Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут”

Кафедра АСОІУ

**ЗВІТ**

про виконання комп’ютерного практикуму №8

з дисципліни

«Архітектура програмного забезпечення»

Тема: «Зворотне проектування»

|  |  |
| --- | --- |
| Прийняв: | Виконала: |
| Кузнєцов  Олександр  Вікторович | студент 3-го курсу  гр. ІП-51 ФІОТ  Зарічковий Олександр Анатолійович |

Київ – 2017

**ЗМІСТ:**

1. Мета РОБОТИ 3
2. Постановка задачі 4
3. Результати та пояснення 5
4. ЗАПИТАННЯ ТА ВІДПОВІДІ 30
5. Висновок 32

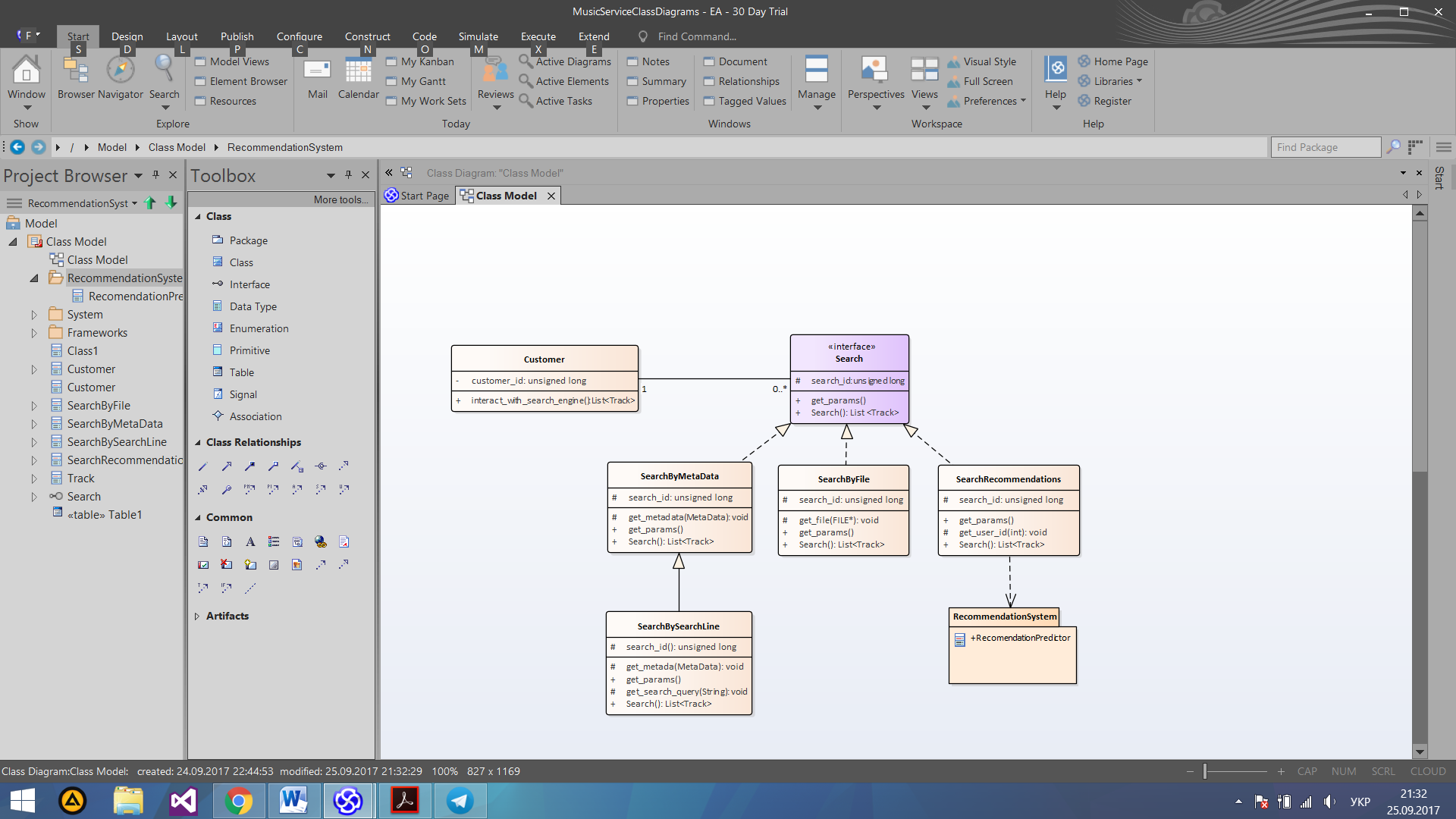
# Мета РОБОТИ

Ознайомитись з принципами зворотного проектування. Навчитися генерувати діаграми класів з початкового коду.

# Постановка задачі

* Використати діаграму класів одного з процесів з попередньої лабораторної роботи та написати для неї програмний код. В коді програми обов’язково реалізувати всі види зв’язку, та описати всі поля і методи, що були позначені на діаграмі класів.
* Використовуючи програмний засіб для побудови UML-діаграм, на основі написаного коду згенерувати UML-діаграму класів.
* Порівняти початкову UML-діаграму, з попередньої лабораторної роботи з отриманою. В випадку якщо діаграми відрізняються описати в чому різниця та чим вона пояснюється.

# Результати та пояснення

**Початкова діаграма класів:** 

**Програмний код на основі діаграми:**

# pylint: disable=protected-access

"""Home of the Sequential model, and the `save\_model`/`load\_model` functions.

"""

from \_\_future\_\_ import absolute\_import

from \_\_future\_\_ import division

from \_\_future\_\_ import print\_function

import copy

import json

import os

import numpy as np

from tensorflow.python.framework import ops

from tensorflow.python.keras.\_impl.keras import backend as K

from tensorflow.python.keras.\_impl.keras import layers as layer\_module

from tensorflow.python.keras.\_impl.keras import optimizers

from tensorflow.python.keras.\_impl.keras.engine import topology

from tensorflow.python.keras.\_impl.keras.engine.topology import Input

from tensorflow.python.keras.\_impl.keras.engine.topology import Layer

from tensorflow.python.keras.\_impl.keras.engine.topology import TFBaseLayer

from tensorflow.python.keras.\_impl.keras.engine.training import Model

from tensorflow.python.keras.\_impl.keras.utils.generic\_utils import has\_arg

from tensorflow.python.keras.\_impl.keras.utils.io\_utils import ask\_to\_proceed\_with\_overwrite

from tensorflow.python.platform import tf\_logging as logging

# pylint: disable=g-import-not-at-top

try:

import h5py

except ImportError:

h5py = None

try:

import yaml

except ImportError:

yaml = None

# pylint: enable=g-import-not-at-top

def save\_model(model, filepath, overwrite=True, include\_optimizer=True):

"""Save a model to a HDF5 file.

The saved model contains:

- the model's configuration (topology)

- the model's weights

- the model's optimizer's state (if any)

Thus the saved model can be reinstantiated in

the exact same state, without any of the code

used for model definition or training.

Arguments:

model: Keras model instance to be saved.

filepath: String, path where to save the model.

overwrite: Whether we should overwrite any existing

model at the target location, or instead

ask the user with a manual prompt.

include\_optimizer: If True, save optimizer's state together.

Raises:

ImportError: if h5py is not available.

"""

if h5py is None:

raise ImportError('`save\_model` requires h5py.')

def get\_json\_type(obj):

"""Serialize any object to a JSON-serializable structure.

Arguments:

obj: the object to serialize

Returns:

JSON-serializable structure representing `obj`.

Raises:

TypeError: if `obj` cannot be serialized.

"""

# if obj is a serializable Keras class instance

# e.g. optimizer, layer

if hasattr(obj, 'get\_config'):

return {'class\_name': obj.\_\_class\_\_.\_\_name\_\_, 'config': obj.get\_config()}

# if obj is any numpy type

if type(obj).\_\_module\_\_ == np.\_\_name\_\_:

if isinstance(obj, np.ndarray):

return {'type': type(obj), 'value': obj.tolist()}

else:

return obj.item()

# misc functions (e.g. loss function)

if callable(obj):

return obj.\_\_name\_\_

# if obj is a python 'type'

if type(obj).\_\_name\_\_ == type.\_\_name\_\_:

return obj.\_\_name\_\_

raise TypeError('Not JSON Serializable:', obj)

from tensorflow.python.keras.\_impl.keras import \_\_version\_\_ as keras\_version # pylint: disable=g-import-not-at-top

# If file exists and should not be overwritten.

if not overwrite and os.path.isfile(filepath):

proceed = ask\_to\_proceed\_with\_overwrite(filepath)

if not proceed:

return

with h5py.File(filepath, mode='w') as f:

f.attrs['keras\_version'] = str(keras\_version).encode('utf8')

f.attrs['backend'] = K.backend().encode('utf8')

f.attrs['model\_config'] = json.dumps(

{

'class\_name': model.\_\_class\_\_.\_\_name\_\_,

'config': model.get\_config()

},

default=get\_json\_type).encode('utf8')

model\_weights\_group = f.create\_group('model\_weights')

model\_layers = model.layers

topology.save\_weights\_to\_hdf5\_group(model\_weights\_group, model\_layers)

if include\_optimizer and hasattr(model, 'optimizer'):

if isinstance(model.optimizer, optimizers.TFOptimizer):

logging.warning(

'TensorFlow optimizers do not '

'make it possible to access '

'optimizer attributes or optimizer state '

'after instantiation. '

'As a result, we cannot save the optimizer '

'as part of the model save file.'

'You will have to compile your model again after loading it. '

'Prefer using a Keras optimizer instead '

'(see keras.io/optimizers).')

else:

f.attrs['training\_config'] = json.dumps(

{

'optimizer\_config': {

'class\_name': model.optimizer.\_\_class\_\_.\_\_name\_\_,

'config': model.optimizer.get\_config()

},

'loss': model.loss,

'metrics': model.metrics,

'sample\_weight\_mode': model.sample\_weight\_mode,

'loss\_weights': model.loss\_weights,

},

default=get\_json\_type).encode('utf8')

# Save optimizer weights.

symbolic\_weights = getattr(model.optimizer, 'weights')

if symbolic\_weights:

optimizer\_weights\_group = f.create\_group('optimizer\_weights')

weight\_values = K.batch\_get\_value(symbolic\_weights)

weight\_names = []

for w, val in zip(symbolic\_weights, weight\_values):

name = str(w.name)

weight\_names.append(name.encode('utf8'))

optimizer\_weights\_group.attrs['weight\_names'] = weight\_names

for name, val in zip(weight\_names, weight\_values):

param\_dset = optimizer\_weights\_group.create\_dataset(

name, val.shape, dtype=val.dtype)

if not val.shape:

# scalar

param\_dset[()] = val

else:

param\_dset[:] = val

f.flush()

def load\_model(filepath, custom\_objects=None, compile=True): # pylint: disable=redefined-builtin

"""Loads a model saved via `save\_model`.

Arguments:

filepath: String, path to the saved model.

custom\_objects: Optional dictionary mapping names

(strings) to custom classes or functions to be

considered during deserialization.

compile: Boolean, whether to compile the model

after loading.

Returns:

A Keras model instance. If an optimizer was found

as part of the saved model, the model is already

compiled. Otherwise, the model is uncompiled and

a warning will be displayed. When `compile` is set

to False, the compilation is omitted without any

warning.

Raises:

ImportError: if h5py is not available.

ValueError: In case of an invalid savefile.

"""

if h5py is None:

raise ImportError('`load\_model` requires h5py.')

if not custom\_objects:

custom\_objects = {}

def convert\_custom\_objects(obj):

"""Handles custom object lookup.

Arguments:

obj: object, dict, or list.

Returns:

The same structure, where occurrences

of a custom object name have been replaced

with the custom object.

"""

if isinstance(obj, list):

deserialized = []

for value in obj:

deserialized.append(convert\_custom\_objects(value))

return deserialized

if isinstance(obj, dict):

deserialized = {}

for key, value in obj.items():

deserialized[key] = convert\_custom\_objects(value)

return deserialized

if obj in custom\_objects:

return custom\_objects[obj]

return obj

with h5py.File(filepath, mode='r') as f:

# instantiate model

model\_config = f.attrs.get('model\_config')

if model\_config is None:

raise ValueError('No model found in config file.')

model\_config = json.loads(model\_config.decode('utf-8'))

model = model\_from\_config(model\_config, custom\_objects=custom\_objects)

# set weights

topology.load\_weights\_from\_hdf5\_group(f['model\_weights'], model.layers)

# Early return if compilation is not required.

if not compile:

return model

# instantiate optimizer

training\_config = f.attrs.get('training\_config')

if training\_config is None:

logging.warning('No training configuration found in save file: '

'the model was \*not\* compiled. Compile it manually.')

return model

training\_config = json.loads(training\_config.decode('utf-8'))

optimizer\_config = training\_config['optimizer\_config']

optimizer = optimizers.deserialize(

optimizer\_config, custom\_objects=custom\_objects)

# Recover loss functions and metrics.

loss = convert\_custom\_objects(training\_config['loss'])

metrics = convert\_custom\_objects(training\_config['metrics'])

sample\_weight\_mode = training\_config['sample\_weight\_mode']

loss\_weights = training\_config['loss\_weights']

# Compile model.

model.compile(

optimizer=optimizer,

loss=loss,

metrics=metrics,

loss\_weights=loss\_weights,

sample\_weight\_mode=sample\_weight\_mode)

# Set optimizer weights.

if 'optimizer\_weights' in f:

# Build train function (to get weight updates).

if isinstance(model, Sequential):

model.model.\_make\_train\_function()

else:

model.\_make\_train\_function()

optimizer\_weights\_group = f['optimizer\_weights']

optimizer\_weight\_names = [

n.decode('utf8')

for n in optimizer\_weights\_group.attrs['weight\_names']

]

optimizer\_weight\_values = [

optimizer\_weights\_group[n] for n in optimizer\_weight\_names

]

try:

model.optimizer.set\_weights(optimizer\_weight\_values)

except ValueError:

logging.warning('Error in loading the saved optimizer '

'state. As a result, your model is '

'starting with a freshly initialized '

'optimizer.')

return model

def model\_from\_config(config, custom\_objects=None):

"""Instantiates a Keras model from its config.

Arguments:

config: Configuration dictionary.

custom\_objects: Optional dictionary mapping names

(strings) to custom classes or functions to be

considered during deserialization.

Returns:

A Keras model instance (uncompiled).

Raises:

TypeError: if `config` is not a dictionary.

"""

if isinstance(config, list):

raise TypeError('`model\_from\_config` expects a dictionary, not a list. '

'Maybe you meant to use '

'`Sequential.from\_config(config)`?')

return layer\_module.deserialize(config, custom\_objects=custom\_objects)

def model\_from\_yaml(yaml\_string, custom\_objects=None):

"""Parses a yaml model configuration file and returns a model instance.

Arguments:

yaml\_string: YAML string encoding a model configuration.

custom\_objects: Optional dictionary mapping names

(strings) to custom classes or functions to be

considered during deserialization.

Returns:

A Keras model instance (uncompiled).

Raises:

ImportError: if yaml module is not found.

"""

if yaml is None:

raise ImportError('Requires yaml module installed.')

config = yaml.load(yaml\_string)

return layer\_module.deserialize(config, custom\_objects=custom\_objects)

def model\_from\_json(json\_string, custom\_objects=None):

"""Parses a JSON model configuration file and returns a model instance.

Arguments:

json\_string: JSON string encoding a model configuration.

custom\_objects: Optional dictionary mapping names

(strings) to custom classes or functions to be

considered during deserialization.

Returns:

A Keras model instance (uncompiled).

"""

config = json.loads(json\_string)

return layer\_module.deserialize(config, custom\_objects=custom\_objects)

class Sequential(Model):

"""Linear stack of layers.

Arguments:

layers: list of layers to add to the model.

# Note

The first layer passed to a Sequential model

should have a defined input shape. What that

means is that it should have received an `input\_shape`

or `batch\_input\_shape` argument,

or for some type of layers (recurrent, Dense...)

an `input\_dim` argument.

Example:

```python

model = Sequential()

# first layer must have a defined input shape

model.add(Dense(32, input\_dim=500))

# afterwards, Keras does automatic shape inference

model.add(Dense(32))

# also possible (equivalent to the above):

model = Sequential()

model.add(Dense(32, input\_shape=(500,)))

model.add(Dense(32))

# also possible (equivalent to the above):

model = Sequential()

# here the batch dimension is None,

# which means any batch size will be accepted by the model.

model.add(Dense(32, batch\_input\_shape=(None, 500)))

model.add(Dense(32))

```

"""

def \_\_init\_\_(self, layers=None, name=None):

self.layers = [] # Stack of layers.

self.model = None # Internal Model instance.

self.inputs = [] # List of input tensors

self.outputs = [] # List of length 1: the output tensor (unique).

self.\_trainable = True

self.\_initial\_weights = None

self.\_input\_layers = []

# Model attributes.

self.inbound\_nodes = []

self.outbound\_nodes = []

self.built = False

# Set model name.

if not name:

prefix = 'sequential\_'

name = prefix + str(K.get\_uid(prefix))

self.name = name

# The following properties are not actually used by Keras;

# they exist for compatibility with TF's variable scoping mechanism.

self.\_updates = []

self.\_losses = []

self.\_scope = None

self.\_reuse = None

self.\_base\_name = name

self.\_graph = ops.get\_default\_graph()

# Add to the model any layers passed to the constructor.

if layers:

for layer in layers:

self.add(layer)

def add(self, layer):

"""Adds a layer instance on top of the layer stack.

Arguments:

layer: layer instance.

Raises:

TypeError: If `layer` is not a layer instance.

ValueError: In case the `layer` argument does not

know its input shape.

ValueError: In case the `layer` argument has

multiple output tensors, or is already connected

somewhere else (forbidden in `Sequential` models).

"""

if not isinstance(layer, (Layer, TFBaseLayer)):

raise TypeError('The added layer must be '

'an instance of class Layer. '

'Found: ' + str(layer))

if not self.outputs:

# first layer in model: check that it is an input layer

if not layer.inbound\_nodes:

# create an input layer

if not hasattr(layer, 'batch\_input\_shape'):

raise ValueError('The first layer in a '

'Sequential model must '

'get an `input\_shape` or '

'`batch\_input\_shape` argument.')

# Instantiate the input layer.

x = Input(

batch\_shape=layer.batch\_input\_shape,

dtype=layer.dtype,

name=layer.name + '\_input')

# This will build the current layer

# and create the node connecting the current layer

# to the input layer we just created.

layer(x)

if len(layer.inbound\_nodes) != 1:

raise ValueError('A layer added to a Sequential model must '

'not already be connected somewhere else. '

'Model received layer ' + layer.name + ' which has ' +

str(len(layer.inbound\_nodes)) +

' pre-existing inbound connections.')

if len(layer.inbound\_nodes[0].output\_tensors) != 1:

raise ValueError('All layers in a Sequential model '

'should have a single output tensor. '

'For multi-output layers, '

'use the functional API.')

self.outputs = [layer.inbound\_nodes[0].output\_tensors[0]]

self.inputs = topology.get\_source\_inputs(self.outputs[0])

# We create an input node, which we will keep updated

# as we add more layers

topology.Node(

outbound\_layer=self,

inbound\_layers=[],

node\_indices=[],

tensor\_indices=[],

input\_tensors=self.inputs,

output\_tensors=self.outputs)

else:

output\_tensor = layer(self.outputs[0])

if isinstance(output\_tensor, list):

raise TypeError('All layers in a Sequential model '

'should have a single output tensor. '

'For multi-output layers, '

'use the functional API.')

self.outputs = [output\_tensor]

# update self.inbound\_nodes

self.inbound\_nodes[0].output\_tensors = self.outputs

self.inbound\_nodes[0].output\_shapes = [K.int\_shape(self.outputs[0])]

self.layers.append(layer)

self.built = False

def pop(self):

"""Removes the last layer in the model.

Raises:

TypeError: if there are no layers in the model.

"""

if not self.layers:

raise TypeError('There are no layers in the model.')

self.layers.pop()

if not self.layers:

self.outputs = []

self.inbound\_nodes = []

self.outbound\_nodes = []

else:

self.layers[-1].outbound\_nodes = []

self.outputs = [self.layers[-1].output]

# update self.inbound\_nodes

self.inbound\_nodes[0].output\_tensors = self.outputs

self.inbound\_nodes[0].output\_shapes = [K.int\_shape(self.outputs[0])]

self.built = False

def get\_layer(self, name=None, index=None):

"""Retrieve a layer that is part of the model.

Returns a layer based on either its name (unique)

or its index in the graph. Indices are based on

order of horizontal graph traversal (bottom-up).

Arguments:

name: string, name of layer.

index: integer, index of layer.

Returns:

A layer instance.

"""

if not self.built:

self.build()

return self.model.get\_layer(name, index)

def call(self, inputs, mask=None):

if not self.built:

self.build()

return self.model.call(inputs, mask)

def build(self, input\_shape=None):

if not self.inputs or not self.outputs:

raise TypeError('Sequential model cannot be built: model is empty.'

' Add some layers first.')

# actually create the model

self.model = Model(self.inputs, self.outputs[0], name=self.name + '\_model')

self.model.trainable = self.trainable

# mirror model attributes

self.supports\_masking = self.model.supports\_masking

self.\_output\_mask\_cache = self.model.\_output\_mask\_cache

self.\_output\_tensor\_cache = self.model.\_output\_tensor\_cache

self.\_output\_shape\_cache = self.model.\_output\_shape\_cache

self.\_input\_layers = self.model.\_input\_layers

self.\_output\_layers = self.model.\_output\_layers

self.\_input\_coordinates = self.model.\_input\_coordinates

self.\_output\_coordinates = self.model.\_output\_coordinates

self.\_nodes\_by\_depth = self.model.\_nodes\_by\_depth

self.\_network\_nodes = self.model.\_network\_nodes

self.output\_names = self.model.output\_names

self.input\_names = self.model.input\_names

self.\_feed\_input\_names = self.model.\_feed\_input\_names

self.\_feed\_inputs = self.model.\_feed\_inputs

# Make sure child model callbacks

# will call the parent Sequential model.

self.model.callback\_model = self

self.built = True

@property

def uses\_learning\_phase(self):

if not self.built:

self.build()

return self.model.uses\_learning\_phase

def \_gather\_list\_attr(self, attr):

all\_attrs = []

for layer in self.layers:

all\_attrs += getattr(layer, attr, [])

return all\_attrs

@property

def trainable(self):

return self.\_trainable

@trainable.setter

def trainable(self, value):

if self.model:

self.model.trainable = value

self.\_trainable = value

@property

def trainable\_weights(self):

if not self.trainable:

return []

return self.\_gather\_list\_attr('trainable\_weights')

@property

def non\_trainable\_weights(self):

weights = self.\_gather\_list\_attr('non\_trainable\_weights')

if not self.trainable:

trainable\_weights = self.\_gather\_list\_attr('trainable\_weights')

return trainable\_weights + weights

return weights

@property

def updates(self):

if not self.built:

self.build()

return self.model.updates

@property

def state\_updates(self):

if not self.built:

self.build()

return self.model.state\_updates

def get\_updates\_for(self, inputs):

if not self.built:

self.build()

return self.model.get\_updates\_for(inputs)

@property

def losses(self):

if not self.built:

self.build()

return self.model.losses

def get\_losses\_for(self, inputs):

if not self.built:

self.build()

return self.model.get\_losses\_for(inputs)

@property

def regularizers(self):

if not self.built:

self.build()

return self.model.regularizers

def get\_weights(self):

"""Retrieves the weights of the model.

Returns:

A flat list of Numpy arrays

(one array per model weight).

"""

if not self.built:

self.build()

return self.model.get\_weights()

def set\_weights(self, weights):

"""Sets the weights of the model.

Arguments:

weights: Should be a list

of Numpy arrays with shapes and types matching

the output of `model.get\_weights()`.

"""

if not self.built:

self.build()

self.model.set\_weights(weights)

def load\_weights(self, filepath, by\_name=False):

if h5py is None:

raise ImportError('`load\_weights` requires h5py.')

f = h5py.File(filepath, mode='r')

if 'layer\_names' not in f.attrs and 'model\_weights' in f:

f = f['model\_weights']

layers = self.layers

if by\_name:

topology.load\_weights\_from\_hdf5\_group\_by\_name(f, layers)

else:

topology.load\_weights\_from\_hdf5\_group(f, layers)

if hasattr(f, 'close'):

f.close()

def save\_weights(self, filepath, overwrite=True):

if h5py is None:

raise ImportError('`save\_weights` requires h5py.')

# If file exists and should not be overwritten:

if not overwrite and os.path.isfile(filepath):

proceed = ask\_to\_proceed\_with\_overwrite(filepath)

if not proceed:

return

layers = self.layers

f = h5py.File(filepath, 'w')

topology.save\_weights\_to\_hdf5\_group(f, layers)

f.flush()

f.close()

def compile(self,

optimizer,

loss,

metrics=None,

sample\_weight\_mode=None,

weighted\_metrics=None,

\*\*kwargs):

"""Configures the learning process.

Arguments:

optimizer: str (name of optimizer) or optimizer object.

See [optimizers](/optimizers).

loss: str (name of objective function) or objective function.

See [losses](/losses).

metrics: list of metrics to be evaluated by the model

during training and testing.

Typically you will use `metrics=['accuracy']`.

See [metrics](/metrics).

sample\_weight\_mode: if you need to do timestep-wise

sample weighting (2D weights), set this to "temporal".

"None" defaults to sample-wise weights (1D).

weighted\_metrics: list of metrics to be evaluated and weighted

by `sample\_weight` or `class\_weight` during training and testing.

\*\*kwargs: These are passed into `tf.Session.run`.

Example:

```python

model = Sequential()

model.add(Dense(32, input\_shape=(500,)))

model.add(Dense(10, activation='softmax'))

model.compile(optimizer='rmsprop',

loss='categorical\_crossentropy',

metrics=['accuracy'])

```

"""

# create the underlying model

self.build()

# call compile method of Model class

self.model.compile(

optimizer,

loss,

metrics=metrics,

sample\_weight\_mode=sample\_weight\_mode,

weighted\_metrics=weighted\_metrics,

\*\*kwargs)

self.optimizer = self.model.optimizer

self.loss = self.model.loss

self.total\_loss = self.model.total\_loss

self.loss\_weights = self.model.loss\_weights

self.metrics = self.model.metrics

self.weighted\_metrics = self.model.weighted\_metrics

self.metrics\_tensors = self.model.metrics\_tensors

self.metrics\_names = self.model.metrics\_names

self.sample\_weight\_mode = self.model.sample\_weight\_mode

self.sample\_weights = self.model.sample\_weights

self.targets = self.model.targets

def fit(self,

x,

y,

batch\_size=32,

epochs=10,

verbose=1,

callbacks=None,

validation\_split=0.,

validation\_data=None,

shuffle=True,

class\_weight=None,

sample\_weight=None,

initial\_epoch=0):

"""Trains the model for a fixed number of epochs.

Arguments:

x: input data, as a Numpy array or list of Numpy arrays

(if the model has multiple inputs).

y: labels, as a Numpy array.

batch\_size: integer. Number of samples per gradient update.

epochs: integer, the number of epochs to train the model.

verbose: 0 for no logging to stdout,

1 for progress bar logging, 2 for one log line per epoch.

callbacks: list of `keras.callbacks.Callback` instances.

List of callbacks to apply during training.

See [callbacks](/callbacks).

validation\_split: float (0. < x < 1).

Fraction of the data to use as held-out validation data.

validation\_data: tuple (x\_val, y\_val) or tuple

(x\_val, y\_val, val\_sample\_weights) to be used as held-out

validation data. Will override validation\_split.

shuffle: boolean or str (for 'batch').

Whether to shuffle the samples at each epoch.

'batch' is a special option for dealing with the

limitations of HDF5 data; it shuffles in batch-sized chunks.

class\_weight: dictionary mapping classes to a weight value,

used for scaling the loss function (during training only).

sample\_weight: Numpy array of weights for

the training samples, used for scaling the loss function

(during training only). You can either pass a flat (1D)

Numpy array with the same length as the input samples

(1:1 mapping between weights and samples),

or in the case of temporal data,

you can pass a 2D array with shape (samples, sequence\_length),

to apply a different weight to every timestep of every sample.

In this case you should make sure to specify

sample\_weight\_mode="temporal" in compile().

initial\_epoch: epoch at which to start training

(useful for resuming a previous training run)

Returns:

A `History` object. Its `History.history` attribute is

a record of training loss values and metrics values

at successive epochs, as well as validation loss values

and validation metrics values (if applicable).

Raises:

RuntimeError: if the model was never compiled.

"""

if not self.built:

raise RuntimeError('The model needs to be compiled ' 'before being used.')

return self.model.fit(

x,

y,

batch\_size=batch\_size,

epochs=epochs,

verbose=verbose,

callbacks=callbacks,

validation\_split=validation\_split,

validation\_data=validation\_data,

shuffle=shuffle,

class\_weight=class\_weight,

sample\_weight=sample\_weight,

initial\_epoch=initial\_epoch)

def evaluate(self, x, y, batch\_size=32, verbose=1, sample\_weight=None):

"""Computes the loss on some input data, batch by batch.

Arguments:

x: input data, as a Numpy array or list of Numpy arrays

(if the model has multiple inputs).

y: labels, as a Numpy array.

batch\_size: integer. Number of samples per gradient update.

verbose: verbosity mode, 0 or 1.

sample\_weight: sample weights, as a Numpy array.

Returns:

Scalar test loss (if the model has no metrics)

or list of scalars (if the model computes other metrics).

The attribute `model.metrics\_names` will give you

the display labels for the scalar outputs.

Raises:

RuntimeError: if the model was never compiled.

"""

if not self.built:

raise RuntimeError('The model needs to be compiled ' 'before being used.')

return self.model.evaluate(

x,

y,

batch\_size=batch\_size,

verbose=verbose,

sample\_weight=sample\_weight)

def predict(self, x, batch\_size=32, verbose=0):

"""Generates output predictions for the input samples.

The input samples are processed batch by batch.

Arguments:

x: the input data, as a Numpy array.

batch\_size: integer.

verbose: verbosity mode, 0 or 1.

Returns:

A Numpy array of predictions.

"""

if not self.built:

self.build()

return self.model.predict(x, batch\_size=batch\_size, verbose=verbose)

def predict\_on\_batch(self, x):

"""Returns predictions for a single batch of samples.

Arguments:

x: input data, as a Numpy array or list of Numpy arrays

(if the model has multiple inputs).

Returns:

A Numpy array of predictions.

"""

if not self.built:

self.build()

return self.model.predict\_on\_batch(x)

def train\_on\_batch(self, x, y, class\_weight=None, sample\_weight=None):

"""Single gradient update over one batch of samples.

Arguments:

x: input data, as a Numpy array or list of Numpy arrays

(if the model has multiple inputs).

y: labels, as a Numpy array.

class\_weight: dictionary mapping classes to a weight value,

used for scaling the loss function (during training only).

sample\_weight: sample weights, as a Numpy array.

Returns:

Scalar training loss (if the model has no metrics)

or list of scalars (if the model computes other metrics).

The attribute `model.metrics\_names` will give you

the display labels for the scalar outputs.

Raises:

RuntimeError: if the model was never compiled.

"""

if not self.built:

raise RuntimeError('The model needs to be compiled ' 'before being used.')

return self.model.train\_on\_batch(

x, y, sample\_weight=sample\_weight, class\_weight=class\_weight)

def test\_on\_batch(self, x, y, sample\_weight=None):

"""Evaluates the model over a single batch of samples.

Arguments:

x: input data, as a Numpy array or list of Numpy arrays

(if the model has multiple inputs).

y: labels, as a Numpy array.

sample\_weight: sample weights, as a Numpy array.

Returns:

Scalar test loss (if the model has no metrics)

or list of scalars (if the model computes other metrics).

The attribute `model.metrics\_names` will give you

the display labels for the scalar outputs.

Raises:

RuntimeError: if the model was never compiled.

"""

if not self.built:

raise RuntimeError('The model needs to be compiled ' 'before being used.')

return self.model.test\_on\_batch(x, y, sample\_weight=sample\_weight)

def predict\_proba(self, x, batch\_size=32, verbose=1):

"""Generates class probability predictions for the input samples.

The input samples are processed batch by batch.

Arguments:

x: input data, as a Numpy array or list of Numpy arrays

(if the model has multiple inputs).

batch\_size: integer.

verbose: verbosity mode, 0 or 1.

Returns:

A Numpy array of probability predictions.

"""

preds = self.predict(x, batch\_size, verbose)

if preds.min() < 0. or preds.max() > 1.:

logging.warning('Network returning invalid probability values. '

'The last layer might not normalize predictions '

'into probabilities '

'(like softmax or sigmoid would).')

return preds

def predict\_classes(self, x, batch\_size=32, verbose=1):

"""Generate class predictions for the input samples.

The input samples are processed batch by batch.

Arguments:

x: input data, as a Numpy array or list of Numpy arrays

(if the model has multiple inputs).

batch\_size: integer.

verbose: verbosity mode, 0 or 1.

Returns:

A numpy array of class predictions.

"""

proba = self.predict(x, batch\_size=batch\_size, verbose=verbose)

if proba.shape[-1] > 1:

return proba.argmax(axis=-1)

else:

return (proba > 0.5).astype('int32')

def fit\_generator(self,

generator,

steps\_per\_epoch,

epochs=1,

verbose=1,

callbacks=None,

validation\_data=None,

validation\_steps=None,

class\_weight=None,

max\_queue\_size=10,

workers=1,

use\_multiprocessing=False,

initial\_epoch=0,

\*\*kwargs):

"""Fits the model on data generated batch-by-batch by a Python generator.

The generator is run in parallel to the model, for efficiency.

For instance, this allows you to do real-time data augmentation

on images on CPU in parallel to training your model on GPU.

Arguments:

generator: A generator.

The output of the generator must be either

- a tuple (inputs, targets)

- a tuple (inputs, targets, sample\_weights).

All arrays should contain the same number of samples.

The generator is expected to loop over its data

indefinitely. An epoch finishes when `steps\_per\_epoch`

batches have been seen by the model.

steps\_per\_epoch: Total number of steps (batches of samples)

to yield from `generator` before declaring one epoch

finished and starting the next epoch. It should typically

be equal to the number of unique samples of your dataset

divided by the batch size.

epochs: Integer, total number of iterations on the data.

verbose: Verbosity mode, 0, 1, or 2.

callbacks: List of callbacks to be called during training.

validation\_data: This can be either

- A generator for the validation data

- A tuple (inputs, targets)

- A tuple (inputs, targets, sample\_weights).

validation\_steps: Only relevant if `validation\_data`

is a generator.

Number of steps to yield from validation generator

at the end of every epoch. It should typically

be equal to the number of unique samples of your

validation dataset divided by the batch size.

class\_weight: Dictionary mapping class indices to a weight

for the class.

max\_queue\_size: Maximum size for the generator queue

workers: Maximum number of processes to spin up

use\_multiprocessing: If True, use process based threading.

Note that because

this implementation relies on multiprocessing,

you should not pass

non picklable arguments to the generator

as they can't be passed

easily to children processes.

initial\_epoch: Epoch at which to start training

(useful for resuming a previous training run)

\*\*kwargs: support for legacy arguments.

Returns:

A `History` object.

Raises:

RuntimeError: if the model was never compiled.

ValueError: In case the generator yields

data in an invalid format.

Example:

```python

def generate\_arrays\_from\_file(path):

while 1:

f = open(path)

for line in f:

# create Numpy arrays of input data

# and labels, from each line in the file

x, y = process\_line(line)

yield (x, y)

f.close()

model.fit\_generator(generate\_arrays\_from\_file('/my\_file.txt'),

steps\_per\_epoch=1000, epochs=10)

```

"""

# Legacy support

if 'max\_q\_size' in kwargs:

max\_queue\_size = kwargs.pop('max\_q\_size')

logging.warning('The argument `max\_q\_size` has been renamed '

'`max\_queue\_size`. Update your method calls accordingly.')

if 'pickle\_safe' in kwargs:

use\_multiprocessing = kwargs.pop('pickle\_safe')

logging.warning('The argument `pickle\_safe` has been renamed '

'`use\_multiprocessing`. '

'Update your method calls accordingly.')

if kwargs:

raise ValueError('Unrecognized keyword arguments: ' + str(kwargs))

if not self.built:

raise RuntimeError('The model needs to be compiled ' 'before being used.')

return self.model.fit\_generator(

generator,

steps\_per\_epoch,

epochs,

verbose=verbose,

callbacks=callbacks,

validation\_data=validation\_data,

validation\_steps=validation\_steps,

class\_weight=class\_weight,

max\_queue\_size=max\_queue\_size,

workers=workers,

use\_multiprocessing=use\_multiprocessing,

initial\_epoch=initial\_epoch)

def evaluate\_generator(self,

generator,

steps,

max\_queue\_size=10,

workers=1,

use\_multiprocessing=False,

\*\*kwargs):

"""Evaluates the model on a data generator.

The generator should return the same kind of data

as accepted by `test\_on\_batch`.

Arguments:

generator: Generator yielding tuples (inputs, targets)

or (inputs, targets, sample\_weights)

steps: Total number of steps (batches of samples)

to yield from `generator` before stopping.

max\_queue\_size: maximum size for the generator queue

workers: maximum number of processes to spin up

use\_multiprocessing: if True, use process based threading.

Note that because this implementation

relies on multiprocessing, you should not pass

non picklable arguments to the generator

as they can't be passed easily to children processes.

\*\*kwargs: support for legacy arguments.

Returns:

Scalar test loss (if the model has no metrics)

or list of scalars (if the model computes other metrics).

The attribute `model.metrics\_names` will give you

the display labels for the scalar outputs.

Raises:

RuntimeError: if the model was never compiled.

ValueError: In case the generator yields

data in an invalid format.

"""

# Legacy support

if 'max\_q\_size' in kwargs:

max\_queue\_size = kwargs.pop('max\_q\_size')

logging.warning('The argument `max\_q\_size` has been renamed '

'`max\_queue\_size`. Update your method calls accordingly.')

if 'pickle\_safe' in kwargs:

use\_multiprocessing = kwargs.pop('pickle\_safe')

logging.warning('The argument `pickle\_safe` has been renamed '

'`use\_multiprocessing`. '

'Update your method calls accordingly.')

if kwargs:

raise ValueError('Unrecognized keyword arguments: ' + str(kwargs))

if not self.built:

raise RuntimeError('The model needs to be compiled ' 'before being used.')

return self.model.evaluate\_generator(

generator,

steps,

max\_queue\_size=max\_queue\_size,

workers=workers,

use\_multiprocessing=use\_multiprocessing)

def predict\_generator(self,

generator,

steps,

max\_queue\_size=10,

workers=1,

use\_multiprocessing=False,

verbose=0,

\*\*kwargs):

"""Generates predictions for the input samples from a data generator.

The generator should return the same kind of data as accepted by

`predict\_on\_batch`.

Arguments:

generator: generator yielding batches of input samples.

steps: Total number of steps (batches of samples)

to yield from `generator` before stopping.

max\_queue\_size: maximum size for the generator queue

workers: maximum number of processes to spin up

use\_multiprocessing: if True, use process based threading.

Note that because this implementation

relies on multiprocessing, you should not pass

non picklable arguments to the generator

as they can't be passed easily to children processes.

verbose: verbosity mode, 0 or 1.

\*\*kwargs: support for legacy arguments.

Returns:

A Numpy array of predictions.

Raises:

ValueError: In case the generator yields

data in an invalid format.

"""

# Legacy support

if 'max\_q\_size' in kwargs:

max\_queue\_size = kwargs.pop('max\_q\_size')

logging.warning('The argument `max\_q\_size` has been renamed '

'`max\_queue\_size`. Update your method calls accordingly.')

if 'pickle\_safe' in kwargs:

use\_multiprocessing = kwargs.pop('pickle\_safe')

logging.warning('The argument `pickle\_safe` has been renamed '

'`use\_multiprocessing`. '

'Update your method calls accordingly.')

if kwargs:

raise ValueError('Unrecognized keyword arguments: ' + str(kwargs))

if not self.built:

self.build()

return self.model.predict\_generator(

generator,

steps,

max\_queue\_size=max\_queue\_size,

workers=workers,

use\_multiprocessing=use\_multiprocessing,

verbose=verbose)

def get\_config(self):

config = []

for layer in self.layers:

config.append({

'class\_name': layer.\_\_class\_\_.\_\_name\_\_,

'config': layer.get\_config()

})

return copy.deepcopy(config)

@classmethod

def from\_config(cls, config, custom\_objects=None):

model = cls()

for conf in config:

layer = layer\_module.deserialize(conf, custom\_objects=custom\_objects)

model.add(layer)

return model

def \_clone\_functional\_model(model, input\_tensors=None):

"""Clone a functional `Model` instance.

Model cloning is similar to calling a model on new inputs,

except that it creates new layers (and thus new weights) instead

of sharing the weights of the existing layers.

Arguments:

model: Instance of `Model`.

input\_tensors: optional list of input tensors

to build the model upon. If not provided,

placeholders will be created.

Returns:

An instance of `Model` reproducing the behavior

of the original model, on top of new inputs tensors,

using newly instantiated weights.

Raises:

ValueError: in case of invalid `model` argument value.

"""

if not isinstance(model, Model):

raise ValueError('Expected `model` argument '

'to be a `Model` instance, got ', model)

if isinstance(model, Sequential):

raise ValueError('Expected `model` argument '

'to be a functional `Model` instance, '

'got a `Sequential` instance instead:', model)

layer\_map = {} # Cache for created layers.

tensor\_map = {} # Map {reference\_tensor: (corresponding\_tensor, mask)}

if input\_tensors is None:

# Create placeholders to build the model on top of.

input\_layers = []

input\_tensors = []

for layer in model.\_input\_layers:

input\_tensor = Input(

batch\_shape=layer.batch\_input\_shape,

dtype=layer.dtype,

sparse=layer.sparse,

name=layer.name)

input\_tensors.append(input\_tensor)

# Cache newly created input layer.

newly\_created\_input\_layer = input\_tensor.\_keras\_history[0]

layer\_map[layer] = newly\_created\_input\_layer

for original\_input\_layer, cloned\_input\_layer in zip(model.\_input\_layers,

input\_layers):

layer\_map[original\_input\_layer] = cloned\_input\_layer

else:

# Make sure that all input tensors come from a Keras layer.

# If tensor comes from an input layer: cache the input layer.

input\_tensors = topology.\_to\_list(input\_tensors)

input\_tensors\_ = []

for i, x in enumerate(input\_tensors):

if not K.is\_keras\_tensor(x):

name = model.\_input\_layers[i].name

input\_tensor = Input(tensor=x, name='input\_wrapper\_for\_' + name)

input\_tensors\_.append(input\_tensor)

# Cache newly created input layer.

original\_input\_layer = x.\_keras\_history[0]

newly\_created\_input\_layer = input\_tensor.\_keras\_history[0]

layer\_map[original\_input\_layer] = newly\_created\_input\_layer

else:

input\_tensors\_.append(x)

input\_tensors = input\_tensors\_

for x, y in zip(model.inputs, input\_tensors):

tensor\_map[x] = (y, None) # tensor, mask

# Iterated over every node in the reference model, in depth order.

depth\_keys = list(model.\_nodes\_by\_depth.keys())

depth\_keys.sort(reverse=True)

for depth in depth\_keys:

nodes = model.\_nodes\_by\_depth[depth]

for node in nodes:

# Recover the corresponding layer.

layer = node.outbound\_layer

# Get or create layer.

if layer not in layer\_map:

# Clone layer.

new\_layer = layer.\_\_class\_\_.from\_config(layer.get\_config())

layer\_map[layer] = new\_layer

layer = new\_layer

else:

# Reuse previously cloned layer.

layer = layer\_map[layer]

# Don't call InputLayer multiple times.

if isinstance(layer, topology.InputLayer):

continue

# Gather inputs to call the new layer.

referenceinput\_tensors\_ = node.input\_tensors

reference\_output\_tensors = node.output\_tensors

# If all previous input tensors are available in tensor\_map,

# then call node.inbound\_layer on them.

computed\_data = [] # List of tuples (input, mask).

for x in referenceinput\_tensors\_:

if x in tensor\_map:

computed\_data.append(tensor\_map[x])

if len(computed\_data) == len(referenceinput\_tensors\_):

# Call layer.

if node.arguments:

kwargs = node.arguments

else:

kwargs = {}

if len(computed\_data) == 1:

computed\_tensor, computed\_mask = computed\_data[0]

if has\_arg(layer.call, 'mask'):

if 'mask' not in kwargs:

kwargs['mask'] = computed\_mask

output\_tensors = topology.\_to\_list(layer(computed\_tensor, \*\*kwargs))

output\_masks = topology.\_to\_list(

layer.compute\_mask(computed\_tensor, computed\_mask))

computed\_tensors = [computed\_tensor]

computed\_masks = [computed\_mask]

else:

computed\_tensors = [x[0] for x in computed\_data]

computed\_masks = [x[1] for x in computed\_data]

if has\_arg(layer.call, 'mask'):

if 'mask' not in kwargs:

kwargs['mask'] = computed\_masks

output\_tensors = topology.\_to\_list(layer(computed\_tensors, \*\*kwargs))

output\_masks = topology.\_to\_list(

layer.compute\_mask(computed\_tensors, computed\_masks))

# Update tensor\_map.

for x, y, mask in zip(reference\_output\_tensors, output\_tensors,

output\_masks):

tensor\_map[x] = (y, mask)

# Check that we did compute the model outputs,

# then instantiate a new model from inputs and outputs.

output\_tensors = []

for x in model.outputs:

assert x in tensor\_map, 'Could not compute output ' + str(x)

tensor, \_ = tensor\_map[x]

output\_tensors.append(tensor)

return Model(input\_tensors, output\_tensors, name=model.name)

def \_clone\_sequential\_model(model, input\_tensors=None):

"""Clone a `Sequential` model instance.

Model cloning is similar to calling a model on new inputs,

except that it creates new layers (and thus new weights) instead

of sharing the weights of the existing layers.

Arguments:

model: Instance of `Sequential`.

input\_tensors: optional list of input tensors

to build the model upon. If not provided,

placeholders will be created.

Returns:

An instance of `Sequential` reproducing the behavior

of the original model, on top of new inputs tensors,

using newly instantiated weights.

Raises:

ValueError: in case of invalid `model` argument value.

"""

if not isinstance(model, Sequential):

raise ValueError('Expected `model` argument '

'to be a `Sequential` model instance, '

'but got:', model)

def clone(layer):

return layer.\_\_class\_\_.from\_config(layer.get\_config())

layers = [clone(layer) for layer in model.layers]

if input\_tensors is None:

return Sequential(layers=layers, name=model.name)

else:

if len(topology.\_to\_list(input\_tensors)) != 1:

raise ValueError('To clone a `Sequential` model, we expect '

' at most one tensor '

'as part of `input\_tensors`.')

x = topology.\_to\_list(input\_tensors)[0]

if K.is\_keras\_tensor(x):

origin\_layer = x.\_keras\_history[0]

if isinstance(origin\_layer, topology.InputLayer):

return Sequential(layers=[origin\_layer] + layers, name=model.name)

else:

raise ValueError('Cannot clone a `Sequential` model on top '

'of a tensor that comes from a Keras layer '

'other than an `InputLayer`. '

'Use the functional API instead.')

input\_tensor = Input(tensor=x, name='input\_wrapper\_for\_' + str(x.name))

input\_layer = input\_tensor.\_keras\_history[0]

return Sequential(layers=[input\_layer] + layers, name=model.name)

def clone\_model(model, input\_tensors=None):

"""Clone any `Model` instance.

Model cloning is similar to calling a model on new inputs,

except that it creates new layers (and thus new weights) instead

of sharing the weights of the existing layers.

Arguments:

model: Instance of `Model`

(could be a functional model or a Sequential model).

input\_tensors: optional list of input tensors

to build the model upon. If not provided,

placeholders will be created.

Returns:

An instance of `Model` reproducing the behavior

of the original model, on top of new inputs tensors,

using newly instantiated weights.

Raises:

ValueError: in case of invalid `model` argument value.

"""

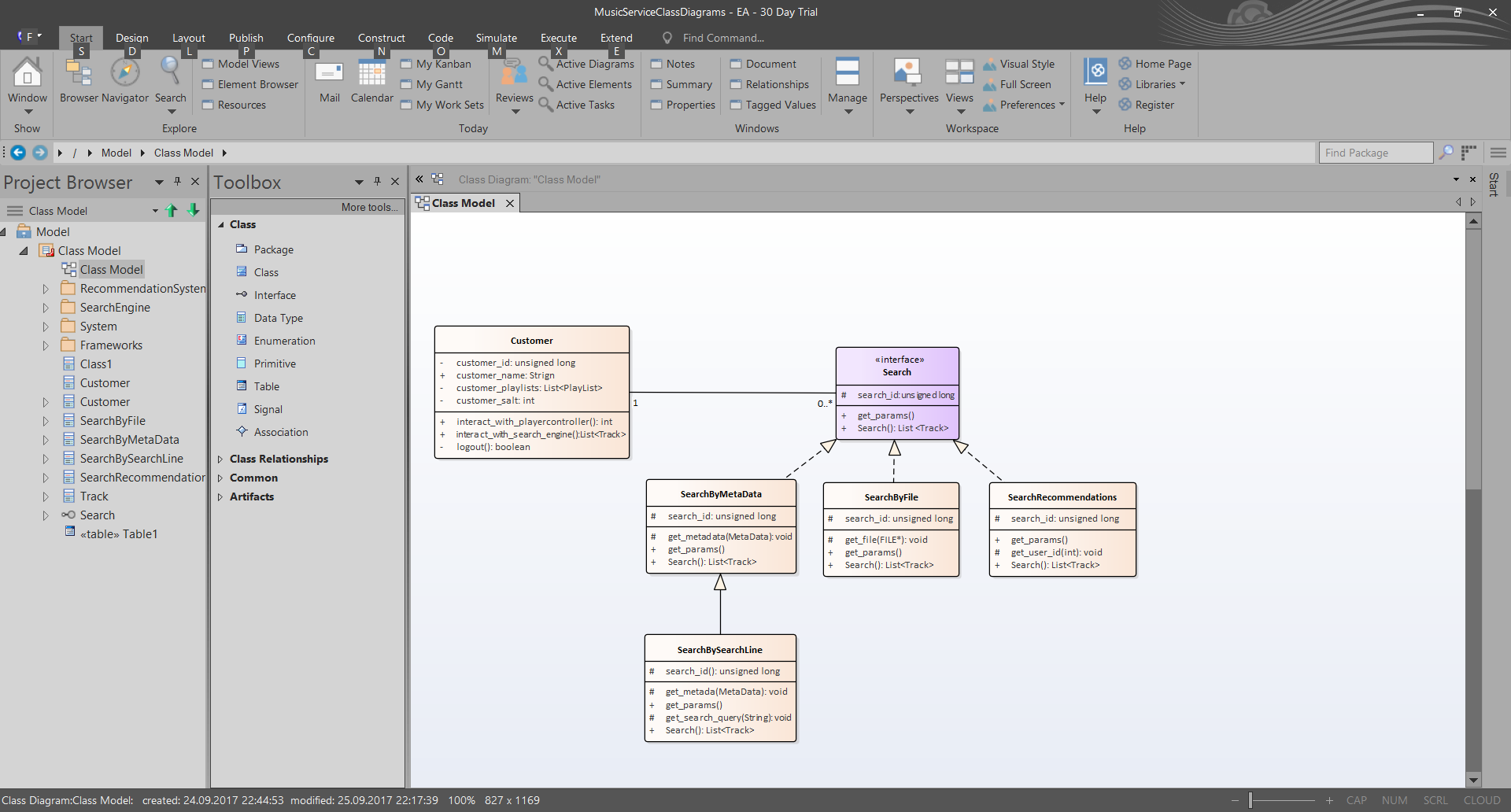
if isinstance(model, Sequential):

return \_clone\_sequential\_model(model, input\_tensors=input\_tensors)

else:

return \_clone\_functional\_model(model, input\_tensors=input\_tensors)

**Згеренована діаграма:**



**Пояснення відмінностей між діаграмами:**

Як бачимо діаграми майже не відрізняються. У згенеровай діаграмі є дві ключові відмінності від спроектованої.

Перш за все згенерована діаграмі містить набагато більше елементів, ніж початкова, оскільки система що генерувала діаграму не здатна відрізняти важливі елементи для даної діаграми, тому воно відображає усі елементи.

По-друге зникло стрілка асоціації з рекомендаційною системою, оскільки система що генерувала діаграму не здатна аналізувати програмний код і відображати виклики елементів з інших пакетів та модулів.

# ЗАПИТАННЯ ТА ВІДПОВІДІ

* + - 1. **Що таке пряме та зворотнє проектування?**

Пряме проектування (forward engineering) - це процес трансформації моделі в код з відображенням на мову реалізації. В результаті прямого проектування відбувається втрата інформація, оскільки моделі, описані на UML, семантично багатшими, ніж будь-який сучасний об'ектно-орієнтованний підхід в програмуванні.

Зворотне проектування (reverse engineering) - це процес трансформації коду в модель. Зворотне проектування породжує надлишок інформації, частина якої представлена на більш низькому рівні деталізації, ніж потрібно для побудови зручною моделі.

**2. У чому полягає суть зворотного проектування?**

Зворотне проектування полягає у тому, щоб спочатку писати програмний код — тобто, власне, реалізовувати систему, а лише потім генерувати її діаграму, отримаючи уявлення, як система побудована всередині, чи правильні були уявлення про архітектуру програмного забезпечення під час його реалізації, чи є якісь неточності.

3. **Як створити діаграму класів зворотнім проектуванням?**

Визначте правила для перетворення з обраного вами мови реалізації. Це можна зробити на рівні проекту або організації в цілому.

За допомогою інструментального засобу вкажіть код, який ви хочете піддати зворотному проектування. Скористайтеся цим засобом для створення нової моделі або для модифікації раніше створеної.

Користуючись інструментальними засобами, створіть діаграму класів шляхом опитування отриманої моделі. Можна почати, наприклад, з одного або декількох класів, а потім розширити діаграму, слідуючи уздовж деяких відносин або додавши сусідні класи.

**4. Для чого може знадобитись зворотне проектування?**

Зворотне проектування може бути корисним у багатьох випадках. По-перше, воно можно значно зекономити час на проектування, якщо система є невеликою. У такому випадку швидше відразу перейти до розробки, а потім подивитись на діаграму та зробити виправлення за потреби. Також таким чином можна перевірити свої навички проектування.

**4. Коли треба і не треба використовувати зворотнє проектування??**

Зворотне проектування може бути корисним у багатьох випадках. По-перше, воно можно значно зекономити час на проектування, якщо система є невеликою. У такому випадку швидше відразу перейти до розробки, а потім подивитись на діаграму та зробити виправлення за потреби. У випадку, коли система є великою та складною, краще навпаки витратити час на детальне проектування та розробку архітектури, адже шанси, що усю структуру можна тримати в голові, є досить невеликими.

**5. Які типові відмінності між створеною та сгеренованою діаграмою?**

Відмінності можуть полягати у будь-чому, особливо у деталях. Найчастіше відрізняються відношення між класами: їх наявність, види тощо. Також може бути різниця у типах полів та значень, що повертають методи, адже архітектор не завжди володіє повною інформацію про можливості обраної технології та мови програмування. Треба не забувати, що відмінності можуть бути спричинені не помилками, а особливостями засобів проектування.

# Висновок

Отже, я отримав навички зворотного проектування, навички генерації діаграм з програмного коду. Також я ознайомився з деякими помилками, що можна допустити про проектуванні діаграми та при написанні програмного коду за діаграмою.