_position") if params else self.DEFAULT\_IMAGE\_X\_POSITION

self.image\_y\_position = params.get("image\_y\_position") if params else self.DEFAULT\_IMAGE\_Y\_POSITION

self.layout\_columns = params.get("layout\_columns") if params else self.DEFAULT\_LAYOUT\_COLUMNS

self.layout\_x\_position = params.get("layout\_x\_position") if params else self.DEFAULT\_IMAGE\_X\_POSITION

self.layout\_row\_spacing = params.get("layout\_row\_spacing") if params else self.DEFAULT\_LAYOUT\_ROW\_SPACING

self.layout\_column\_spacing = params.get("layout\_column\_spacing") if params else self.DEFAULT\_LAYOUT\_COLUMN\_SPACING

self.deployment\_counter = 0

self.row\_elements\_height = []

self.last\_widget\_id = None

def calc\_position(self,last\_widget\_id=None):

#...

def get\_widget\_attribute(self, widget\_id, attribute\_path):

#...

def deploy(self, fp,

#...

Listing X: Code snippet showing the deploy method of the MiroChartDeployer class.

The calc\_position method is designed to calculate the position for placing a new image widget on the Miro board according to the parameters defined for a given structured layout such as number of columns and column and row spacing.

The get\_widget\_attribute method serves the purpose of fetching specific attributes from a widget already deployed on the Miro board. It takes two parameters: widget\_id, the ID of the widget from which an attribute needs to be fetched, and attribute\_path, a list describing the nested keys to reach the target attribute in the widget's data structure. It is specifically used to get the height of the last widget which is essential for determining how much vertical space a row of widgets will occupy in a structured layout with multiple rows and columns. Specifically, the height attribute helps to calculate the next y-coordinate (image\_y\_position) for starting a new row of widgets.

In the calc\_position method, after each widget deployment, the height of the last deployed widget is fetched and stored in the row\_elements\_height list. When it's time to move to a new row, i.e., when the number of widgets in the current row equals the predefined maximum number of columns (layout\_columns), the maximum height in the row\_elements\_height list is used to calculate the new y-coordinate.