DDD API MODELS

This repository / data set contains coded models and data from a qualitative on the interrelation of DDD and APIs. The paper on the study is:

Apitchaka Singjai, Uwe Zdun, Olaf Zimmermann. Practitioner Views on the Interrelation of Microservice APIs and Domain-Driven Design: A Grey Literature Study Based on Grounded Theory. Accepted for publication in the 18th IEEE International Conference on Software Architecture (ICSA 2021).

IMPORTANT: Please note that the Code in the repository only regenerates the PlantUML models and the Latex tables used in the paper. Mainly, this artifact contains the data on the open and axial coding, as well as the formal coding in Python.

This repository / data set contains the following elements:

- The open coding data for each of the sources can be found a separate .md file in the folder:`field_notes_open_coding_axial_coding`
- In each file we also documented the axial coding steps performed in the coded models, to ensure traceability between open coding and formal coding in Python
- All classes and instances are in the `api_ddd_models` folder
    - The file `api_ddd_models.py` contains all models of ADDs
    - The file `sources_and_codes_models.py` contains all models of evidences
- The `generators` folder contains generators for Plant UML models, markdown files, and Latex tables. These are generated into `_generated'. This markdown file renders almost the complete model.
- The file `GT_coding.py` in the `metamodels` folder shows the meta-class of grounded theory coding
- The `map_models` folder´ is a dependency model.

Getting Started

In order to generate the files, the following artifacts must be downloaded and installed.

- [Codeable Models](https://github.com/uzdun/CodeableModels/)
- [PlantUML](http://plantuml.com/download)
- [Graphviz](https://graphviz.org/download/)

Take a look at: generators/generate_all.py

which generates all figures for the models and .md/latex files. Run it in the `generators` directory using: **python .\generate_all.py**

Prerequisites

PYTHONPATH must be correctly set:

- The directory containing `codeableModels` and `plantUMLRenderer` must be on the PYTHONPATH.
- The directory containing this project must be on the PYTHONPATH.

See plant_uml_renderer directory in Codeable Models for instructions how to set up the plant UML jar. Without further configuration it is assumed to be in the default directory:

```

self.directory = "../_generated"

self.plant_uml_jar_path = "../../libs/plantuml.jar"

```

Remarks:

To double-check we tested on a couple of other machines. In the following configurations our instructions worked without problems for us.

1. Installing prerequisite
   - https://plantuml.com/starting
   - https://plantuml.com/graphviz-dot
2. Setting PYTHONPATH
   - ../DDDAPIModelsArtifact/CodeableModels
   - ../DDDAPIModelsArtifact/python/ddd_api_codeablemodels
3. Running the script
   - ../DDDAPIModelsArtifact/python/ddd_api_codeablemodels\generators
4. Getting the results
   - ../DDDAPIModelsArtifact/python/ddd_api_codeablemodels_generated

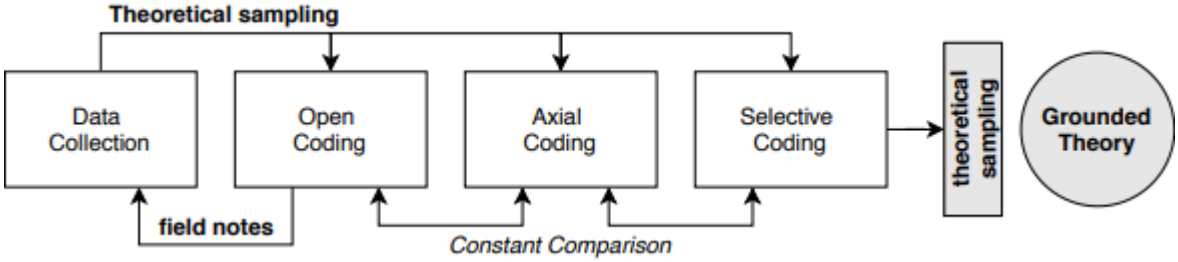
If everything is alright, you will see four subdirectories in the `_generated` directory.

- ddd_api
- domain_model
- latex_files
- Textual_model_rendering

Please note that, this artifact have been test with Windows10 + python3.6 or more, Ubuntu18.04.3 + python3.6, macOS Big sur + python3.6

Reproduction

Below you find a clear step-by-step description how we have achieved the following data collection and coding steps.



1. How to analyze the source in field notes, here using the example of: s1.md
    a. Fill in the attributes for the source (mandatory) and the reference (if it exists), e.g.:

````
# s
1
## url
https://nhpatt.com/bounded-context-in-apis/
## tiny url
https://tinyurl.com/api-ddd-s1
## archive url
https://bit.ly/2BsvnjK
## title
Bounded Context in APIs (1/2)
## source code
no
## example
no
## source type
Practitioner Audience Article
## author type
Practitioner
**AXIAL CODING TRACE:**
``` python
s1 = CClass(source, "s1", values={
    "title": "Bounded Context in APIs (1/2)",
    "url": "https://nhpatt.com/bounded-context-in-apis/",
    "archive url": "https://bit.ly/2BsvnjK",
    "author type": "Practitioner",
    "type": "Practitioner Audience Article",
    "example": False,
    "source code": False})
ddd_vernon_book_2013 = CClass(reference, "vernon2013implementing", values={
    "bibliographic reference": "Vernon, Vaughn: Implementing domain-
driven design}" + "Addison-Wesley Professional, 2013",
    "author type": "Practitioner",
    "type": "Practitioner Book"})
add_links({s_example: ddd_vernon_book_2013}, role_name="referenced")
  ```
````

2. An example of coding from the example file: s1.md. Of course, other structures for recording the coding process can be chosen in a replication.

Lines from knowledge sources:
This is an open question: Does it makes sense to expose the bounded contexts (from DDD) in your REST APIs?
Open Coding:
- Option: Expose Bounded Context in API - this is not REST specific, would be the same question in grpc API, so removed REST here. - This is about how to map the domain model to the API; the context of the decision is the domain model.
Axial Coding:

`expose_bounded_context_in_API = CClass(practice, "Expose Bounded Context in API")`
`domain_model_mapping_decision = CClass(decision, "How to map a domain model and its elements to an API?")`
`add_links({domain_model_mapping_decision: domain_model_and_api}, role_name="context", stereotype_instances=decide_for_some_instances_of)`

Selective Coding:
`expose_each_bounded_context_as_an_API = CClass(practice, "Expose Each Bounded Context as an API", superclasses=expose_bounded_contexts_as_APIs)` `expose_selected_bounded_context_as_an_API = CClass(practice, "Expose Selected Bounded Contexts as APIs", superclasses=expose_bounded_contexts_as_APIs)`

3. How to add the evidences into the model coded in Python. Of course, for replication any other modelling tool or informal coding only can be used, too.

```
add_links({s30: [  api_as_contract_decision, api_contract_specified,
      api_contract_specified_first, api_code_first, api_stability,
      api_modifiability,separation_of_api_contract_and_domain_concerns,
      initial_effort_required, design_and_implementation_effort,
      maintainability_of_api_and_consumers, api_understandability
                  ]}, role_name="contained_code")
```

4. How to generate the figures of ADDs in **python .\generate_all.py.** (To reproduce the generation of figures simply run the whole file as a Python script).

```
from plant_uml_renderer import PlantUMLGenerator
from api_ddd_models.api_ddd_model import ddd_api_views

# UMLgenerator
generator = PlantUMLGenerator(delete_gen_dir_during_init=True)
generator.object_model_renderer.name_break_length = 45
generator.object_model_renderer.left_to_right = True

object_models = {'ddd_api': ddd_api_views}
for key, value in object_models.items():
    generator.generate_object_models(key, value)
```

5. How to generate the tables in **python .\generate_all.py.** (To reproduce the generation of figures simply run the whole file as a Python script).
    a. knowledge source table : generate_sources_table()
    b. study result table : generate_overview_table()

```
from generators.generate_latex_files import LatexTableRenderer
from metamodels.guidance_metamodel import decision

# Latex
source_table_renderer = LatexTableRenderer()
source_table_renderer.generate(decisions)
```