Iterative Generation of Code Samples

Yu-Han Liu

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation
Liu, Yu-Han, "Iterative Generation of Code Samples", Technical Disclosure Commons, (October 12, 2020)
https://www.tdcommons.org/dpubs_series/3668

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.
Iterative Generation of Code Samples

ABSTRACT

Owing to an abundance of software libraries originating from a wide range of application programming interface (API) design patterns, developers who use the libraries need to spend substantial time and effort reading reference documentation and implementing experimental code before they can start writing and testing production code. This disclosure describes a framework to automate the creation, from software libraries, of high-quality, highly readable code samples with low maintenance and improvement cost.

KEYWORDS

● Software library
● Application programming interface (API)
● Iterative code generation
● Automatic code generation
● Compiler
● Language translator

BACKGROUND

Owing to an abundance of software libraries originating from a wide range of application programming interface (API) design patterns, developers who use the libraries need to spend substantial time and effort reading reference documentation and implementing experimental code before they can start writing and testing production code. The process of creating code samples from libraries often requires manual effort or is limiting:

1. Manually-written code samples are usually of higher quality and easier to learn from, but they are costly to create and keep up to date.
2. Common approaches to automatically create code samples rely on templates, which can support only a limited variety of code samples. Scaling up to more varieties results in complicated templates that are difficult to maintain or improve. Many code modification operations natural to human programmers cannot be expressed in templates.

3. Automatic programs that generate code only meet functional requirements; they are not structured, annotated, or commented for use as code samples, and as such can be practically unreadable by human programmers.

4. Compilers, e.g., language translators, have an iterative optimization passes design, but these often produce code that is unreadable by human programmers and not suitable to be used as code samples.

**DESCRIPTION**

This disclosure describes a framework to automate the creation, from software libraries, of high-quality, highly readable code samples with low maintenance and improvement cost. The techniques described herein reduce the time required to be spent by library users to read reference documentation and allow users to rapidly proceed to writing high-quality production code. Similar to compilers, the techniques described herein have iterative optimization passes, each pass performing flexible operations similar to human programmers such that code quality improves over successive iterations.

The described framework can be implemented as a tool that accepts as input a software library and configurations set by a programmer, and produces as output code samples that call functions in the library. The module iteratively improves the code sample.
Fig. 1: Automatic generation and iterative improvement of code samples

Fig. 1 illustrates a framework for automatic generation and iterative improvement of code samples, per the techniques of this disclosure. The framework treats the process of generating code samples as an iterative (or stage-wise) process starting from a minimal and potentially low-quality initial sample and applying a sequence of optimization passes (code modifications), as opposed to a single step filling of a template. Example passes are labeled in the figure as first_pass, second_pass, etc. Individual passes can be implemented using tools for parsing code into syntax trees. Code samples are generated and iteratively improved based on the signature, e.g., set of input/output parameters, their types, overloading, default values, etc. of functions and methods, class declarations, etc., as found in imported software libraries; readable display of output values of functions; etc.

Each pass has the quality of expressiveness, e.g., the capability of inspecting the code sample generated up to right before its application, enabling the pass to perform flexible operations similar to human programmers. The sequence of passes can be expressed in a high-level abstraction and can be modified or replayed as necessary to regenerate code samples.

Example 1

In the code snippets below, example_module.py is a module that declares two functions, compute_area and compute_diagonal.
Fig. 2: An example code module that declares two functions available in a library

In Fig. 2, the code snippet `compute_area_sample` computes the area of a rectangle by including `example_module.py` as well as possibly other libraries such as `numpy`, `random`, and (through `example_module.py`) `math`. Figs. 3(a)-(i) illustrate successive improvements in the code quality and readability of `compute_area_sample` obtained by applying the techniques of this disclosure.

0. Init

(a)

(b)

Fig. 3(a): An (zeroth-pass) initial code sample of low quality; (b) A legend, indicating that the entire code of Fig. 3(a) is an initial change
Fig. 3(c): A first-pass improvement to the parameter handler. Based on configurations set by the user, the red highlight indicates code that can be deleted, and the green highlight indicates code that can be added. The code on the left panel is from Fig. 3(a), while the code in the right panel indicates code at the end of the first pass.

Fig. 3(d): A second-pass improvement to the inline helper-function. The code on the left panel is from the right panel of Fig. 3(c), while the code in the right panel indicates code at the end of the second pass.

Fig. 3(e): A third pass removes a dummy assignment statement
4. AssignInnerFuncCallReturnValue

```python
from example_module import Geometry
from example_module import Rectangle
from numpy import random

def compute_area_sample(width: float, height: float):
    geometry_client = Geometry()
    result = geometry_client.compute_area(Rectangle(width=width, height=height))
```

Fig. 3(f): A fourth pass assigns a value to the return value of a called function

5. ProcessResponse

```python
from example_module import Geometry
from example_module import Rectangle
from numpy import random

def compute_area_sample(width: float, height: float):
    geometry_client = Geometry()
    result = geometry_client.compute_area(Rectangle(width=width, height=height))
    print(f"Result = {result}")
```

Fig. 3(g): A fifth pass creates a function to process the result

6. InlineHelperFunc

```python
from example_module import Geometry
from example_module import Rectangle
from numpy import random

def compute_area_sample(width: float, height: float):
    geometry_client = Geometry()
    result = geometry_client.compute_area(Rectangle(width=width, height=height))
    print(f"Area = {result}")
```

Fig. 3(h): A sixth pass improves an inline helper function

7. RemoveUnusedImports

```python
from example_module import Geometry
from example_module import Rectangle
from numpy import random)

def compute_area_sample(width: float, height: float):
    geometry_client = Geometry()
    result = geometry_client.compute_area(Rectangle(width=width, height=height))
    print(f"Area = {result}")
```

Fig. 3(i): A seventh pass removes an unused import command
The iterative improvement in code quality and readability is seen by comparing the initial sample of Fig. 3(a) with each incremental stage, to the final code of Fig. 3(i).

**Example 2**

The code snippet `compute_area_height_two_sample` computes the area of a rectangle with a default height set to 2. Through the use of the configuration mechanisms provided by the framework, the programmer can fine tune the shape of the outcome sample code and potentially create multiple different sample variants, e.g., `compute_area_sample` and `compute_area_height_two_sample`, that call the same library method, `compute_area`, with different arguments. Figs. 4(a)-(i) illustrate the successive improvements in the code quality and readability of `compute_area_height_two_sample` obtained by the techniques of this disclosure.

![0. Init]

**Fig. 4(a)**: An (zeroth-pass) initial code sample of low quality; **(b)** A legend, indicating that the entire code of Fig. 4(a) is an initial change
1. ParamHandler

```python
from example_module import Geometry
from example_module import Rectangle
from numpy import import random

def compute_area_height_two_sample(rectangle: Rectangle):
    geometry_client = Geometry()

gometry_client.compute_area(rectangle=rectangle)
```

Fig. 4(c): A first-pass improvement to the parameter handler. Based on configurations set by the user, the red highlight indicates code that can be deleted, and the green highlight indicates code that can be added. The code on the left panel is from Fig. 4(a), while the code in the right panel indicates code at the end of the first pass.

2. InlineHelperFunc

```python
from example_module import Geometry
from example_module import Rectangle
from numpy import import random

def compute_area_height_two_sample(width: float):
    geometry_client = Geometry()

    rectangle = Rectangle(width=width, height=2.0)
    return rectangle

    rectangle = make_rectangle(width=width)
    geometry_client.compute_area(rectangle=rectangle)
```

Fig. 4(d): A second-pass improvement to the inline helper-function. The code on the left panel is from the right panel of Fig. 4(c), while the code in the right panel indicates code at the end of the second pass.

3. RemoveDummyAssign

```python
from example_module import Geometry
from example_module import Rectangle
from numpy import import random

def compute_area_height_two_sample(width: float):
    geometry_client = Geometry()

    rectangle = Rectangle(width=width, height=2.0)
    return rectangle

    rectangle = make_rectangle(width=width)
    geometry_client.compute_area(rectangle=rectangle)
```

Fig. 4(e): A third pass removes a dummy assignment statement
4. AssignInnerFuncCallReturnValue

Fig. 4(f): A fourth pass assigns a value to the return value of a called function

5. ProcessResponse

Fig. 4(g): A fifth pass creates a function to process the result

6. InlineHelperFunc

Fig. 4(h): A sixth pass improves an inline helper function

7. RemoveUnusedImports

Fig. 4(i): A seventh pass removes an unused import command
In this manner, the described framework for automatic code generation from libraries encodes and simulates behaviors of human programmers. Some advantages of the described framework for code generation include:

- **Ease of development**: The individual passes can be implemented, developed, tested, and debugged in isolation.

- **Ease of reasoning about**: The programmer can attribute a line of code in the generated sample to a particular pass.

CONCLUSION

This disclosure describes a framework to automate the creation, from software libraries, of high-quality, highly readable code samples with low maintenance and improvement cost.

REFERENCES

