# Convolutional Neural Network programs modeling for automatic detection of design smells

Anonymous Author(s)

***Abstract*—TODO: To be completed**

***Keywords*—** Deep Learning, Convolutional Neural Network, Design Smells, Famix, Modeling, Software Engineering

## I. Introduction

Deep learning is a subset of machine learning that uses artificial neural networks to learn from unstructured data. Deep learning has become very popular in recent years due to its performance in several fields, such as image recognition, speech recognition, machine translation, and more. Artificial neural networks are mathematical models inspired by the workings of the human brain. They are composed of several layers of interconnected neurons. Each neuron has an activation function that calculates the neuron's output according to its inputs. There are several types of artificial neural network architecture. These architectures have been developed to address specific problems, such as convolution architecture for image recognition, recurrent neural networks for speech recognition, and so on. In this paper, we focus on convolution architecture, as it can handle a wide range of problems, such as image recognition, text recognition, and sequence recognition. Moreover, convolution architecture is part of feed-forward architectures, a class of artificial neural networks composed of several layers of neurons and containing no cycles. Feed-forward neural networks are the most widely used in deep learning.

### A. Convolutional Neural Network

In this paper, we will refer to deep learning programs as those containing convolutional neural networks (CNN). Convolutional Neural Network consists of the following three main stages:

*Convolution:* Convolution is the action of passing filters over an image to extract patterns. A filter (kernel) is a classical neural network that scans an image part by part to detect a pattern. The filter scans a window (part) of the image and then calculates the output in one pixel. The filter then browses the entire image to calculate the output of each pixel in a new image. The new image is called a feature map. Convolution increases the depth of the input image by generating multiple feature maps while reducing its size. Recall that each depth layer is an image (reduced in size) generated by a filter.

*Pooling:* Pooling is an operation that reduces image size by selecting the largest pixel in a window. This reduces computation and memory consumption. It also reduces Overfitting on training data, as information on the position of detected patterns is lost. So we're going to use pooling on the feature maps generated by convolution. Convolution is followed by pooling, and this operation is repeated several times. The deeper we go in the convolution, the more precise patterns are detected, likely to facilitate classification.

*Flatten:* The flatten operation transforms the feature maps generated by convolution and pooling into a single vector. This vector is then used as the input to a classical neural network (*dense layer*). The output of the convolutional neural network will come from this neuron.

### B. Design smell

A code smell is a poor design and implementation choice that can have a negative impact on software quality [1]. Suryanarayana et al. define design smells like structures in the design that suggest a violation of the primary design principles, which might negatively affect design quality [2]. Just like any other program, deep learning programs can contain code smells. In this paper, we focus on design smells because they are introduced early in the software development cycle and can have a significant negative impact on software performance and quality. These smells are introduced by the developer during the software design phase.
Nikanjam et al. proposed a list of 8 design smells for feed-forward learning programs [3]. They found these smells in literature reviews and the open-source platforms Github and StackOverflow. Their empirical study showed that the proposed smells are perceived as relevant by developers. We have therefore used this list of smells to propose an automatic smell detection system for developers.
The 8 proposed smells are classified into two main categories as follows:

1) **Design smells during convolution and pooling**: These smells are introduced when feature maps are formed. They are linked to the size of the feature map or filter and the layer layout. The list of smells is presented in the table I below.
2) **Design smells linked to the use of regularization methods**: These smells are introduced when using regularization. They are linked to the use of regularization methods such as the dropout layer and the layout of these methods. The list of smells is presented in the table II below.

### C. Modeling

Thomas Kühne defines a model as an artifact formulated in a modeling language, such as UML, that describes a system using different types of diagrams [4]. This abstract representation of a system is defined by another model called a metamodel. A model allows a system to be analyzed, understood, and transformed efficiently. Its operations can be automated [5]. There are several types of models, such as behavioral models (representing the dynamic behavior of the system) or design models (representing the static structure of the system).
In this paper, we are interested in design models because the smells studied are represented in static form and are introduced at the system structure level.

We'll also be using ***FAST*** (*Famix Abstract Syntax Tree*) models to represent the systems studied in this paper. The *FAST* models are syntax trees derived from the Pharo object-oriented programming language [6], [7], [8]. It inherits the ***Famix*** model, which is defined by Tichelaar et al. as a language-independent representation of object-oriented source code. It is an entity-relationship model that models object-oriented source code at the program entity level [9] [10]. The *FAST* models are developed in the Moose reverse engineering environment [11].

Table I: *Design smells in convolution and pooling layers*

| Design Smell | Description |
|---|---|
| Non-expanding feature map | Keep the same number of features or reduce it as the architecture becomes deeper. |
| Losing local correlation | Start with a relatively large window size for spatial filtering and maintain it for all convolutional layers. |
| Heterogeneous blocks of CNNs | Build a deeper model by stacking only a set of convolution and pooling layers without appropriate configuration. |
| Too much down-sampling | Pooling right after each convolutional layer, especially for the first layers. |
| Non-dominating down-sampling | Use of average-pooling. |

Table II: *Design smells related to regularization*

| Design Smell | Description |
|---|---|
| Useless Dropout | Using Dropout before Pooling Layers. |
| Bias with Batchnorm | Keep the bias values in the layers when using Batchnorm. FNN learning layers are biased with different initializations. |
| Non-representative Statistics Estimation | Use of Batchnorm after Dropout. |

### *D. Motivation and research questions*

In the Nikanjam et al. study, developers identified not only the proposed design smells but also their relevance and hence the value of avoiding them. However, there is no automatic smell detection system in the literature. In this paper, we propose such a system, which will enable smell detection in CNN deep learning programs. Moreover, our system will cover the most widely used open-source Frameworks on the market. To this aim, we have defined the following research questions:

*RQ1: Is it possible to detect design smells in CNN deep learning programs with modeling?*

*RQ2: What are the most prevalent design smells in CNN deep learning programs?*

*RQ3: Are there any correlation between the program design smells?*

Our paper is organized as follows: section II presents past work related to our topic. Section III describes the data collection and processing. Section IV presents the results. Section V discusses the results presented above. Section VI presents threats to validity and section VII presents the limitations of our study and future work.

## II. Background

In this section, we present previous work on detecting code smells in software.

The idea of detecting smells in software source code is a much-studied topic in the literature. Indeed, several works have been proposing detection techniques for more than a decade. Code smells have been studied from several angles, such as their presence in software developed in a specific programming language [12], in the security of Infrastructure as Code source code [13], in microservices [14], or their impact on the performance of Android applications [15]. Fard et al [16] present JSNOSE (precision: 93% and average recall: 98%) a technique for detecting code smells in the JavaScript language. A combination of static and dynamic analysis to detect client-side code smells. Chen et al. [17] propose Pysmell, a tool capable of detecting Python code smells with an average precision of 97.7%. Code smell detection has an impact on software quality, with Van Emden et al. [18] doing Java code quality assurance using code smell detection and visualization. To cite just one example, Moha et al. [19] propose a code smell detection and specification approach called DECOR with a precision of 60.5%, and a recall of 100%. There are also modeling-based approaches to smell detection in the literature, such as the approach proposed by Araujo et al. [20] to identify organizational smells in multi-agent systems via metamodeling, or that of Khomh et al. [21] who convert detection rules in the literature into a probabilistic model to detect code smells. The latter leads the way to the detection of code smells with the use of artificial intelligence. In recent years, several works have proposed artificial intelligence-based approaches to code smell detection [22], [23], [24]. In light of these works, we asked ourselves the following questions: **Is it possible to efficiently and statically detect code smells in deep learning software?**
In the literature, Nikanjam et al. have initiated the answer to these questions by proposing NeuraLint (precision: 100% and recall: 70,5%) [25], a model-based approach to code smell detection. This approach uses modeling to achieve its goal. However, NeuraLint's scope was limited to the feed-forward multilayer perceptron (MLP) architectures. Deep learning is a growing field, and neural network architectures are becoming increasingly complex. It is therefore necessary to be able to detect code smells from other neural network architectures. The CNN architecture being one of the most widely used in practice, we aim in this paper to allow the detection of code smells in CNN-type neural networks.

## III. Study Design

In this sub-section, we describe the details of the data collection and processing approach followed to answer our different research questions.

### *A. Data Collection*

The development of the design smell detection system was based on code samples of deep learning programs. These examples represent deep learning programs in which design smells are introduced. These samples were collected by Nikanjam et al. [3] as part of their empirical study of design smells in deep learning programs. Indeed, in this study, they presented samples of deep learning programs containing the design smells listed in the introduction I-B. These source code examples were collected from the StackOverflow and Github platforms.

The designed smell detection system thus developed was used on a set of deep learning programs collected from the GitHub platform. These deep learning programs were collected according to a well-defined process. We used the GitHub API and, more precisely, search queries of the type *search code*. This query searches Github repositories for files containing specific keywords defined in our system. As our paper focuses solely on the *Keras*, *Tensorflow* and *Pytorch* frameworks, we have used the keywords presented in the table III. These keywords represent the modules and functions of these three frameworks.

Table III: *List of keywords used to search for deep learning programs in Github repositories.*

| Key word | Framework |
|---|---|
| keras.layers | Keras |
| keras.layers.convolutional | Keras |
| AveragePooling2D | Keras/TensorFlow |
| MaxPooling2D | Keras/TensorFlow |
| tensorflow.keras.layers | Tensorflow |
| Conv2D | Keras/TensorFlow |
| Convolution2D | Keras/TensorFlow |
| BatchNormalization | Keras/TensorFlow |
| import torch | Pytorch |
| import torchvision.models | Pytorch |
| torch.nn.Sequential | Pytorch |
| torch.nn.Conv2d | Pytorch |
| torch.nn.BatchNorm2d | Pytorch |
| torch.nn.MaxPool2d | Pytorch |

The search query type *search code* returns a set of information about the repository and the file containing the searched keywords. This returns a raw data set of 9572 different Github repositories. We then randomly select a subset of 10% of our raw data set, i.e. 958 Github repositories, to optimize the manual filtering step below. From this subset, we then filter the directories according to the following criteria: (1) the number of commits ($\geq$ 100), (2) the number of stars ($\geq$ 100), (3) last commit date ($\leq$ 5 years), (4) the number of contributors ($\geq$ 5). This filtering allows us to keep only the most popular and active directories. We then filtered the directories manually, eliminating projects that are not deep learning programs or not relevant to our study (don't use the chosen frameworks, or don't implement neural networks). As a result, we ended up with 500 projects on which to apply our design smell detection system.
The schema 1 presents the process of collecting deep learning programs.

### B. Data Processing

The process of collecting deep learning programs has provided us with a set of deep learning programs on which we will apply our design smell detection system. However, it's important to note that the deep learning programs in our case are programs written solely in the Python language. It is therefore essential to note that, the design smell detection system developed in this paper is a system that is only capable of detecting design smells in deep learning programs written in the Python language.

To answer the research question *RQ1*, we have developed a system for detecting design smells in deep learning programs. This system is divided into three parts. The first part deals with the modeling of deep learning programs. The second part concerns the detection of design smells in deep learning programs. The third is the analysis of detection results.

*1) Modeling deep learning programs:* Modeling deep learning programs is the first step in the design smell detection process. This involves transforming the source code of the collected deep learning programs into an intermediate model that will be used for design smell detection. This intermediate model is a *Famix Abstract Syntax Tree* (*FAST*) model 2. The *FAST* model is used to represent programs as syntax trees. It inherits from the abstract source code representation *Famix* developed in the *Pharo* programming language. With *Famix*, source code is represented in object form and according to a defined metamodel to simplify the analysis and representation of complex programs. A *Famix* model is a language-agnostic representation of the source code and includes functions for navigation and transformation.

Transformation of the collected source code into the *FAST* model is as follows. Each collected project is cloned into a local directory. Its source code is then parsed using the Python library *ast.py*. This library transforms the source code into a syntax tree in *json* form. The resulting *json* is then transformed into a *FAST* model using an importer developed under the visitor design pattern. Visitors are functions that can navigate a syntax tree to perform specific operations on its nodes. In our case, we have developed visitors that enable us to browse the syntax tree (*json*) and transform each node of this tree into a *FAST* entity.

The diagram 3 illustrates the modeling process for deep learning programs.

*2) Design smell detection:* The *FAST* model obtained from the modeling of deep learning programs is then used for design smell detection. This step consists of browsing the *FAST* model and applying design smell detection rules. These rules are defined in the *Pharo* programming language. One or more rules can be applied to a node in the syntax tree to detect a design smell.

The diagram 4 illustrates the design smell detection process.

*3) Analysis of design smell detection results:* The results of each repository's design smell detection are stored in the form *csv*. For each repository, we record the name of each design smell detected, the number of times that smell is detected in a repository file and the path to that file. These results are used to answer the research questions *RQ2* and *RQ3*.
We then calculate the distribution of design smells in each repository by dividing the number of occurrences of a design smell by the total number of design smells detected in the repository. This distribution answers the research question *RQ2*. The distribution of design smells across all 500 repositories is also calculated by dividing the number of occurrences of a design smell by the total number of design smells detected across all 500 repositories. This distribution also answers the research question *RQ2*.

We then calculate the correlation matrix between design smells in all 500 repositories. This matrix answers the research question *RQ3*. A correlation matrix is used to calculate the correlation between two variables.

A correlation coefficient is calculated using the following formula: $r = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$ where $r$ is the correlation coefficient, $cov(X, Y)$ is the covariance between the two variables $X$ and $Y$, $\sigma_X$ is the standard deviation of the variable $X$ and $\sigma_Y$ is the standard deviation of the variable $Y$. This correlation coefficient ranges from -1 to 1. A positive correlation exists when the correlation coefficient ranges from 0 to 1. In this case, the closer it is to 1, the more positively correlated the two variables are, and the closer it is to 0, the less correlated the two variables are. A negative correlation is defined as a correlation coefficient between -1 and 0. In this case, the closer the correlation coefficient is to -1, the more negatively correlated the two variables are, and the closer it is to 0, the less correlated the two variables are.

In our case, the variables are the design smells. The correlation between two design smells is calculated by dividing the number of occurrences the two design smells are detected together by the total number of times the two design smells are detected. This correlation is calculated for each pair of design smells. The closer it is to 1, the more the two design smells are correlated, and the closer it is to 0, the less the two design smells are correlated.

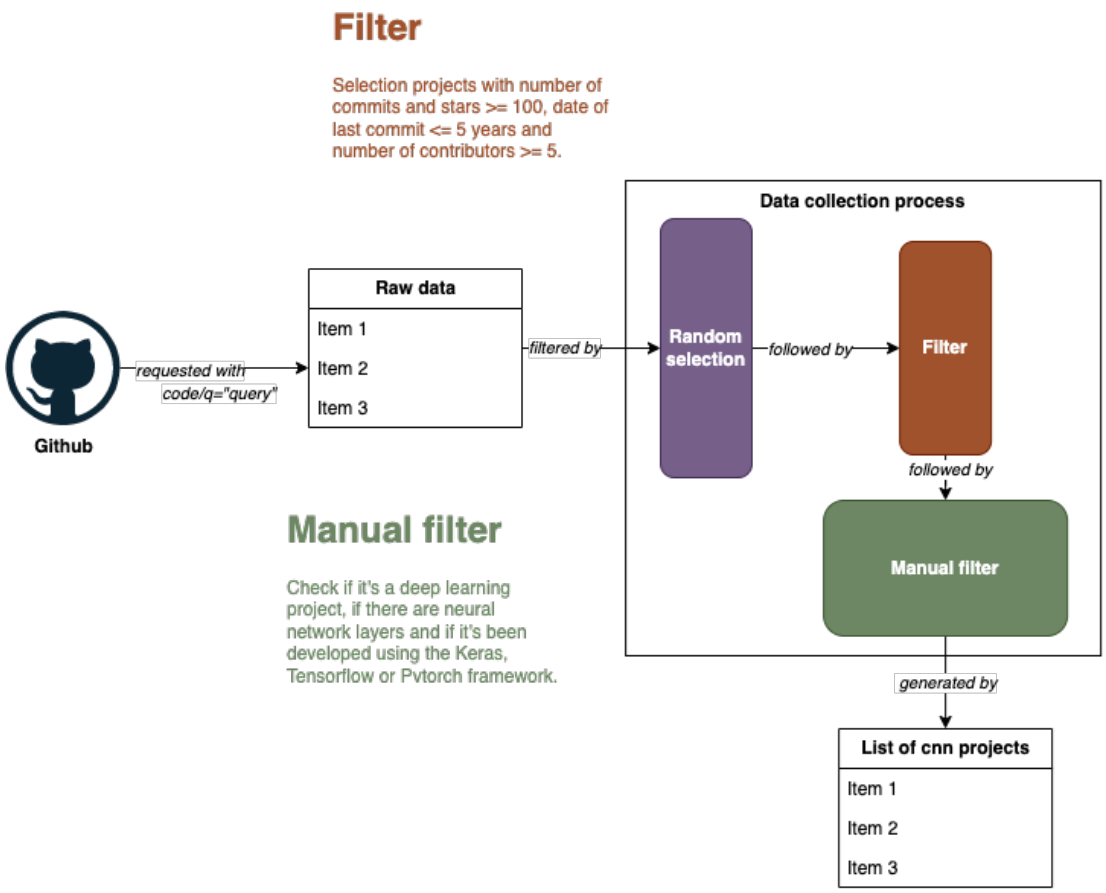


Figure 1: *Process for collecting deep learning projects in Github.*

A positive correlation means that the two design smells are often detected together. A negative correlation means that the two design smells are rarely detected together. Finally, a zero correlation means that the two design smells are never detected together.

The diagram 5 illustrates the process of this last part.

### C. Replication Package

Source code for the *json* syntax tree source code parser is available on Github at https://github.com/aurpur/parserPythonToJson, and source code for the smell detection system is available on Github at https://github.com/aurpur/famixPythonImporter. The data collection system is also available on Github at https://github.com/aurpur/ms-github-data-collection.

## IV. Case Study Results

In this section, we present the results of our study. First, we present the results relating to the detection of design smells in the programs studied. Then, we present the results relative to the analysis of the results. **TODO: To be completed**

### A. Design smell detection

Here we will present numbers on the detection of design smells in all repositories, such as the distribution of design smells in the dataset (the number of repositories per design smell), or the depth (number of occurrences of a design smell in a repository). This will support the hypothesis that design smells can be detected on CNN programs through a *FAST* model in Pharo. As a result, design smells can be detected using modeling.

### B. Detection results analysis

Here we present the figures on the analysis of design smell detection results in the programs studied. The aim is to present the ranking of design smells and to present numbers on the possible correlation between them. This will support the hypothesis that some design smells are more prevalent than others in CNN programs (in our study frame of reference) and that some design smells show correlations between them.

## V. Discussion

In this section, we discuss the results of our study and answer our research questions. We also discuss the implications of our results in the context of software research and development. **TODO: To be completed**

### A. RQ1: Is it possible to detect design smells in CNN deep learning programs with modeling?

Here we will discuss the results of our first research question. We will discuss the proposed design smell detection technique and its performance. We will calculate the accuracy, precision, and recall of the proposed technique. The execution time of the proposed system will also be discussed. We'll talk about the objective (or motivation) behind the question, and recall the method used and the findings.

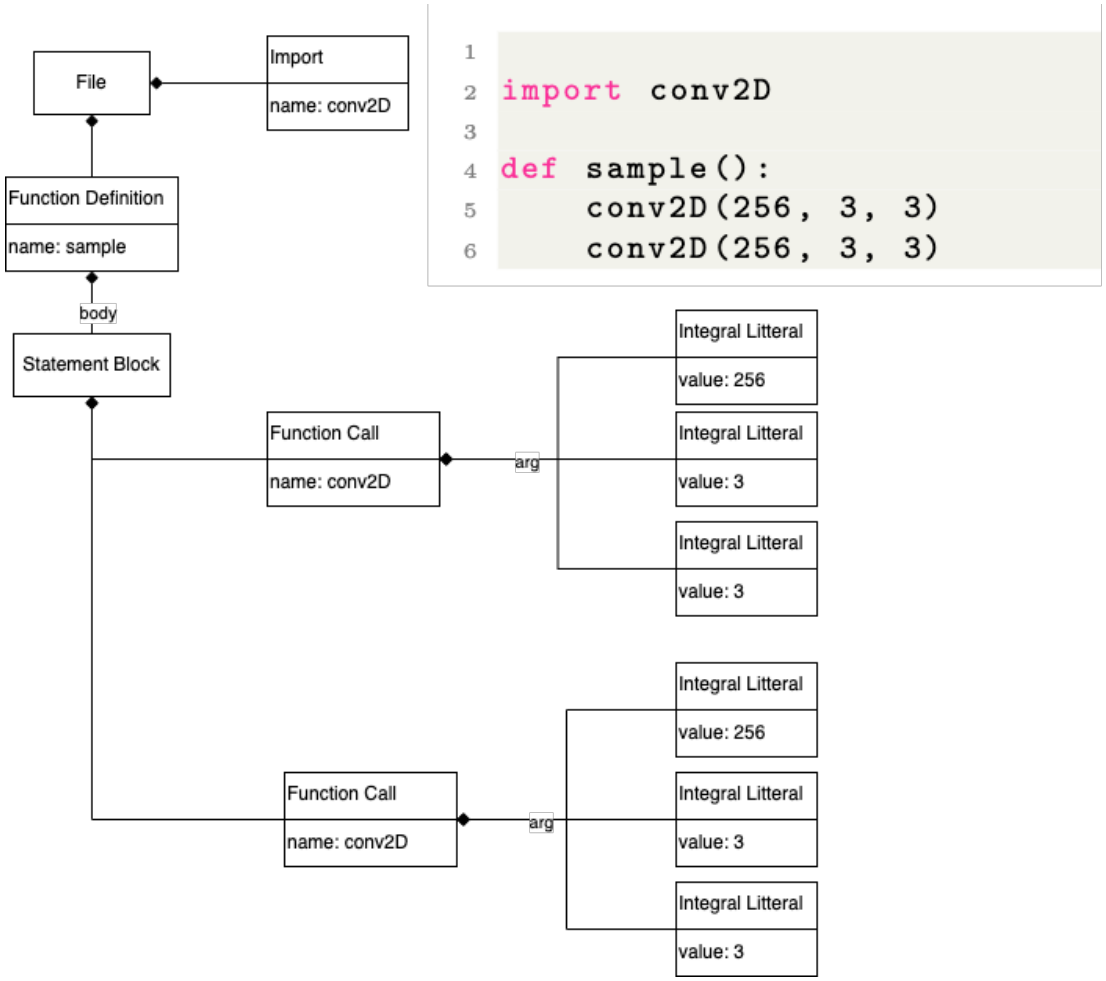


Figure 2: *Sample* FAST *model of a fictive deep learning program.*

### *B. RQ2: What are the most prevalent design smells in CNN deep learning programs?*

Here we will discuss the results of our second research question. We'll discuss the distribution and depth of design smells in the dataset. We'll talk about the objective (or motivation) behind the question, and recall the method used and the findings.

### *C. RQ3: Are there any correlation between the program design smells?*

Here we'll discuss the relationship between design smells in deep learning programs. We'll talk about the objective (or motivation) behind the question, and recall the method used and the findings.

### *D. Implications*

Here, we will discuss the impact of our study on software research and development. We'll also talk about the impact of our results.

## VI. Threats to validity

Our study is subject to several threats to validity. In this section we present these threats and the strategies we have used to mitigate them.

Firstly, our study is subject to the threat of construct validity. Indeed, the *FAST* model generated by our approach might not be correct and consistent with the defined metamodel, or it might misrepresent the input deep learning program. To mitigate this threat, we have set up unit tests for each visiting method. These unit tests enable us to check that the methods implemented in the importer are correctly implemented and that the *FAST* model generated conforms to the defined metamodel. In addition, for a set of test data (fictive samples), we have implemented functional tests (Smock Testing) to check that the *FAST* model generated does indeed represent the input deep learning program.

Secondly, our study is subject to the threat of internal validity. There remains a risk that the conclusions obtained are not caused by the input source code. To mitigate this threat, in addition to testing our system on synthetic examples, we used real projects collected from Github to validate our system's operation.

Thirdly, our study is subject to the threat of external validity. There is a risk that the results obtained may not be generalizable. To mitigate this threat, we randomly collected projects from Github to represent different applications of the CNN architecture.

Fourthly, our study is subject to the threat of reliability validity. Finally, there is a risk that the results obtained may not be reproducible. To mitigate this threat, in addition to using projects from different applications of the CNN architecture, we used projects implemented in the three most widely used open-source libraries in the industry (Tensorflow, PyTorch, and Keras).

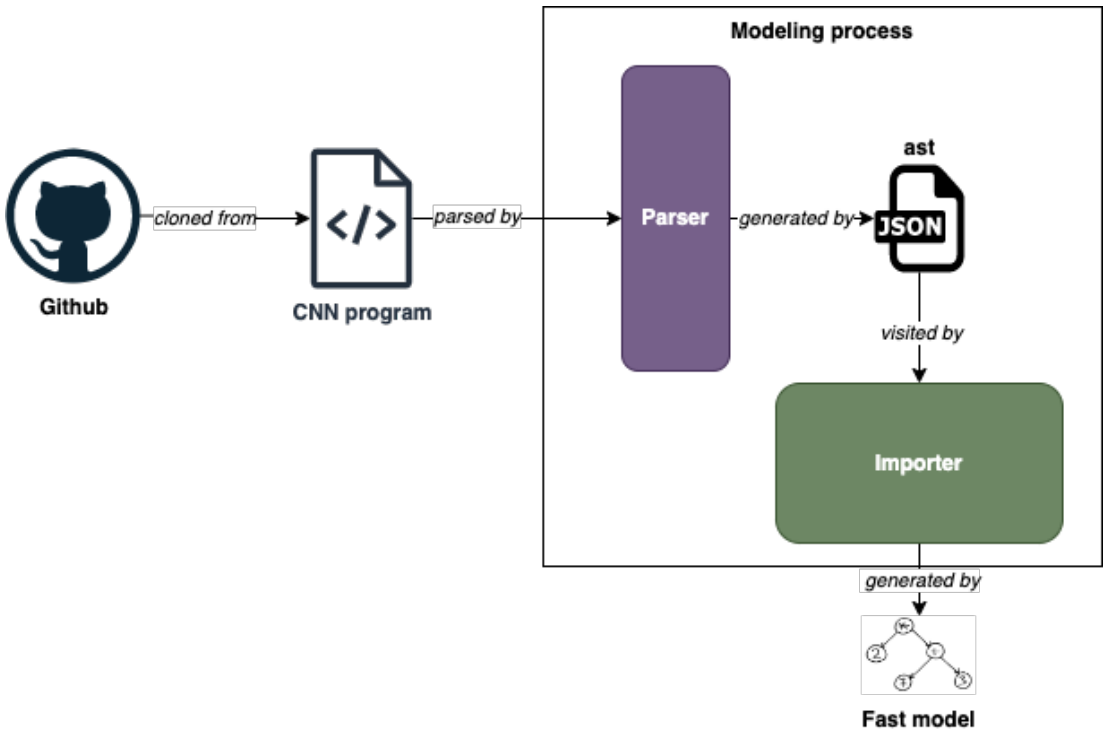


Figure 3: *Modeling process for deep learning projects.*

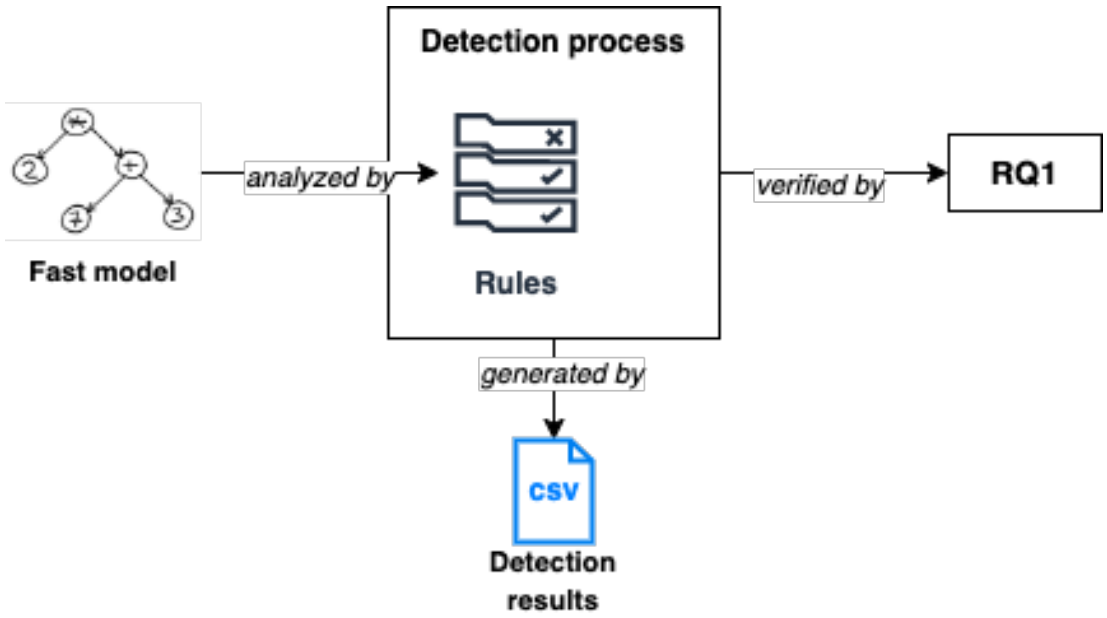


Figure 4: *Design smell detection process.*

## VII. Limitations and Future Work

*1) Limitations:* Our study took place in a limited context, which we will describe in this section. When we speak of deep learning programs, we are referring to programs containing a convolutional neural network (CNN). It is therefore important to note that other types of deep learning programs are not covered by our study. In addition, several types of libraries can be used to create deep learning programs. Our study is limited to programs created using the TensorFlow, Keras, and PyTorch frameworks. And our data comes exclusively from open-source repositories such as Github or StackOverflow (for samples). Finally, our design smell detection system is limited to detecting the 8 design smells described above.

*2) Future Work:* Future work can lead in several directions. Indeed, it is possible to apply our approach to other types of deep learning programs, other libraries, other design smells, other programming languages, and other types of repositories (non-open-source, for example). Although we have performed a static analysis of deep

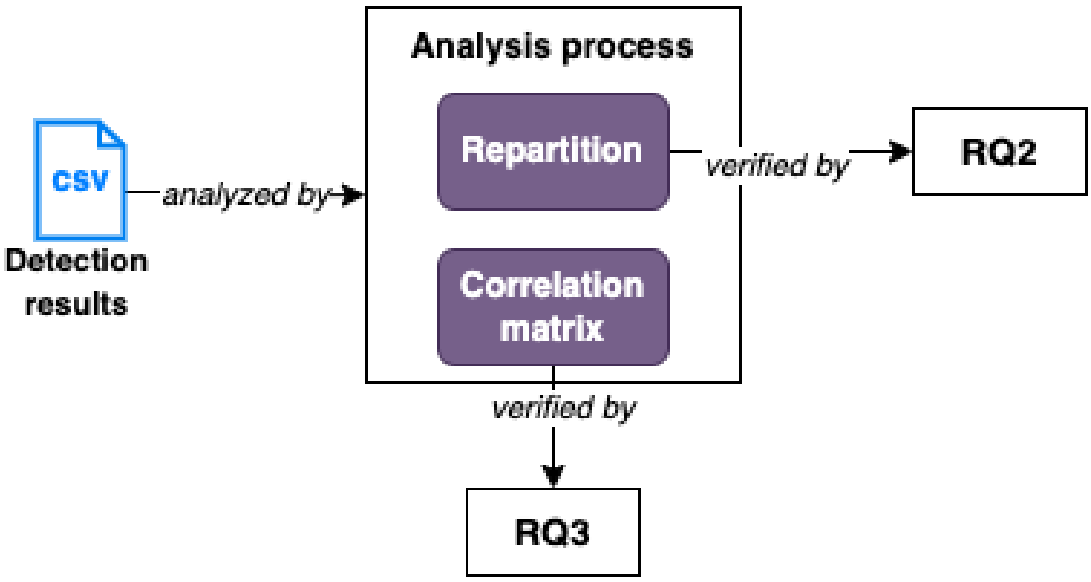


Figure 5: *Process for analyzing design smell detection results.*

learning programs, it is also possible to perform a dynamic analysis by simulating program execution through modeling. In addition to detection via modeling, other fields of research are detection via machine learning and automatic correction of design smells. These are very interesting research topics that could have a major impact on the quality of future deep-learning programs.

## VIII. Conclusion

**TODO: To be completed**

## References


[1] M. Fowler and K. Beck, "Refactoring: Improving the design of existing code," in *11th European Conference. Jyväskylä, Finland*, 1997.

[2] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[3] A. Nikanjam and F. Khomh, "Design smells in deep learning programs: an empirical study," in *2021 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2021, pp. 332–342.

[4] T. Kühne, "Matters of (meta-) modeling," *Software & Systems Modeling*, vol. 5, pp. 369–385, 2006.

[5] C. A. González and J. Cabot, "Formal verification of static software models in mde: A systematic review," *Information and Software Technology*, vol. 56, no. 8, pp. 821–838, 2014.

[6] A. P. Black, O. Nierstrasz, S. Ducasse, and D. Pollet, *Pharo by example*. Lulu. com, 2010.

[7] A. Bergel, D. Cassou, S. Ducasse, and J. Laval, *Deep Into Pharo*. Lulu. com, 2013.

[8] O. Zaitsev, S. Ducasse, and N. Anquetil, "Characterizing pharo code: A technical report," Ph.D. dissertation, Inria Lille Nord Europe-Laboratoire CRIStAL-Université de Lille; Arolla, 2020.

[9] S. Tichelaar, S. Ducasse, and S. Demeyer, "Famix and xmi," in *Proceedings Seventh Working Conference on Reverse Engineering*, 2000, pp. 296–298.

[10] S. Demeyer, S. Ducasse, and S. Tichelaar, "Why famix and not uml," in *Proceedings of UML'99*, vol. 1723, 1999.

[11] S. Ducasse, M. Lanza, and S. Tichelaar, "Moose: an extensible language-independent environment for reengineering object-oriented systems," in *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, vol. 4, 2000.

[12] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol, "An empirical study of code smells in javascript projects," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 294–305.

[13] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security smells in ansible and chef scripts: A replication study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 1, pp. 1–31, 2021.

[14] T. Cerny, A. Al Maruf, A. Janes, and D. Taibi, "Microservice anti-patterns and bad smells. how to classify, and how to detect them. a tertiary study," *How to Classify, and How to Detect Them. A Tertiary Study*.

[15] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the international conference on mobile software engineering and systems*, 2016, pp. 59–69.

[16] A. M. Fard and A. Mesbah, "Jsnose: Detecting javascript code smells," in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2013, pp. 116–125.

[17] Z. Chen, L. Chen, W. Ma, and B. Xu, "Detecting code smells in python programs," in *2016 international conference on Software Analysis, Testing and Evolution (SATE)*. IEEE, 2016, pp. 18–23.

[18] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, 2002, pp. 97–106.

[19] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[20] P. Araujo, S. Rodríguez, and V. Hilaire, "A metamodeling approach for the identification of organizational smells in multi-agent systems: Application to aspecs," *Artificial Intelligence Review*, vol. 49, pp. 183–210, 2018.

[21] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *2009 Ninth International Conference on Quality Software*. IEEE, 2009, pp. 305–314.

[22] S. Alawadi, K. Alkharabsheh, F. Alkhabbas, V. Kebande, F. M. Awaysheh, and F. Palomba, "Fedcsd: A federated learning based approach for code-smell detection," *arXiv preprint arXiv:2306.00038*, 2023.

[23] R. Sandouka and H. Aljamaan, "Python code smells detection using conventional machine learning models," *PeerJ Computer Science*, vol. 9, p. e1370, 2023.

[24] A. Alazba, H. Aljamaan, and M. Alshayeb, "Deep learning approaches for bad smell detection: a systematic literature review," *Empirical Software Engineering*, vol. 28, no. 3, p. 77, 2023.

[25] A. Nikanjam, H. B. Braiek, M. M. Morovati, and F. Khomh, "Automatic fault detection for deep learning programs using graph transformations," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–27, 2021.