Project Final Report: Multi-task Method Generation

Method Generation Group
Changshu Liu (cl4062), Zhangyi Pan (zp2223), Huiyan Zhang (hz2757)

1. Synopsis

a. Overview

Code completion has become one of the most important functions in Integrated Development Environments (IDEs), which can speed up the developing process by providing auto-completion popups, querying parameters of functions, and providing relative hints. Lots of studies have been conducted on it in order to provide programmers with better tools to code conveniently. Recently, researchers have proposed a new direction to help programmers or even non-technical people write code quickly. It is method generation. More specifically, given a description of a method in words which is called docstring in the rest of the paper, and a signature that contains the function name and the input name, researchers believe that there are models which can predict the code of the entire method successfully.

One of the most popular related studies is CodeXGLUE, which inspired our experiment. In the method generation task of CodeXGLUE [1], it concatenated signature and docstring together as a new input. With the help of CodeGPT, the method was able to predict the code of the method body. We argued that a concatenation might not fully exploit the connection between signature and docstring. In this study, we proposed a new multi-task learning framework, containing a common encoder and two different decoders. The first decoder was used to generate the summarization of the docstring and the second decoder could come up with the code of the method body. We tested our newly proposed method and compared it with several baseline models. We also conducted parameter analysis and ablation study to better understand and improve our model.

b. Novelty

Method generation is still a new and hot topic. Most related research was conducted after 2020. And the multi-learning encoder-decoder framework we proposed has never been utilized for method generation tasks before.

c. Value to Our User Community

The development of method generation is able to provide lots of benefits. For software engineers, more accurate method generation tools will release them from heavy coding implementation tasks. Instead of spending time on writing easy or repetitive functions, they can focus more on high-level design and tricky function implementations. When it comes to people who don't know much about coding, method generation becomes even more useful. After coming up with a design of a method, they can get the code by simply typing in their description and method name. This will be extremely helpful for product managers, UI designers, and other non-technical people who work closely with codes. However, even though method generation can bring tons of advantages, the current version of it is not accurate enough for a wide range of use in the real world. We wish what we are doing can propose a possible solution to improve the current solution. And hopefully, method generation can become accurate enough to be used in the industry one day.

2. Related Work

a. CodeXGLUE

CodeXGLUE [1] is a collection of tools developed by Microsoft to assist people in tackling programming problems. It includes tools for clone detection, defect detection, code completion, code translation, and so on. CodeXGLUE is inspired by the recent progress made in NLP and Computer Vision and has taken advantage of those state-of-art techniques to improve the intelligent code assistance system. Among those techniques, the one which arouses our interest is the method generation task. CodeXGLUE uses a model called CodeGPT to finish the code completion task. As for the method generation task, it finetunes the CodeGPT model.

b. CodeGPT

CodeGPT [1] is a code completion and code generation model built upon GPT-2. The model is a language-based pre-trained transformer with 12 layers of decoders. As shown in figure 1, in the training stage, the objective of a GPT method is to predict the current term based on the prediction result from previous terms. In our case, we tokenize the code, and thus our prediction should be tokens of code. In the inference stage, we feed the GPT with tokenized docstring and signature, and then we let the model prompt out the method body token by token.

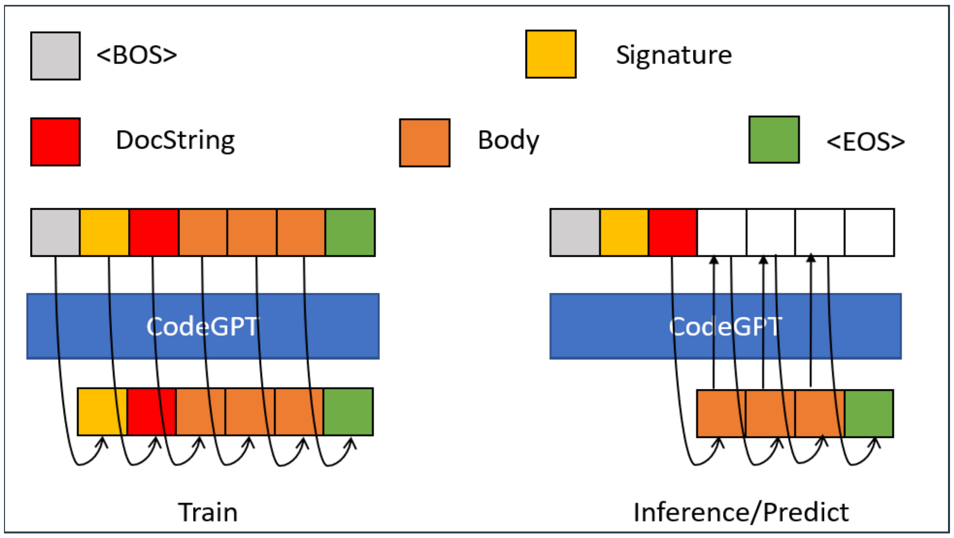

Figure 1. Training Stage and Inference Stage of CodeGPT

c. Encoder-Decoder

The encode-Decoder framework is a popular framework widely used in machine learning models. The encoder takes some sequential input and translates it into another representation. In most cases, the new representation is a more compact one. Then, the decoder takes the new representation to generate an output which has a similar form compared to the original input. In CodeXGLUE, the input code is passed to CodeBert as an encoder. After that, it uses multiple decode layers to generate the output code.

d. Multi-task Learning

In Multi-task learning, we conduct learning on several tasks independently at the same time and examine their interrelationships and commonalities afterwards. Multi-task Learning helps by improving the efficiency and accuracy, as we can take advantage of the similarities between distinct tasks and let them make improvements on each other. In our work, we learn and take advantage of MTFuzz[2], a multi-task learning model targeting to detect software bugs.

3. Proposed Methodology

a. Methodology Overview

An interesting observation we had when we went through CodeXGLUE was about its input of signature and docstring. The CodeXGLUE treated the signature and the docstring as two distinct attributes. Based on our observation, we conducted a small test. Using the Transformer Encoder-Decoder model (initialized with CodeBert) and the dataset we pre-processed for our real evaluation (described below), we trained 30 epochs with different input information. When the input was the docstring which described the method in words, we got a 19.03 BLEU-4 score. If we concatenated the signature and the docstring together as our input, we got a 19.42 BLEU-4 score. However, we believe that concatenation may not be the optimal way to utilize the information. We proposed that generating a succinct description of the docstring might help the encoder to achieve a better understanding of the sequence. Therefore, we came up with the usage of Multitask Learning, with the first task to generalize the docstring and the second task to generate the method body.

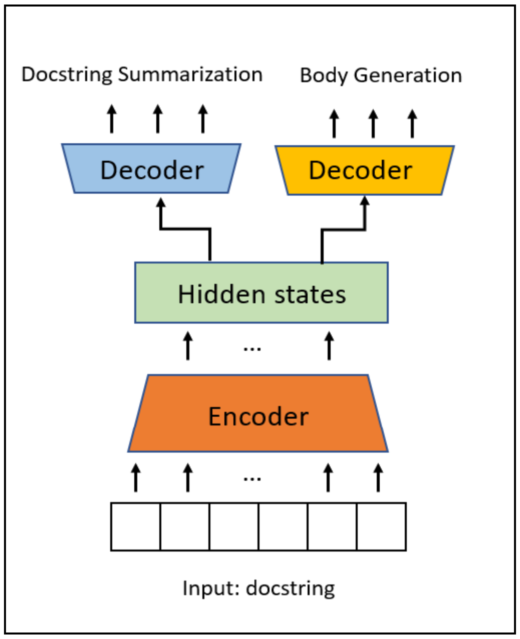

Figure 2. Structure of the proposed methodology

b. Connection between Signature and Docstring

A key hypothesis of our project was that we could extract some meaningful information from the signature, which might serve as the summarization of docstring. Suppose we have a pair of signature and docstring as follows:

```
"signature": "def kill_proc_tree"
"docstring": "Kill a process tree (including grandchildren) with signal\n    \"sig\" and
return a (gone, still_alive) tuple.\n    \"on_terminate\", if specified, is a callback
```

```
function which is\n    called as soon as a child terminates."
```

From the signature, we could extract “kill proc tree” via pattern matching. We argued that “kill proc tree” could be the key words/summarization of the long docstring. Suppose we use an Encoder-Decoder to summarize a source sequence into a target sequence, the role of the encoder is to have a correct understanding of the source sequence and the decoder is expected to generate correct outputs. Another important hypothesis in our project was that the task of docstring summarization might help the encoder have a better understanding of the input sequence, which may be beneficial for the task of using docstring to generate the method body.

c. Multi-task Learning

In our proposed method, we incorporated the idea of Multi-task Learning [2]. In our framework, we trained the neural network on two tasks:

Task1: Docstring summarization: using the docstring as the input, expect the decoder to generate the information extracted from the corresponding signature.

Task 2: Method name generation: using the docstring as the input, expect the decoder to generate the ground-truth method body given by the training data.

As shown in figure 2, we had an encoder to encode the docstrings, and also independent decoders for both tasks. During the training process, since we had multi-tasks, we would need an updated loss function. We defined our loss function as below :

$$loss = loss_{task2} + \alpha\, loss_{task1}$$

In the inference stage, we fed the encoder with docstring and used the decoder of task 2 to generate the body of the method.

d. Architecture

Following the settings of CodeXGLUE, we initialized the common encoder with CodeBert[5]. The two independent decoders were randomly initialized, each of them had a Transformer with 6 layers and 12 attention heads.

Suppose we convert the summarization extracted from the signature into a sequence a tokens, then we can further convert it into a sequence of indexes : $Y = \{y_1, y_2, ..., y_M\}$. The first decoder will output a sequence of probabilities : $P = \{p_1, p_2, ... p_M\}$, where p_i is the probability of the the index of the i-th token generated is exactly y_i. Therefore, the loss function of task 1 can be formulated as:

$$loss_{task1} = crossEntropy(Y, P)$$

Similarly, for task2, for every sample we can derive its corresponding index sequence $T = \{t_1, t_2, ..., t_N\}$. With the output of the second decoder $G = \{g_1, g_2, ..., g_N\}$, we can formulate the loss function of task2:

$$loss_{task2} = crossEntropy(T, G)$$

4. Research Questions

To better evaluate and understand our newly proposed method, we came up with the following research questions:

- **Research Question 1: Performance.** How does our model perform in comparison with other baselines?

- **Research Question 2: Contributions of the Auxiliary Tasks.** How much does the auxiliary task contribute to the overall performance of our proposed model?
- **Research Question 3: Impact of Pretrained Weights.** How do pretrained weights affect the performance of our model?

a. Dataset

We used the same dataset as the method generation task of CodeXGLUE used. It was derived from CodeSearchNet Python dataset [4] with real-world code generation tasks for Python. Methods that didn’t contain docstring and whose function name had the word test were removed. Literal normalization was applied to signatures and bodies, which replaced \n with <EOL> and kept track of INDENT and DEDENT. Due to the restrictions of our training and testing environment, we narrowed down the dataset to 1/40. All instances were selected randomly. The final dataset had around 22,500 training instances, 500 testing instances, and 500 dev instances.

```
{
   "signature": "def do_transform(self, v=<NUM_LIT:1>):",
   "body": "if not
self.transform:<EOL><INDENT>return<EOL><DEDENT>try:<EOL><INDENT>self.l
atest_value = utils.Transform ...",
   "docstring": "Apply the transformation (if it exists) to the latest_value",
   "id": "f19:c4:m1"
}
```

Figure 3. Dataset sample [3]

b. Research Question 1: Performance. How does our model perform in comparison with other baselines?

We compared our model with 1) CodeGPT 2) the Transformer Encoder-Decoder model. We trained each model for 30 epochs with the same training set and then tested it using the same test set. We used Bleu-4 to evaluate the performances of models. Due to the limitation of the computational resources we could get access to, when training CodeGPT we adopted a small batch size 2, as a larger batch size would cause an out-of-memory error. When training our proposed model and Transformer Encoder-Decoder model, we used a batch size of 16. The experiment results are shown in Table 1. We have the following findings:

1) GPT-based models will always need more space and time during training.
2) Our proposed model outperforms others, which indicates that our multitask learning strategy is better than simply concatenating docstring and signature together.

Model	**Bleu-4**	**Training Time**
CodeGPT	17.82	>84 h
Transformer Encoder-Decoder	19.42	>10 h
Our Model	21.30	>10 h

Table 1. Performance of different models

c. Research Question 2: Contributions of the Auxiliary Tasks. How much does the auxiliary task contribute to the overall performance of our proposed model?

In this question, we would like to know how much the task of generating the summarization of docstring contributes to the overall performance of our model. To answer this question, we set α to different values (from 0 to 1). Note that when α was set to 0, our model would degenerate into a standard Encoder-Decoder model which only took the docstring as the input. We treated this task as a part of our ablation study. Our experiment settings were similar to RQ1 and we tested our model on the test set every 10 epochs. Our experiment results are shown in Figure 4 and we have the following findings:

1) When α is set to 0.2 or 0.3, our model will have better performances.
2) In most cases, our model can outperform the special variant (α=0), which suggests that our auxiliary task can improve our model's performance on the task of generating the method body.
3) However, when α is set to a larger value like 0.9 or 1, our model will have poor performance compared with the special variant. We speculate that task 1 docstring summarization is much easier than task 2. That is because in task 1 the target sequence (docstring summarization) is much shorter than the target sequence of task 2 and it may contain words that have appeared in the source sequence (docstring). In contrast, task 2 requires the model to generate PL tokens that can not be found in the source sequence. Therefore, when we give task 1 a larger weight, the model will pay more attention to docstring summarization instead of the method body generation.

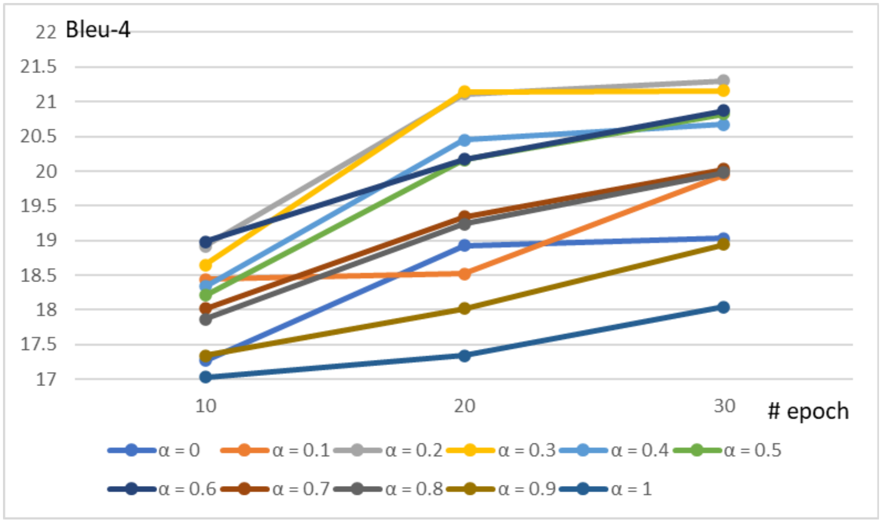

Figure 4. Parameter Analysis and Ablation Study

d. Research Question 3: Impact of Pretrained Weights. How do pretrained weights affect the performance of our model?

In this section, we investigated the impact of pre-trained weights (CodeBert). We repeated most parts of the experiment in RQ2. The only difference was that models were trained from scratch in RQ3. We report the performances of models after training for 30 epochs in Figure 5 and we have the following findings:

1) Finetuning on pre-trained weights can improve the performance of our model. For example, when we set α to 0.1, the result of the training from scratch is 17.87 while the result of finetuning on CodeBert is 19.95.
2) Surprisingly, we find out that when models are trained from scratch, our auxiliary task improves the performance on method generation by a considerable margin (around 5 on Bleu-4). This further proves the contribution of our multi-task learning setting.

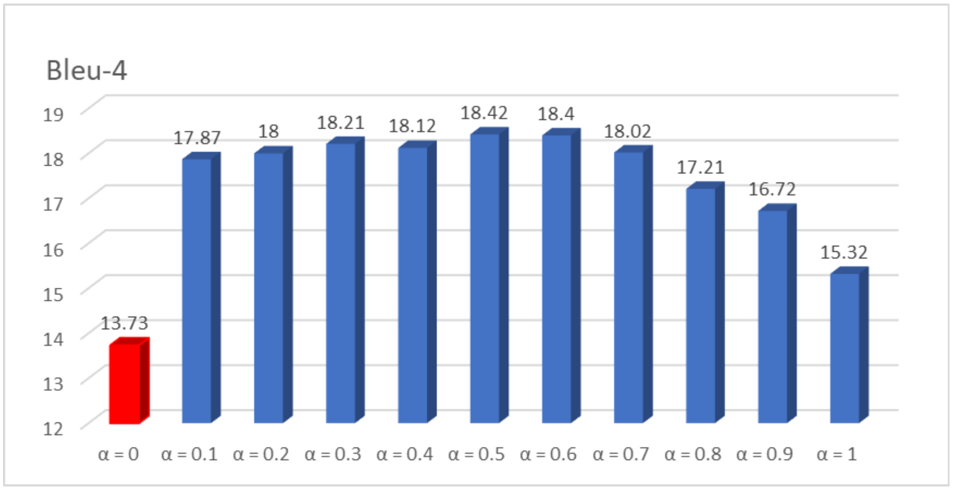

Figure 5. Experiment Results When Trained from Scratch

5. Deliverables

a. Github Repository

All materials related to this project are organized in the following repository: https://github.com/Huiyan0409/6156FinalProject.

b. Reproduction

To reproduce our work, we require at least one GPU. Detailed instructions of training and testing our model can be found in the ReadMe file in the above repository. To check the result of our trained model without any training, please go to this link https://drive.google.com/drive/folders/1w9C-Hc5XNbWQXIZko0JTIEDzOOsEsWc6?usp=sharing to download checkpoints and follow the instructions in our repository to load them.

6. Future Work

Because of the restrictions on training and testing environments, our research was based on the mini dataset. To evaluate our model more accurately, we should conduct further research on the whole dataset.

Moreover, the signature contains more useful information such as the variable type and the variable name. In our future work, we should consider how to incorporate them to further improve our model.

7. Self-Evaluation

a. Huiyan Zhang

In this project, I obtained the original dataset from CodeXGLUE and narrowed it down to a reasonable size for our training and testing. I also took charge of the ablation study to train and test the standard Encoder-Decoder model as well as the newly proposed model with different α. And we worked together on the demo presentation and the project report.

During the process, I encountered some difficulties when I tried to reproduce the Code-to-Text task in CodeXGLUE and transform it into our standard Encoder-Decoder model. The provided command line to run their code on an online notebook caused some misunderstandings for me. And it failed to state the required training environment. I realized the importance of writing an easy-to-understand and complete ReadMe file for people who were interested in learning the project. Also, I was able to gain lots of hands-on experience in reproducing other people's work and solving related bugs. Moreover, I learned a lot about the concepts and methodologies behind frequently used models in code generation. I noticed that, instead of only focusing on finetuning machine learning models to get better results, we should always pay attention to the methodologies and think about whether there are better architectures that can produce better results.

Last but not least, I personally think I have learned a lot from this class. But it would be better if we can learn a little bit more about how to give a great presentation, how to conduct good research, and how to write an excellent research paper in class.

b. Zhangyi Pan

In this project, I conducted the experiments for the baseline method CodeGPT on our mini-dataset. I also implemented the helper function to extract information from the signature of the input functions.

During this process, I spent a lot of time addressing the dependency issues with CodeGPT, and I had to read through some parts of the code to learn how to conduct proper tests for its method generation task. This invokes me to provide more detailed instructions in the README file on the dependencies and how to run the code. By researching CodeGPT and our proposed method, I gained a more thorough understanding about method generation. In our model, I also learned about multi-task learning, which is a relatively new topic to me.

c. Changshu Liu

In this project, I designed the whole framework of our multi-task method generation model. Referring to the related projects of CodeXGLUE, I built the model using Pytorch and Transformers (Huggingsface) and trained it using the Colab Tesla P100 GPU and 1080Ti GPU. After that I conducted parameter analysis to further investigate the contribution of our proposed auxiliary task.

In this project I learnt more about how to use language models to solve software engineering problems. By implementing the neural network we designed, I greatly improved my coding skills.

References

1. Lu, Shuai and Guo, Daya and Ren, Shuo and Huang, Junjie and Svyatkovskiy, Alexey and Blanco, Ambrosio and Clement, Colin and Drain, Dawn and Jiang, Daxin and Tang, Duyu and Li, Ge and Zhou, Lidong and Shou, Linjun and Zhou, Long and Tufano, Michele and Gong, Ming and Zhou, Ming and Duan, Nan and Sundaresan, Neel and Deng, Shao Kun and Fu, Shengyu and Liu, Shujie. ‘CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation’. 2021. https://arxiv.org/abs/2102.04664.

2. Dongdong She and Rahul Krishna and Lu Yan and Suman Jana and Baishakhi Ray. ‘MTFuzz: fuzzing with a multi-task neural network’. *Proceedings of the* 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2020. https://arxiv.org/abs/2005.12392.

3. https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Method-Generation.

4. Clement, Colin B and Lu, Shuai and Liu, Xiaoyu and Tufano, Michele and Drain, Dawn and Duan, Nan and Sundaresan, Neel and Svyatkovskiy, Alexey. ‘Long-Range Modeling of Source Code Files with eWASH: Extended Window Access by Syntax Hierarchy’. 2021. https://arxiv.org/abs/2109.08780.

5. Feng, Zhangyin, et al. "Codebert: A pre-trained model for programming and natural languages." Findings of EMNLP 2020, 2020. https://arxiv.org/abs/2002.08155

APPENDIX

Gold (expected output)	Output
return self.__tag<EOL>	return self.__tag<EOL>
if hasattr(v, "<STR_LIT>"):<EOL><INDENT>v = v._utype(v)<EOL><DEDENT>try:<EOL><INDENT>t = YANGDynClass(<EOL>v,<EOL>base=config.config,<EOL>is_container="<STR_LIT>",<EOL>yang_name="<STR_LIT>",<EOL>parent=self,<EOL>path_helper=self._path_helper,<EOL>extmethods=self._extmethods,<EOL>register_paths=True,<EOL>extensions=None,<EOL>namespace="<STR_LIT>",<EOL>defining_module="<STR_LIT>",<EOL>yang_type="<STR_LIT>",<EOL>is_config=True,<EOL>)<EOL><DEDENT>except (TypeError, ValueError):<EOL><INDENT>raise ValueError(<EOL>{<EOL>"<STR_LIT>": """<STR_LIT>""",<EOL>"<STR_LIT>": "<STR_LIT>",<EOL>"<STR_LIT>": """<STR_LIT>""",<EOL>}<EOL>)<EOL><DEDENT>self.__config = t<EOL>if hasattr(self, "<STR_LIT>"):<EOL><INDENT>self._set()<EOL><DEDENT>	if hasattr(v, "<STR_LIT>"):<EOL><INDENT>v = v._utype(v)<EOL><DEDENT>try:<EOL><INDENT>t = YANGDynClass(v,base=config.config, is_container='<STR_LIT>', presence=False, yang_name="<STR_LIT>", rest_name="<STR_LIT>", parent=self, path_helper=self._path_helper, extmethods=self._extmethods,
return self.__description<EOL>	return self.__description<EOL>
raw_result = self._prepare_network_policies_initial(<EOL>resource_group_name=resource_group_name,<EOL>virtual_network_name=virtual_network_name,<EOL>subnet_name=subnet_name,<EOL>prepare_network_policies_request_parameters=prepare_network_policies_request_parameters,<EOL>custom_headers=custom_headers,<EOL>raw=True,<EOL>**operation_config<EOL>)<EOL>def get_long_running_output(response):<EOL><INDENT>if raw:<EOL><INDENT>client_raw_response = ClientRawResponse(None, response)<EOL>return client_raw_response<EOL><DEDENT><DEDENT>lro_delay = operation_config.get(<EOL>'<STR_LIT>',<EOL>self.config.long_running_operation_timeout)<EOL>if polling is True: polling_method = ARMPolling(lro_delay,	raw_result = self._create_initial(<EOL>resource_group_name=resource_group_name,<EOL>virtual_network_network_name=virtual_network_network_name,<EOL>virtual_network_ip_parameters=virtual_network_parameters,<EOL>custom_headers=custom_headers,<EOL>raw=True,<EOL>**operation_config<EOL>)<EOL>def get_long_running_output(response):<EOL><INDENT>deserialized = self._

**operation_config)<EOL>elif polling is False: polling_method = NoPolling()<EOL>else: polling_method = polling<EOL>return LROPoller(self._client, raw_result, get_long_running_output, polling_method)<EOL>	
return self.legendPatch<EOL>	return self._rectangles<EOL>