The Journal of Systems & Software 193 (2022) 111431

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Automatically recognizing the semantic elements from UML class diagram images☆

Fangwei Chen [a], Li Zhang [a], Xiaoli Lian [a,*], Nan Niu [b]

[a] State Key Laboratory of Software Development Environment, Beihang University, Beijing, 100191, China
[b] University of Cincinnati, Cincinnati, OH, 45221, USA

ARTICLE INFO

Article history:
Received 25 October 2021
Received in revised form 4 July 2022
Accepted 5 July 2022
Available online 9 July 2022

MSC:
00-01
99-00

Keywords:
UML image recognition
Class diagram
Inconsistency check

ABSTRACT

Context: Design models are essential for multiple tasks in software engineering, such as consistency checking, code generation, and design-to-code tracing. Almost all of these works need a semantically analyzable model to represent the software architecture design, e.g., a UML class diagram. Unfortunately, many design models are stored as images and embedded in text-based documentations, impeding the usage and evolution of these models. Thus, identifying the semantic elements of design models from images is important. However, there are lots of design models with different elements in diverse representations, which ask for different approaches for semantic elements extraction.
Objective: In order to grasp an overview of the commonly used design model types, we conduct a survey on both open-source communities and industry. We find that design model diagrams are usually embedded in documents as pictures (73.72%), and UML class diagrams are the most used type (55.43%). Considering that there are limited studies on automatically recognizing the semantic elements from class diagram images, we propose an approach, which we call ReSECDI.
Method: ReSECDI includes our customized design for extracting UML class diagram elements based on image processing technologies. We design a rectangle clustering method for class recognition, to address the challenge that the presentation of classes may vary due to the UML constraints and tools' styles. We design a polygonal line merging method and double-recognition-approximation method for relationship recognition to deal with the impact of low resolution on the detection.
Results: We evaluate the applicability of ReSECDI on 30 images drawn by three popular UML tools and 50 diagrams collected from the open-source communities, and get promising performances.
Conclusion: ReSECDI can recognize all types of semantic elements commonly used. It has well applicability and can be used to process the images drawn by the mainstream tools and stored in different resolutions.

Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.

1. Introduction

Design models are essential for multiple tasks in software engineering, such as source code understanding (Gravino et al., 2010), design-source code consistency checking (Caracciolo et al., 2015; Buckley et al., 2013; Adersberger and Philippsen, 2011), code generation (Amnell et al., 2003; Hains et al., 2015; Sharaf et al., 2019), and design-to-code tracing (Javed and Zdun, 2014; Sinkala and Herold, 2021; Chavez et al., 2015). Almost all of these works need at least one semantically analyzable model representing the design, such as a UML component diagram or class diagram (Koschke and Simon, 2003; Passos et al., 2009).

However, according to our practical experience and the collaboration with several industrial partners, we find that the models are usually stored as images embedded in documents, and the original editable models are easily lost. This definitely impedes the usage and evolution of these models. But, *is this problem commonly seen in practice?* If this problem is general, we need to explore automatically recognizing the semantic elements from images since the manual identification is tedious and time-consuming. However, there are lots of design models and each model has its own set of semantic elements with different representations. This means that it is impossible to design a unified approach that can be used on all of the design models. Therefore, *we need to know the popularly used design models as the focus of our*

☆ Editor: Matthias Galster.
* Corresponding author.
E-mail addresses: by1506151@buaa.edu.cn (F. Chen), lily@buaa.edu.cn (L. Zhang), lianxiaoli@buaa.edu.cn (X. Lian), nan.niu@uc.edu (N. Niu).

https://doi.org/10.1016/j.jss.2022.111431

Table 1
The semantic elements involved in the current inconsistency checking approaches.

	JITTAC (Buckley et al., 2013)	Biehl's (Biehl and Löwe, 2009)	Haitzer's (Haitzer and Zdun, 2012)	CCUJ (Chavez et al., 2015)	ArchTrace (Murta et al., 2006)	Chen's (Chen et al., 2020)
Class	✓	✓	✓	✓	✓	✓
Class's properties	×	✓	×	✓	✓	×
Generalization	×	✓	✓	✓	✓	✓
Realization	×	✓	✓	✓	✓	✓
Dependency	✓	✓	✓	✓	✓	✓
Aggregation	×	×	×	×	✓	✓

study. To answer these two questions, we design a survey to grasp an overview and understanding of the design model diagram usage among both open source communities and industry. We had two main observations from this survey:

- Design model diagrams are usually embedded in documents as pictures (73.72%), which are difficult to use, to reuse, to modify, and to evolve.
- According to the industry survey, UML diagrams are the most commonly used as standard design model diagrams (67%). Among all types of UML diagrams, the UML class diagram is the second commonly used one (the first is sequence diagram). The survey on open-source projects offers different results: the UML class diagram is the mostly used UML diagram (55.43%).

These observations drive us to focus on the automated recognition of semantic elements from UML class diagram images. However, such work faces four challenges:

(a) A semantic element may have different graphical representations, according to the standards of UML specification (Selic et al., 2015). For example, the rectangle number of the element *Class* may vary, depending on the existence of its *attributes* and *operations*.
(b) The graphical elements of one semantic element, constructed in different design tools, may vary. For example, the *implement* relationship between Classes is presented with dash lines in Rational Rose and dotted lines in StarUML.
(c) The diagram images may have different resolutions. This is an obstacle to distinguish similar graphical elements, such as the solid line with dotted line or dashed line.
(d) Gradient colors are common in the image background. This interferes with the distinguishment of some blocks and minor elements. For example, when the background color of a class's rectangle is gradually changed from white to black, the solidness of this rectangle is also gradually changed. It is highly possible that the class rectangle is recognized as two rectangles (i.e., one white and one black), causing a false recognition.

These challenges determine that the present image recognition technologies cannot solve our problem directly since they focus on the general image processing, while the UML-oriented graphical semantics need some extra works to deal with. We usually need to approximately compose the identified atomic shapes into valid and nested graphical elements of UML diagrams, such as the rectangle combinations of classes.

Currently, there is some related work on class semantic elements recognition, such as Img2UML (Karasneh and Chaudron, 2013b) and Tahuti (Hammond and Davis, 2006). However, the covered elements are limited, especially the relationships between classes. But some essential elements are required by the downstream work such as the consistency checking approaches. For the convenience of clarification, we take the consistency checking task as one example to show the elements required in the usage of design models. We collect the common consistency checking approaches such as JITTAC (Buckley et al., 2013) and ArchTrace (Murta et al., 2006), and list the involved semantic elements in Table 1. We can observe that the realization and dependency relationships are utilized by almost all of the approaches. However, they are not involved in the current work on semantic elements recognition from class diagram images (Karasneh and Chaudron, 2013b; Hammond and Davis, 2006). Besides, they only focus on the models built in a specific tool, e.g., StarUML tool by Img2UML (Karasneh and Chaudron, 2013b), and Rational Rose by Tahuti (Hammond and Davis, 2006). It is highly possible that they cannot effectively identify the elements built by other tools considering the difference between the models built in different tools. In addition, the challenges caused by the resolution and gradient colors are not specified in these approaches, since they focus on images built by the authors.

In this work, we propose an approach, ReSECDI, for the automated ***Re***cognition of ***S***emantic ***E***lements in ***C***lass ***D***iagram ***I***mages built with the popular modeling tools and stored in different resolutions. In our approach, we propose a rectangle merging method to deal with the challenge of different class styles with varied numbers of rectangles, a polygonal line merging method to deal with the recognition of polygonal relationship lines in different styles and resolutions, and a double-recognition-approximation method to address the challenge of relationship type symbols in different sizes and resolutions. Experiments on the diagrams drawn with three widely-used UML tools and collected from the open source communities show our approach has an overall recall and precision around 90%. Our approach can also cover all types of semantic elements that are commonly used by the existing consistency checking approaches.

In general, this work makes three main contributions:

- We conduct a survey to gain an overview of the current status of the design diagram usage in both open-source communities and industry.
- We propose an approach for recognizing semantic elements in the UML class diagram images, with the aim to suit more situations of diagrams drawn by different tools in different resolutions.
- We evaluate our approach on 30 diagrams drawn by three widely-used UML tools, as well as on 50 diagrams collected from open-source communities. Results show that our method performs well with an average recall and precision around 90%.

The rest of this paper is organized as follows: Section 2 states the related works. Section 3 introduces the design and result of the survey toward the usage of design model diagrams in both open source community and industry. Section 4 explains our approach for the UML class diagram recognition. Section 5 presents the experiment design and result analysis for our approach. Section 6 states the conclusion of our work. A bundle of Open Science materials are provided on Zenodo (Chen, 2022).

2. Background and related works

2.1. Background: Semantic elements in class diagrams

According to UML 2.5 Specification (Selic et al., 2015), the UML class diagram is a type of static structure diagram that

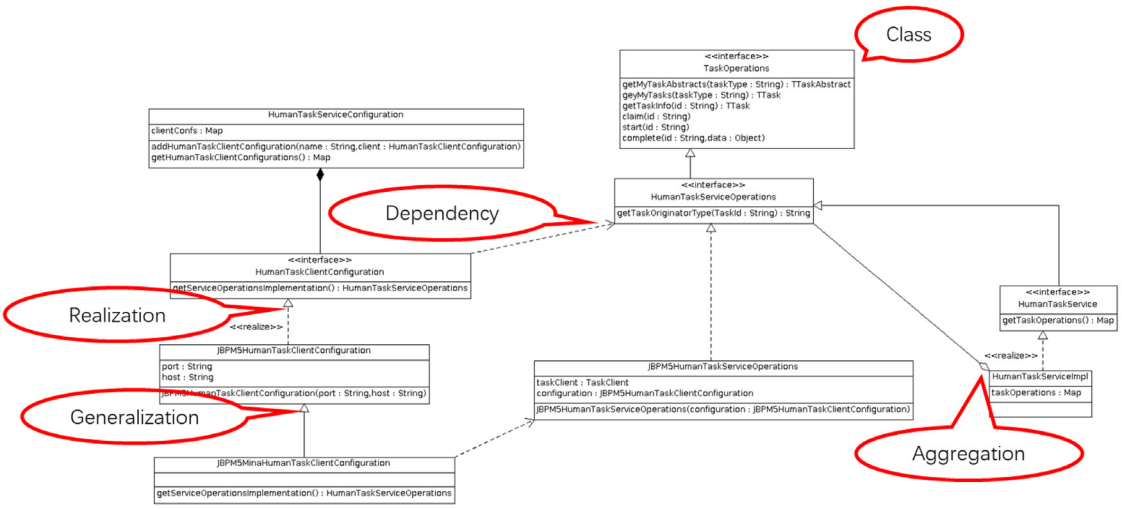


Fig. 1. An example of class diagram.

describes the structure of a system by showing the system's classes and the relationships among them. Note that although a class diagram may contain lots of contents, such as its class name, comments, various types of relationships, or other additional information, the most commonly used elements are limited (the commonly used semantic elements are shown in Fig. 1). As we just discussed in the Introduction (the details can be found in Section 2.2), although the current approaches perform well on the recognition of class and generalization relationship drawn by a specific tool (i.e., StarUML), the recognition of realization and dependency relationships still needs improvements. Thus, we would like to cover more popular UML tools. And we concentrate the recognition of the elements as follows:

- Class: A classification of instances according to their features. Classes are organized in hierarchies by Generalizations.
- Relationship: A relationship is composed of the relationship line and the relationship type symbol. We need to recognize the solid lines and dashed lines, and the shapes representing the types of relationships (including triangle, arrow, and diamond).
  - The Generalization (a taxonomic relationship between a more general Class and a more specific Class) is composed of a solid line and a triangle.
  - The Realization (a specialized abstraction relationship between a specification and an implementation) is composed of a dashed line and a triangle.
  - The Dependency (a relationship that signifies a class requires other classes) is composed of a dashed line and an arrow.
  - The Aggregation (a whole/part relationship) is composed of a solid line and a diamond.
- Text: The names, attributes, and operations of Classes.

We aim to recognize the elements listed above within a class diagram image, and generate a file containing all the semantic information of the recognized elements. Since we only focus on the semantic usage (for example, the consistency checking) of class diagrams, we do not record the location information of elements (such as the coordinates, or the sizes of areas).

The recognition is faced with some challenges.

- **A semantic element may have different graphical presentations.** For example, according to the standards of UML specification (Selic et al., 2015), the Class may not have attributes or methods, making the rectangles of one Class vary, as shown in Fig. 2. Classes that consist of various numbers of rectangles make the recognition of classes' contents difficult.
- **Diagrams drawn by different tools may have different presentations**. This may impact the recognition of semantic elements. To the best of our knowledge, the major UML modeling tools are StarUML, Enterprise Architect ("EA" in the rest of this paper), and Rational Rose. For example, the segment distances in the dashed line or dotted line are different, as shown in (Fig. 3(a)). In addition, for the same graphical element, different tools may present different styles (e.g., different size of diamond), as shown in (Fig. 3(b)). Since we expect to propose an approach of processing the images built in different tools, these differences are an obstacle.
- **Diagrams may be stored in different resolutions**. As shown in Fig. 4, the same dashed line is stored in the files with different resolutions. Since the recognition is based on the calculation of line segments, this may infer the decision of whether a line is solid or dotted.
- **Gradient colors are commonly used in the image background**. This impacts the recognition of rectangles and minor elements. For example, when the background color of a class's rectangle is gradually changed from light to deep (as shown in Fig. 5), the solidness of this rectangle is also gradually changed. Then, the class rectangle is possibly recognized as two rectangles (i.e., one with light color and the other one with deep color), causing a false recognition.

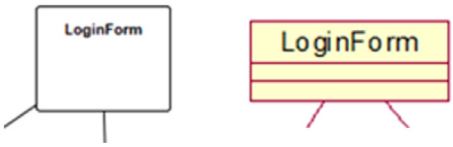
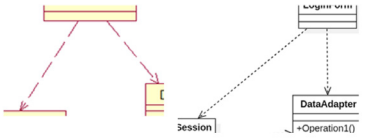


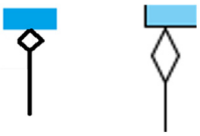
Fig. 2. A semantic element may have different graphical representation: one example of Class.

2.2. Related work

There are some studies on the usage of design models among open source communities and industry. Hebig et al. (2016) scan ten percent of all GitHub projects and collect UML stored in images. They identify 21,316 UML diagrams within 3295 projects,



(a) Different segment distances impact the relationship line recognition.



(b) Different symbol styles impact the relationship type recognition.

Fig. 3. Diagrams drawn by different tools may have different presentation.

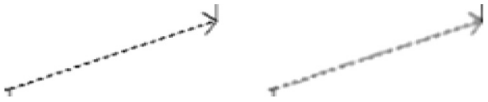


Fig. 4. The images may be stored with different resolutions: one example.

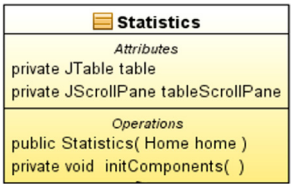


Fig. 5. The background color may be gradient: one example of a class. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

and provide an online repository,[1] which contains a list of open-source projects that use UML. Robles et al. (2017) extend Hebig's work, and scan over 12 million GitHub projects. They identify 93,607 UML diagrams within 24,730 projects. Karasneh and Chaudron (2013c) also use a crawling approach to establish an online UML repository, models-db,[2] with more than 700 model images. Their work focuses on the aspects of the model usage, such as the number of models in projects, or the model update frequency, etc. These online repositories are a good basis for us to do further researches, such as finding the most commonly used types of UML diagrams. Reggio et al. (2014) investigated which UML diagrams are used among online books or modeling tools with a survey in the industry. The results show that the most widely used UML diagrams are class diagrams and sequence diagrams. Ozkaya and Erata (2020) aim at understanding the practitioners' UML usage for the software architecture modeling. The results show that the UML class diagram is the practitioners' top choice for modeling the functional structure, software code structure, and so on, while the UML sequence diagram is the top choice for modeling the data lifecycle. They also conclude that the top popular UML modeling tool is EA, which could be helpful for the case selection during our approach's evaluation.

Since the technical problem in this work is how to address the UML class diagram recognition, the rest of the related works mainly review current recognition approaches. Few studies on the semantic recognition of design model diagrams exist, and the recognition subjects are mainly on UML class diagrams. Currently, the main method is Img2uml (Karasneh and Chaudron, 2013a,b). This approach uses AForge.NET (Kirillov, 2013) and Gaussian Sharpen (Dogra and Bhalla, 2014; Machado et al., 2015; Nam and Ahuja, 2012) to detect the shape of a class, and then uses the Optical Character Recognition (OCR) (Mithe et al., 2013) technique to recognize the texts in a class. This work can also detect some simple forms of relationship lines and types, such as straight lines and solid lines, while it has difficulties on the recognition of realization and dependency relationships, and the integration of the atomic elements of relationship and class. This team develops a CASE tool that can recognize the semantic information (i.e., classes and relationships) in class diagram images (e.g., BMP, JPG, PNG, and GIF files), and save the recognition results in a UML file. The evaluation of this approach involves 10 UML class diagrams which mainly follow the StarUML style, resulting in good performance on the recognition of class and generalization relationship. This method can be a good reference for class rectangle detection, although the recognition of realization and dependency still needs some improvement. Hammond and Davis (2006) proposed Tahuti, a tool that uses feature extraction and area detection methods to recognize class rectangles in hand-drawn class diagrams. This approach can recognize the classes with few attributes and methods (currently no more than 10), but it has limitations as the number of attributes and methods grows larger.

Some studies focus on the classification of UML diagrams, and provide valuable reference on the recognition algorithm and feature selection. The dtsUMLClassifier (Ho-Quang et al., 2014) uses feature extraction and machine learning techniques to address the UML diagram classification problem caused by the variety of diagram patterns and styles that exist in open source projects. They found 19 significant features for the class diagram recognition, including the area of rectangles, separators in rectangles, the relationships connecting rectangles, etc. Although this study lacks solid methods for relationship recognition, its feature selection method can be helpful for our class recognition. Moreno et al. (2020) develop i2 m (image-to-model) to automatically classify web images as UML static diagrams (class diagram, component diagram, etc.) with machine learning techniques. They provide some values of recognition parameters, such as the gaps in dashed lines is five pixels, and the minimum length for vertical/horizontal segments is 30 pixels, etc. Gosala et al. (2021) apply Convolutional Neural Networks (CNN) for identifying UML class diagrams from all images (UML or non-UML images). Evaluations show that the classification accuracy is greatly affected by parameters like the image size, the layers of neural networks, and the size of kernels, etc. Tavares et al. (2021) also use CNN to classify six types of UML diagrams in order to help software engineering education. Ott et al. (2019) explore the low-shot learning for classifying UML class and sequence diagrams from GitHub, while Best et al. (2020) explore the applicability of transfer learning for this problem. Both works can detect the location of classes in class diagrams and messages in sequence diagrams. However the relationship types (e.g., the dependencies) are ignored. Shcherban et al. (2021) propose a neural network architecture to identify four types of UML diagrams (class diagrams, sequence diagrams, use case diagrams, and activity diagrams) from web images. Compared with other neural network architectures, their architecture has the least parameters (around 2.4 millions) and spends the least time per image for the classification. Osman et al. (2018) present an automated classifier for differentiating forward-designed class diagram (FwCD) from reverse-engineered class diagram (RECD). They propose some features for the classification, such as the number of classes, attributes, and operations in the class diagram.

1 http://oss.models-db.com/.
2 http://models-db.com/.

Since there are few studies on the recognition of semantic elements (e.g., classes and relationships) of UML diagrams, we broaden our survey scope on the field of computer vision, in order to find some additional supporting approaches. Actually, there exist several approaches used for class rectangle detection, relationship line, and type detection, such as morphological operations (Chudasama et al., 2015) and hough transform (Mukhopadhyay and Chaudhuri, 2015; Beltrametti et al., 2013; Barinova et al., 2012). Some object detection methods can be helpful for detecting the graphical elements in an image. Traditional object detection methods recognize a pre-defined object through area selection and feature extraction (Zhao et al., 2019; Szegedy et al., 2013). The R-CNN methods (Girshick, 2015; Ren et al., 2015; Purkait et al., 2017) transform the detection problem to a classification problem, which can detect objects within a proposed region. YOLO (Laroca et al., 2018) transforms the detection problem into a regression problem, which performs well on the real-time automatic license plate recognition. However, the object detection methods can only detect standard shapes like rectangles, while it has limitations on integrating the detected shapes and texts into UML classes. Some methods can help the relationship detection. The hough transform (Illingworth and Kittler, 1988; Hassanein et al., 2015; QIN et al., 2010) is an image feature extraction algorithm, which performs well on line, rectangle and circle detection. Traditional hough transform searches the pixels (such as black pixels) and transforms the search results into a linear function (y=kx+b). In the UML class diagram, many lines are vertical with an infinite gradient, which may cause low recognition performance with the traditional hough transform. The improved hough transform (Duan et al., 2010) can detect lines with the polar coordinate function, which might be useful for UML relationship detection. However, these methods can only detect relationship lines or relationship type symbols separately, while the integration of these semantic elements is not involved.

To sum up, the element recognition of class diagrams is of great importance. Currently, the recognition of the elements of *Class* (including the class name, attributes and operations) from the images built in specific tools (i.e., StarUML) yields great performance. However, the identification of relationships, especially the realization and dependency, is obviously weak. Besides, the integration of the elements of relationship and the corresponding classes are rarely involved in the current work. The field of computer vision provides many useful methods, while some methods specific to the UML element processing still need to be explored.

3. Survey on the design model diagram usage

We design a survey to grasp an overview of the design model diagram usage in practice. Since GitHub is the most popular hosting site for open-source community repositories, it can somehow represent the developers' habits of design model diagram usage. Moreover, according to Hebig et al. (2016)'s online UML repository, there are many projects using UML on GitHub. So, we would like to know the most commonly used types of UML diagrams and their presentation forms.

As GitHub is an online hosting platform for software projects, source code is the primary artifact to be managed. It does not contain the information on the diagram usage during the software life-cycle. Therefore, we also design a questionnaire for the industry practitioners, complementary to the online survey in GitHub.

Our survey aims at answering the following research questions:

- **What are the commonly used design models?** We want to gain an overview about the commonly used design models, including the number of design model diagrams used in projects, the commonly used diagram types (maybe UML class diagram, sequence diagram, etc.), the frequencies of each model type, and the distribution of UML diagrams among different scales of projects. Besides, the overview of design model usage can help us eliciting our research scope, such as the types of design model diagram we need to recognize, and the diagram drawing tools we should cover.
- **What are the main concerns on design model usage?** We want to grasp the main concerns on model usage, in order to figure out the necessity of semantic recognition of design model diagrams.

3.1. The usage of design model diagrams in open source communities

In this part, we aim to collect the UML diagrams used by GitHub projects, and find out the most commonly used types of UML diagrams, the forms of diagrams' presentation, and the commonly used tools.

For the data collection from open-source communities, we take models-db (Hebig et al., 2016; Robles et al., 2017) as the reference data set, and mine the information of used UML diagrams for the analysis. Models-db is an online UML repository created in 2016, and updated in 2017. It contains 24,730 GitHub projects using UML, which come from over 12 million GitHub projects. The total number of UML diagrams in models-db is 93,607. It provides the URLs of projects and images. Among the few available online repositories of UML diagrams from GitHub, the scale of models-db is the biggest. However, due to the fact that many projects and links to UML diagrams are not available anymore, we need to recollect the data. We also would like to know the distribution of UML diagrams among different scales of projects, but the LOC of projects are not collected in the models-db. In order to get the data we need, we use the URLs of GitHub projects and identified UML images provided by models-db to collect project metadata (including the project's programming language, and the source code files) and UML diagram images till November 2020, and filter out all invalid links (about 25% of all links). Since GitHub does not provide an effective way to count the LOC of repositories, we count the LOC by uploading source code files to Codetabs[3] (an online API for counting LOC).

Eventually, we get 18,054 projects with a total of 68,313 UML diagrams, including images and XML-formed files. More than 90% of the projects contain 1 to 9 UML diagrams, while only less than 10% projects contain more than 9 diagrams. From these projects, we want to know what types of diagrams are mostly used. Thus, we design a tactic to count their frequencies. Particularly, for each type of UML diagram, we choose the diagram's name and its synonyms as the search query to get all related images. For example, we choose "class" as the query for class diagram; for component diagram, we use not only "comp", but also words like "architect" or "module". We also calculate the median and maximum LOC for projects using each type of UML diagrams, since we wish to find out what are the scales of projects that use UML. The statistics of UML diagram type usage are shown in Table 2.

In Table 2, we show the distribution of different UML diagrams in the projects with different numbers of UML diagrams in GitHub (i.e., 1–9 shown in the column of *#UML Diagrams*). For example, according to the first row, in the projects containing only one UML diagram, 18.12% contain Class diagrams, taking the top position. The remaining columns in this row represent the ratio of sequence diagrams, component diagrams, use case diagrams, state diagrams, deployment diagrams, activity diagrams, and package diagrams. In the last zone of *LOC*, we give the median

[3] https://codetabs.com/count-loc/count-loc-online.html.

Table 2
UML diagram type usage in GitHub projects.

#UML diagrams	Class (%)	Seq. (%)	Comp. (%)	UseCase (%)	State (%)	Deploy. (%)	Act. (%)	Package (%)	LOC	
									Median	Max
1	18.12	1.53	0.01	1.37	1.04	0.20	0.56	0.31	3132	16,117,762
2	11.66	2.28	3.76	1.50	2.71	0.30	0.59	0.74	3914	13,577,004
3	6.37	1.72	1.37	0.94	0.59	0.13	0.46	0.77	3760	6,409,345
4	5.87	1.51	1.04	0.96	0.70	0.14	0.43	0.83	4561	5,201,617
5	4.78	1.40	0.87	0.71	0.55	0.14	0.66	1.29	6949	9,577,981
6	3.42	1.71	0.95	0.55	0.20	0.10	0.31	1.41	6517	6,642,455
7	2.05	0.67	0.59	0.61	0.19	0.08	0.23	1.37	4075	7,319,934
8	1.44	1.02	0.41	0.32	0.22	0.05	0.20	1.33	8036	15,808,148
9	1.73	0.52	0.43	0.37	0.12	0.09	0.32	1.41	5256	2,375,521
Total	55.43	12.36	9.45	7.33	6.32	1.24	3.77	9.47	Avr. 5133	Avr. 9,225,529

Table 3
Distribution of UML model diagrams by storage formats.

	Images (73.72%)						Text-based Models (26.28%)	
Format	.png	.jpg	.gif	.svg	.jpeg	.bmp	.uml	.xmi
Ratio (%)	49.80	14.94	4.78	2.17	1.12	0.51	21.65	4.63

and max values of the projects' LOC. Looking at the last row, we can infer that the most commonly used UML diagrams are class diagrams (55.43%) and sequence diagrams (12.36%), highlighted with a green cell background.

During the software development process, the UML diagrams may be saved as different formats. We count the distribution of the storage formats of UML models in GitHub projects, shown in Table 3. We can see that most diagrams are saved as raster image files (73.72% in total), including popular picture formats (such as PNG, JPG, BMP, GIF, etc.). The remaining 26.28% is text-based models, including UML files (.uml) containing the complete design model information, or an XML-formed file only containing semantic information of the design model.

We also manually identify the involved UML modeling tools according to the diagram styles, since the diagrams built on the popular tools have distinguishing features. For example, the classes drawn by Rational Rose have pink edges and a pure yellow background color. The relationships drawn by StarUML have bigger arrows than those drawn by many other tools. Diagrams satisfying the features of a tool will be counted as drawn by such tool. Since it is impossible to manually check all of the images, we choose some samples through a stratified sampling. We divide the projects according to the number of UML diagrams used into 10 groups. For each group, we sort the projects by their LOCs decreasingly. We select the projects from top to bottom with a random step from one to five, then we select a random diagram from the projects. We keep selecting diagrams until the project's LOC reaches 10,000. To eliminate the diagram demos which only contain one or two rectangles, we check if each selected diagram contains at least three semantic elements. Eventually, we select 200 diagrams from all categories of projects. We use these samples to identify the involved tools. The result is shown in Table 4. We can see that the major tools are StarUML, EA, and Rational Rose. Some other tools are also used, such as ArgoUML, Visio, Papyrus, etc. We put these tools out the scope of this study since they are used rarely and we wish to propose an approach to processing the images built in the commonly used tools.

Table 4
Distribution of UML tools in collected image diagrams.

Tools	Ratio
StarUML	35%
EA	31.5%
Rational Rose	25%
Other	8.5%

3.2. Design model diagram usage in industry

GitHub is a software development platform. It is artifact-based in the sense that it stores code, bugs, and pull-requests, among others. The design model usage in daily works may not be reflected on GitHub, for example, *the diagram types that practitioners use during software development and routine discussion, or the ways that design models are stored.*

To answer these questions, we design a questionnaire with the following two goals. (a) We aim to figure out whether or not the design model diagrams play an important role in engineers' daily practice. And (b) we aim to elicit the most commonly used design model diagrams.

To achieve these goals, we design several questions[4] based on the following points:

- What are the presentations of the design models during the engineers' routine work? As far as we know, design models may be presented in various ways, such as UML diagrams and non-UML wireframes. We wish to find the most commonly used diagrams in industry, and figure out whether the usage is similar to open-source communities.
- How do practitioners use design model diagrams to check consistency between designs and source code? What are their opinions on the diagram usage? Design model diagrams can be used manually or with tools. The automation of usage may also depend on the diagrams' storage form, tools' functionality, or other aspects. We wish to find the concerns during design model diagram usage, and figure out what can be done to be helpful.

We distribute about 200 copies of our questionnaire among different groups of people. They include: industrial practitioners who are from various IT companies (such as Alibaba, Baidu, etc.) or workplaces (such as research institutes) in different positions; the academic researchers on software engineering from USA, Sweden and China in Beihang University; graduates with CS master or doctoral degrees. The reason that we choose these people is as follows: IT practitioners have plenty of experience on the system or feature design/development/evolution. Researchers in software engineering, especially software architecture, may have accessed some models for case study or data collection. They

[4] The questionnaire is posted at: https://gitee.com/chiefeweight/uml-code-trace.

may also read design models during paper review or project review. We choose the CS graduates who satisfy at least one of the following conditions: (i) the graduate should have participated in at least one project which can be from the industry with his/her tutor, or from his/her supervisor in the university; or (ii) the graduate should have worked as an intern for a company. Thus, they should know how to read and draw design models, especially in their assignments. These people can provide some insights for the survey considering their experience and knowledge.

We received 44 answers. The distributions of their workplaces and positions are shown in Figs. 6(a) and 6(b). The most relevant questions to this study are Question 1.1.1, 1.1.6, 2, and 2.1 (as listed below), while other questions are either premises or extensions. Question 1.1.1, 1.1.6, and 2.1 are multiple choice questions. Question 2 is a single-choice question.

- **Question 1.1.1**: If design models do exist, then what are their forms of presentation? (multiple-choice)
- **Question 1.1.6**: How do you check the consistency between design model diagrams and source code? (multiple-choice)
- **Question 2**: When you begin to work on a legacy project, do you think design models can help you understand the source code? (single-choice)
- **Question 2.1**: If you wish to have design model diagrams to help you understand the source code, then what level of the diagrams would be enough? (multiple-choice)

The statistics of these questions are shown in the sub-figures of Fig. 6.

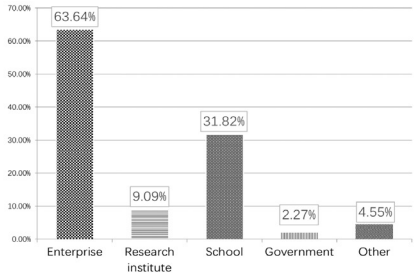
Fig. 6(c) shows the forms of design models' presentation. Since a practitioner may use more than one form of design model, the sum of the answers' percentages may exceed 100%. Among all these forms of presentation, 46.43% design models present as UML class diagrams. We can still calculate the ratio of UML diagram usage by summing up all the UML diagram usage percentages, and then divide it by the sum of all percentages in Fig. 6(c). It shows that 67% forms of design models' presentation are UML diagrams. This indicates that UML is still used most in industry.

Fig. 6(d) shows that 57.14% participants perform the consistency checking between design model diagrams and source code manually, while 54.55% of the practitioners think that the design model is helpful since the source code can be understood better during this process. This shows that practitioners may need the semantic information provided by design models for daily works. However, the manual usage of design models requires huge efforts. We think that the UML diagrams' semantic recognition can help reduce the human efforts of design model usage.

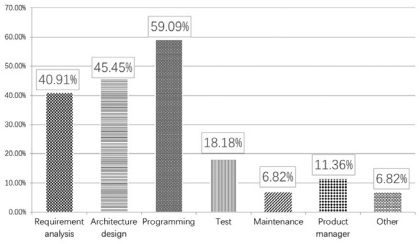
Fig. 7 shows the level of diagrams that practitioners wish to use. Since one practitioner may have multiple wishes, we illustrate the ratio of each answer in Question 2.1. 50% of the practitioners wish to use detailed design models (such as class diagrams) that could be traced to source code. This motivates our selection of diagrams which contain classes that explain the structures of projects.

3.3. Summary

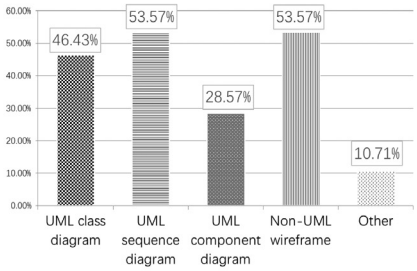
To figure out the current status of design model diagram usage among open source communities and industry, we conduct a survey on the open projects in GitHub and have distributed questionnaires among CS-related researchers and engineers. We conclude that design model diagrams are still used by plenty of projects among both open source communities and industry. Moreover, the analysis results of our collected data show these conclusions:



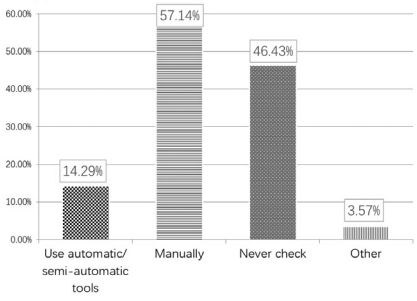
(a) Distribution of our partitioners' workplaces.



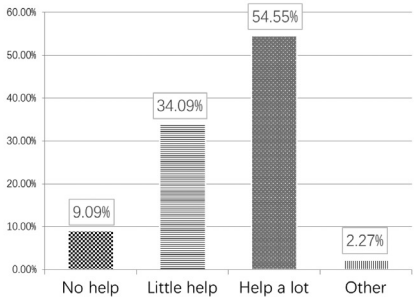
(b) Distribution of our our partitioners' positions.



(c) Forms of design models' presentation.



(d) How to check consistency between design model diagrams and source codes.



(e) Practitioners' opinion on design model's helpfulness.

Fig. 6. Statistics of the most relevant questions.

- According to the survey on open source communities, design model diagrams are usually stored as images (73.72%), which is inconvenient for (semi-)automated usage and modification. This may be the primary reason that, in practice, 47% consistency checking is done manually.

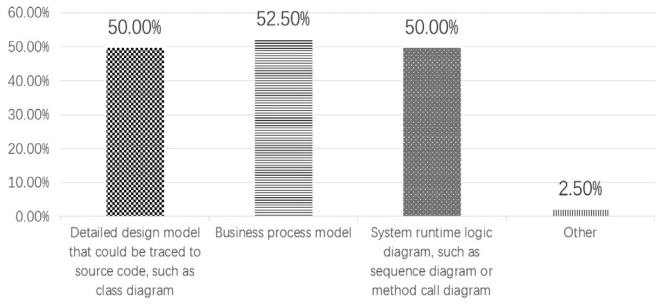


Fig. 7. Level of diagrams practitioners wish to use.

- 67% of the design models are presented as standard design models, and UML class diagram is the most important one, with a ratio of 24% among all forms of design models' presentation.

Based on these conclusions and the state-of-art of class diagram recognition, we propose an improved approach to cover more cases in practice.

4. Approach on the semantic element recognition of class diagram images

In this section, we propose an approach, ReSECDI, for the ***Re***cognition of ***S***emantic ***E***lements in ***C***lass ***D***iagram ***I***mages drawn by different tools and stored in different resolutions. We firstly give the overview procedure of ReSECDI in Section 4.1. Then each of the steps are given in detail.

4.1. Overview of the ReSECDI

The overview of our approach is shown in Fig. 8. The input of our approach is a UML class diagram saved in a commonly seen picture file format, e.g., PNG, JPEG, etc. The output is called a semantic design model of the original class diagram image, which is a text file containing most of the semantic information in the class diagram, including class names, attributes, methods, and relationships. We recognize four types of relationships that are commonly seen in practice: generalization, realization, dependency, and aggregation.

Our approach mainly contains two parts (which involves five steps as shown in Fig. 8): (1) The recognition of classes, including the recognition of class rectangles and texts in classes (i.e., class name, attributes, and methods). (2) The recognition of the relationships between classes, including the recognition of relationship lines and relationship types, and the integration of relationships and classes.

Since the qualities of images may not be good enough for recognition, we perform image pre-processing before the above two recognition phrases to eliminate the noise of the input image, such as the points near relationship lines, short segments near class rectangles, etc.

After the pre-processing, we recognize the semantic elements in the following order: We recognize the class rectangles firstly, since they are the most significant elements in the class diagrams, and they are the basics of the relationship detection. Considering that a class may contain one to three small rectangles makes the recognition of classes' contents difficult, we propose a method to cluster the rectangles into classes, with the semantic order considered (which means that the clustered rectangles follow the order of name, attributes, and methods within a class).

After the rectangle recognition, we wipe all rectangles out and recognize the relationship lines. We erase the rectangles because the lines of rectangles can impact the recognition of relationship lines. Since current approaches have limits on the recognition of polygonal lines, we propose a polygonal line merging method based on the point-polygon-test technique for recognizing oblique polygonal relationship lines in class diagram images, and integrate the lines with classes so that a relationship gains the source class and target class. After the lines are recognized, we recognize the relationship type symbols, and combine the symbols with relationship lines in order to get complete relationships. The symbol recognition refers to a method aiming at recognizing different types of symbols in various qualities (for example, the type symbol of generalization appears to be a nonstandard triangle). The last step is the text recognition, which recognizes all texts in all classes, and combines the texts with class rectangles to get complete classes. After all semantic elements are recognized, we generate a file that records all the recognition results, which represents the semantic model of the input diagram. We provide a git repository project,[5] which contains an implementation of ReSECDI.[6] and the diagrams for evaluation[7] The *ClassDetector*[8] handles the class recognition. The *ClassRelationDetector*[9] handles the relationship recognition.

In the following text, we explain our approach step by step.

4.1.1. Image pre-processing

Since the input images are of different qualities, most likely containing a lot of noise points, we pre-process the images by filtering the noises which can impact the later elements recognition. This includes the gray-scale treatment (Zarándy et al., 1998), Gaussian sharpen (Machado et al., 2015; Nam and Ahuja, 2012), dilation and erosion (Deng and Huang, 2017; Ya-jie, 2012).

First, when we process an image's pixels using the RGB model, we should be aware of the fact that each pixel may have different values of the R (red), G (green), and B (blue), which form the pixel's color. However, our recognition job is focused on the locations and sizes of shapes, lines, and texts. The colors of graphical elements do not provide any useful information for the semantic recognition. Moreover, the gradient colors may affect the recognition of a class. So firstly we use the gray-scale treatment to convert each pixel's RGB values into a single value. For each pixel, the gray-scale treatment calculates the average of its RGB values, then changes all RGB values to the average. The resulting image still contains essential semantic information (such as the shape and size) for the following recognition.

Second, the input image may be in low resolution, which implies that the semantic elements in an image may be blurred or with unnecessary noise points. In order to make the image clear enough for recognition, we choose the popular image sharpening method, Gaussian sharpen (Machado et al., 2015; Nam and Ahuja, 2012), to optimize the sharpness of an image. This sharpening method can make an image clearer by enhancing the edges with the Gaussian kernel. It performs well for the sharpening of rectangles and lines. Then, there are still many noise points that need to be removed. The noise points that are near the semantic elements in a class diagram impact the recognition mostly. In order to remove those noise points, we choose the morphology operations: erosion and dilation (Deng and Huang, 2017; Ya-jie, 2012). The erosion expands the width of edges (including edges of rectangles, and relationship lines) so that the noise points can be covered, and then the dilation shrinks the width of edges in order to keep the original sizes and shapes of graphical elements. Once

[5] https://gitee.com/chiefeweight/uml-code-trace.
[6] Under "/uml-code-trace/src/main/java/com/hy/java/uct/umlrecog".
[7] Under "/uml-code-trace/src/main/resources/cd".
[8] Under "./umlrecog/cddetector".
[9] Under "./umlrecog/cddetector".

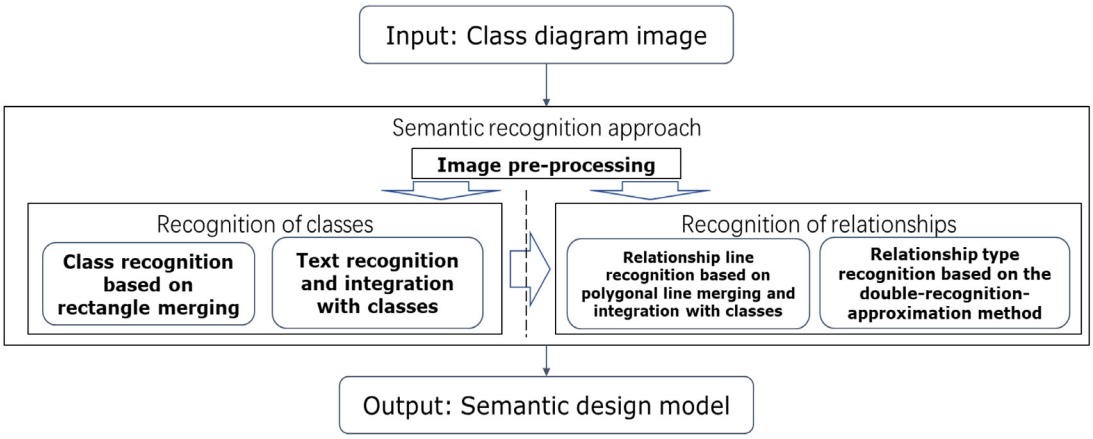


Fig. 8. Overview of class diagram recognition approach.

the image pre-processing is finished, the resulting image will be clearer with fewer noise points than before, and we can use it for the following recognition.

4.1.2. The recognition of class rectangles

The most important thing for recognizing a class is to find the rectangle. Once the rectangles of all classes are determined, the text recognition can be done for each rectangle. The source and target classes of each relationship can also be determined based on the locations of classes. Since the text recognition for each class can only be done after the classes are recognized, in this step, we just explain the detection of classes. The text recognition will be introduced later.

Since the OpenCV image processing library provides efficient and accurate APIs for shape recognition (Xie and Lu, 2013; Chandan et al., 2018), we choose this library for detecting the rectangles in the image. However, the OpenCV can only recognize rectangles in an image without any UML-specific semantics, which means that the OpenCV may recognize all rectangles including the whole classes' rectangles and the small rectangles forming classes. These rectangles spread all over the image without any hints of how they cluster into classes. In order to find the right small rectangles for each class, we propose a method to merge the recognized rectangles into classes.

The rectangles of a class have some constraints: (a) a class may contain one to three small rectangles; (b) a class must contain a rectangle with its name in it; and (c) the order of the small rectangles of a class should be name, attributes, and methods from top to bottom.

We show five examples of classes in Fig. 9. We can see that the class "DB" has some attributes and methods, thus the whole rectangle of this class is divided into three parts: the top rectangle containing the name, the middle rectangle containing the attributes, and the bottom rectangle containing methods. Some classes may not contain attributes or methods, thus their rectangles can be divided into less than three small rectangles. Since the rectangle detection may detect all kinds of rectangles, including the whole rectangles or the small rectangles, we should use a merging method to merge the small rectangles into whole class rectangles.

Our approach on the class rectangles is shown in Fig. 10. The basic principle is that we first detect all candidate rectangles of classes, then perform a fuzzy clustering by grouping the possible rectangles, and finally delicately detect the single classes from the clusters. Briefly, three steps are involved: rectangle recognition, rectangle cluster, and class content detection.

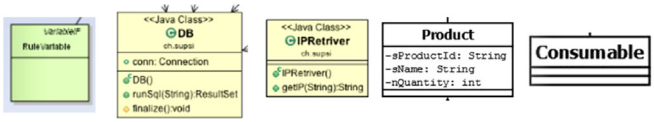


Fig. 9. A class may contain one to three small rectangles.

The purpose of rectangle recognition is to detect all closed contours in the image, i.e., the candidate rectangles. This is performed by the polygon approximation provided by the OpenCV library for the contour detection. It can detect all sizes of closed contours, including whole class rectangles and small rectangles. Then, we need to cluster the rectangles into classes.

The purpose of rectangle clustering is to roughly find the groups of rectangles, i.e., the classes which may be composed of one to three rectangles. However, we cannot directly know the rectangle combinations of single classes, due to the variable distances between classes and the varied number of rectangles in classes. Therefore, we design a rough rectangle cluster step. The idea is actually intuitive. We randomly take a rectangle as the candidate rectangle, then check the coordinates of the left-top points of the candidate rectangle and all other rectangles. If the difference of two rectangles' left-top points' horizontal coordinates is less than 5 pixels, and if their widths are similar, then we cluster them into one pile. We do this process iteratively until all rectangles are checked and clustered into a pile. After this step, the rectangles in each pile are "horizontally close to each other", which means that the differences between the horizontal coordinates of each rectangle's left-top point are all less than 5 pixels, and all these rectangles have similar width. Since the small rectangles that belong to the same class are also "horizontally close to each other", our method can cluster them into the same pile. However, since different classes may have similar horizontal location and width, this step may falsely group the rectangles which are "horizontally close to each other" but "vertically belong to different classes". Thus, we need to distinguish the rectangles in each pile in order to recognize each class.

The purpose of *class content detection* is to delicately detect the constitutions of single classes, since the rough rectangle clustering based on horizontal distance threshold may cause lots of negative grouping, i.e., falsely grouping the rectangles together, which belong to different classes. However, sometimes the rectangles overlap, making the missing of some rectangles, just as the ones shown in the right top zone of Fig. 10. In this case, the recognized rectangles are the whole rectangle of this class, the upper rectangle containing the name and attributes, and

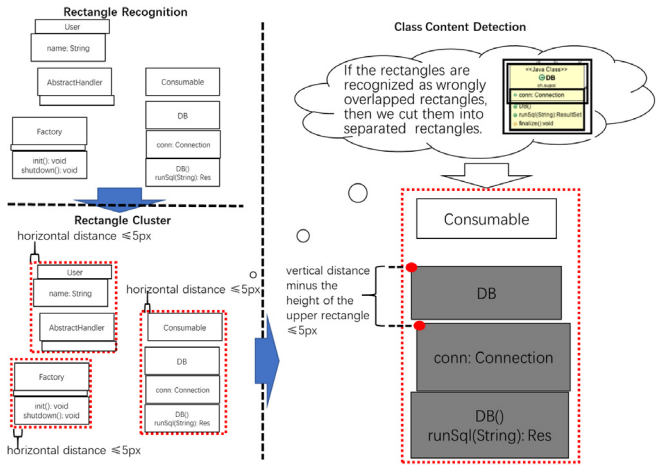


Fig. 10. Class recognition of our ReSECDI. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

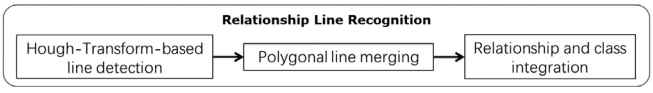


Fig. 11. Relationship line recognition.

the lower rectangle containing the attributes and methods. The constitutions of these classes are mixed. Thus, we first cut the small rectangles based on the constraints of the UML class. Otherwise, since the text recognition of each class is based on the constitutions (for example, we recognize the text of Class name in the top rectangle), if we do not cut these mixed rectangles following UML constraints, the recognition of texts within this class will be disturbed.

For example, in the case shown in Fig. 10, the recognition of a name is based on the upper rectangle. Since it contains both the name and the attributes, the recognition will be the mix of the name and all characters of the attributes. To address this situation, we first check whether a rectangle's left-top point is within another rectangle, which means these two rectangles are overlapped (e.g., the top small rectangle and the bottom small rectangle are overlapped in the attribute area). We cut the overlapped areas to create two separated rectangles, and only one of them contains the overlapped area. Then we can merge the separated rectangles into one class rectangle. To do this, for each pile, we check if the difference of two rectangles' left-top points' vertical coordinates minus the height of the upper rectangle is less than 5 pixels. If so, we merge these rectangles into one class. By keeping the above merging step until all rectangles are checked, we are able to merge all small rectangles into classes that may contain one to three small rectangles.

The reason that we set a threshold for merging rectangles is that we admit the inevitable pixel error in images, even though the image is pre-processed. For instance, two little rectangles which should be aligned may not be, with a dislocation of a few pixels. In fact, we find that the dislocation errors are always within the width of a line, which are about three to eight pixels. If the threshold is set too small (e.g., less than three), the rectangles belonging to one class may be divided into multiple ones, thus these rectangles cannot be correctly merged (in our case, more than 90% of the small rectangles are not merged). However, if the threshold is set too big (e.g., greater than eight), the rectangles belonging to different classes may be wrongly merged. In our work, we choose the average value (rounded down, i.e., five px) as the threshold, and it turns out that this value performs well on most cases.

4.1.3. Relationship line recognition

In a typical UML class diagram, relationship lines can be solid or dashed lines, straight or polygonal lines. Current line detection algorithms (such as the hough transform Illingworth and Kittler, 1988; Hassanein et al., 2015; QIN et al., 2010) can detect all straight solid lines or dashed lines. However, a relationship line may be a polygonal line. The existing line detection algorithms can only detect straight segments of a polygonal line. Previous studies (Karasneh and Chaudron, 2013a,b; Hammond and Davis, 2006) on UML diagram recognition cannot recognize the oblique polygonal line, which actually occurs a lot in practice. So, we need to find a way to merge the oblique segments into one complete polygonal line. Moreover, since the input image may be in low-resolution where noise points are introduced, the conditions caused by noise points need to be addressed in order to recognize polygonal lines. Thus, we have to resolve two problems in this step: (i) the oblique polygonal line recognition problem, and (ii) the noise points problem caused by low-resolution.

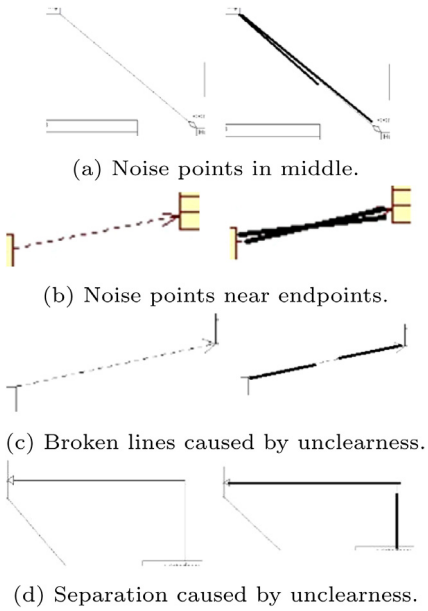
We propose a method to address the oblique polygonal line recognition problem under the circumstance of noise. The whole relationship line recognition step is divided into three parts (as shown in Fig. 11): straight line detection, polygonal line merging, and the relationship target integration.

The first part is for straight line detection. Here, we refer to straight lines as all the line segments without any turns, including direct lines between two classes, and the segments of polygonal lines. Although the rectangles are also formed by four edge lines, we have already wiped them off the image, so the rectangles will not impact this step anymore. The straight lines in a class diagram image include solid lines and dashed lines, which can all be recognized by the hough transform technique (Illingworth and Kittler, 1988; Hassanein et al., 2015; QIN et al., 2010). The hough transform detects all "edges" in an image, and selects all lines satisfying the length threshold. In our approach, we set the threshold to 0.05 of the image's diagonal line, which means all "edges" found by the hough transform technique which are longer than this threshold will be considered as a straight line. If the threshold is selected too big, then many short lines will not be considered as "lines satisfying the threshold". If the threshold is too small, then there will be too many short lines (or even groups of points) considered as "lines satisfying the threshold". We argue that the current selection of threshold is empirical, and the limitation will be discussed in Section 5.4.

Since we cannot assure that all noise points are removed in the pre-processing step and class-wiping step, the line detection algorithm may recognize some lines that are actually not relationship lines. Thus, we need to filter out the true relationship lines. We can use the location information of classes to do the filtering. For each candidate straight line, we reach its two endpoints by five pixels, and then traverse all rectangles of classes to find out which rectangle contains the reached point. If there exists a class rectangle that contains the relationship line's reached endpoint, we record this line as a relationship line.

The first part will result in two situations: (a) All relationships that connect two classes with direct lines will be recorded; and (b) All relationships that connect two classes with polygonal lines will be found with two "end lines", where one point is connected with a class, and the other point is waiting to be merged with segments. The second part is to deal with this situation.

The second part is polygonal line merging. Since in the first part, we have already obtained the "end lines" of polygonal lines, what we need to do next is to find the segments which have an endpoint connecting with the "end lines", and then iteratively



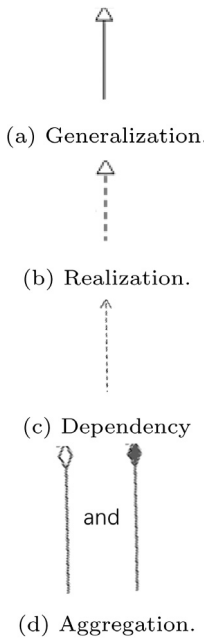
(a) Noise points in middle.

(b) Noise points near endpoints.

(c) Broken lines caused by unclearness.

(d) Separation caused by unclearness.

Fig. 12. Different situations caused by different noise.



(a) Generalization.

(b) Realization.

(c) Dependency

(d) Aggregation.

Fig. 13. Four types of relationship we aim to recognize.

add the found segment into the polygonal line. The general technical way is simple, but it actually encounters many unexpected situations. These situations are caused by various types of noise in the diagram, and different noises may cause different situations, so they are hard to handle.

During the manual analysis of the sample class diagram images, we generally find four types of situations:

- Noisy points in the middle of lines.
- Noisy points near endpoints of lines.
- The image may be blurred, causing the hough transform technique wrongly recognizing one straight line into several broken lines.
- The blurring images increase the difficulties of recognizing the polygonal lines (the connections of several straight lines).

There might be other situations impacting the polygonal line merging, however, they do not occur as frequently as these four situations. In our work, we only address these four situations that occur most as the major barriers to the polygonal line merging. We show the examples taken from our collection of UML class diagrams (Figs. 12(a)–12(d)), and explain how we handle these situations. We can see that these images from practice have low resolutions, thus contain many noise points. The left part of each figure of the situation is the original part of an image, and the right part shows the recognition flaws that need to be addressed. The first and second situations (Figs. 12(a) and 12(b)) happen when there are some noise points near a line, so the hough transform technique will detect two lines instead of one. The third and fourth situations (Figs. 12(c) and 12(d)) happen when the line is not clear, and the line detection will detect two separate segments instead of a whole line or polygonal line. The first and second situations are dealt with line gradient and length check, which basically checks whether or not two lines coincide with each other. The third and fourth situations are dealt by reaching the endpoint of each line to check whether two segments can be merged.

The third part is the relationship target integration. The relationship target means the classes to which the relationship belongs, including the source class (from where the relationship comes out) and the target class (where the relationship comes in). By reaching the endpoint of each polygonal line and checking whether a class rectangle contains it, we are able to integrate all relationships with their source and target classes.

4.1.4. Relationship type recognition

In this step, we aim to recognize the relationship types. Then we can match the relationship types and the relationship line to form the integrated relationships. We recognize four types of relationships which are commonly seen in class diagrams: generalization, realization, dependency, and aggregation. Their graphical elements are shown in Fig. 13. The relationships of generalization and realization have the same representing shape (the triangle), the way to distinguish them is to check whether the relationship line is solid or not. A triangle relationship type shape with a solid relationship line represents generalization, while a triangle with a dashed line represents realization. The dependency relationship consists of an arrow and a dashed line. The aggregation relationship consists of a diamond and a solid line. We only detect directed relationships which are used by the current consistency checking approaches. However, since the line detection part of our approach can detect the dashed lines without arrows, our approach can be adapted to detect the associations without arrow head.

Intuitively, the relationship types recognition depends on the recognition of the relationship type symbols. Since the AForge.NET and Point-Polygon (El-Salamony and Guaily, 2020) techniques provide the shape checkers which can detect triangles, arrows, and rhombus, we choose these techniques for the relationship type symbol recognition. However, it encounters a major problem that the relationship type shapes are usually small and inaccurate. For example, a standard rhombus should have exactly four edges with the same length, while the image stored in low-resolution may have "a pentagon that looks similar to a rhombus". In fact, many relationship type symbols are some shapes that are similar to the standard UML-relationship-type symbols (the triangles, arrows, and rhombus), however, with some noise points or edges caused by the low-resolution. So the direct usage of the existing techniques may cause false recognition. To address this problem, we propose a double-recognition-approximation method.

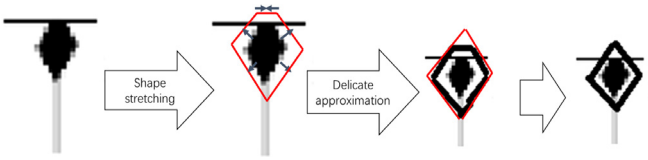


Fig. 14. The double-recognition-approximation method (taking the aggregation symbol as an example). Note: First we stretch the shape into a pentagon, then we delicately approximate the pentagon into a rhombus.

The principle of this method is simple. Since the type symbols are the shapes similar to the standard UML-relationship-type symbols but with some flaws, we can slightly "stretch" the type symbols, then tightly "shrink" so that the flaws can be expected to be reduced and the recognition should be more accurate.

This method refers to the following two steps: First, we locate the relationship type shape from the images which can be retrieved from the OpenCV shape detection API (Xie and Lu, 2013; Chandan et al., 2018). We use the minAreaRect() method provided by OpenCV to get the minimum rectangle containing the area of the located relationship type shape and cut it from the original image. Second, we recognize the relationship type shape within the cutted image using approximation parameters (e.g., edge stretching length, approximation threshold), and approximate the recognition result to a standard UML-relationship-type symbol. The approximation requires two sub-steps (shown in Fig. 14):

(1) We reduce the flaws of the type symbol. Currently, the symbol is still similar to the standard UML-relationship-type symbol in general, just with a few noise points or short noise edges. If we stretch out the long edges of the symbol, then the main shape will be expanded, and the number of noise points or length of noise edges can be reduced. Taking the aggregation symbol as an example, we stretch the longest four edges of the symbol. After this step, the shape of the symbol should be more accurate than before for the recognition.

(2) We delicately approximate the roughly accurate shape to the standard UML-relationship-type symbol. Since the edges of the roughly accurate shape may not be the same with the standard symbol, there may be some distances between the edges of the original shape and the approximated shape. These distances determine the approximation result. In our case, we set a threshold to control the approximation: the longest distance between the edges of the original shape and the approximated shape. We set this threshold to 0.01 of the original shape's girth. Since the first step of the approximation eliminates many noise points and edges, the impact of these flaws on the delicate approximation are reduced. Thus, our delicate approximation performs better than using the existing approximation techniques directly.

After the recognition of shape, we integrate the results with relationship lines, in order to get a complete relationship. The integration method is similar to the relationship target integration, where we check whether the minimum rectangle containing the area of the relationship type shape also contains the endpoints of relationship lines.

4.1.5. Text recognition

In this step, we recognize all the texts in classes, including the names, attributes, and methods of classes. The attributes and methods include letters and modifier characters (such as "+" for "public", "−" for "private", etc.). Since the Tesseract-OCR (Smith, 2007) library (an Optical Character Recognition engine maintained by Google) can reach 90% accuracy on English characters (Smith, 2013), and over 95% accuracy on the modifier characters (Hoshen and Peleg, 2016), it is capable of recognizing all the referred characters in this work.

In order to correctly combine the texts into classes, we recognize the classes' texts one by one, under the following steps:

Firstly, since the name, attributes, and methods of a class are in different small rectangles, we need to cut the class into separate rectangles in order to recognize the texts correspondingly. For each small rectangle in a class, we create a temporary image only containing the small rectangle, then we can recognize the texts in the temporary image. The separation of small rectangles within a class follows such order: if the class consists of one rectangle, we cut it for the recognition of the name. If it consists of three rectangles, we cut each of them for the recognition of name, attributes, and methods. If it consists of two rectangles, we cut the top rectangle for the recognition of name, and the bottom rectangle for the recognition of attributes temporarily, since the texts in the bottom rectangle may actually be operations. We use the cut rectangle to create a new image, which only contains the texts within it (the edges of the rectangle are removed).

Secondly, for each new image containing the corresponding small rectangle, we use the Tesseract-OCR library to recognize texts. For the classes consisting of one or three small rectangles, the text recognition results can automatically bind with each corresponding small rectangle, since the constitutions of the UML class follow the constraint presented in Section 4.1.2. For the classes consisting of two small rectangles, we need to check whether the texts in the bottom rectangle are attributes or methods. Since the texts of methods follow the pattern that the name of the method should have a pair of parentheses right after it, we check the texts in the bottom rectangle to find out whether they can match the pattern. If we find at least one matched sequence of characters, we record the bottom rectangle as the methods. Otherwise, it remains the attributes by default.

After this step, the whole class diagram recognition is finished. We record the recognition result of one class diagram in a text file, which contains the information of all recognized classes and relationships.

5. Evaluation

5.1. Research questions

To evaluate our approach, we design two research questions:

RQ1: To what extent can ReSECDI automatically recognize the semantic elements from UML class diagrams images built with the most popular modeling tools?

Considering that the UML class diagrams constructed with different UML modeling tools vary in the graphical elements presentation, we would like to evaluate the performance of our approach on processing the images from the popular tools. According to the survey in Section 3, we focus on StarUML, EA and Rational Rose.

RQ2: To what extent can ReSECDI automatically recognize the semantic elements in UML class diagrams images from open-source communities?

There are more problems in the images of open-source projects than those specifically created with the popular tools, for RQ1, such as the low-resolution problem and multi-style problem. In order to test the generality of our approach, we would like to evaluate its performance on processing the UML class diagrams images randomly selected from open-source communities.

5.2. Addressing RQ1

5.2.1. Experiment design

Currently, the main approach for semantic recognition of class diagram image is Img2UML (Karasneh and Chaudron, 2013a). It claimed that Img2UML can detect classes, straight line relationships and the texts. However, we cannot replicate their work since we cannot find the related source code or tool, and the details in their publication are not enough. The data set provided in their work (Karasneh and Chaudron, 2013a) cannot be retrieved, either. Given that it is designed for processing the images drawn in StarUML, and the experimental results on 10 diagrams are given in their publication, we directly use their results as the evaluation of this tool.

To address RQ1, we re-draw these 10 images in the three popular UML tools: StarUML, EA, and Rational Rose. This gives us a total of 30 diagrams for the experiment. Overall, the 10 diagrams contain 97 classes and 86 relationships. We redraw all of them with the three tools respectively, making 30 diagrams with a total of 291 classes and 258 relationships. StarUML is selected because it is popular and also the work with Img2UML performs the evaluation on the images drawn by this tool. EA is selected because it is the most popular UML modeling tool according to Ozkaya et al.'s (Ozkaya and Erata, 2020) survey. According to our walk-though on the class diagrams in the open-source community, Rational Rose is also very commonly used. We would like to get the measurements of our approach on these three sets of images.

For the measurement, we are interested in the overall precision and recall of the class and all relationship recognition. Besides, we also would like to know the extent to which our approach can detect each kind of the relationships, since they are the basis of the inconsistency checking tasks.

5.2.2. Measurements

Based on the automatic recognition results and manual verification, we use recall and precision to measure the performance of the approach.

We record both the manually annotated results and the automated results, including the number of classes, generalizations, realizations, dependencies and aggregations in each class diagram. Let the number of manually annotated elements in selected cases be cls_a, gen_a, rlz_a, dpd_a, and agg_a, the number of all recognized elements as cls_r, gen_r, rlz_r, dpd_r, and agg_r, and the number of the correctly recognized elements be cls_{cr}, gen_{cr}, rlz_{cr}, dpd_{cr}, and agg_{cr}. Precision and recall are calculated as (1), taking the class as an example (others are similar):

$$recall_{cls} = \frac{cls_{cr}}{cls_a}, \quad precision_{cls} = \frac{cls_{cr}}{cls_r}. \tag{1}$$

Note that if a class is correctly recognized that means that all the related information of this class is identified correctly, including the class name, attributes, and methods (i.e., the name and modifier).

5.2.3. Results and analysis

The results of Img2UML on StarUML diagrams are shown in Table 5. They recognize the class, relationship line, and relationship type, and the precisions are 100%, 97%, and 85% respectively. The detailed results on each relationship type are not provided in their work, while we list the detailed results for our approach. The results on three selected tools are listed in Tables 6–8, respectively.

Table 5
Results of Img2UML on StarUML diagrams according to Karasneh and Chaudron (2013a).

	Class	Relationship line	Relationship type symbol
Precision	100%	97%	85%

Table 6
Evaluation of ReSECDI on StarUML diagram images.

	Recall	Precision
Class	98.96%	100%
Relationship	Overall 87.35%	Overall 89.47%
Detail	Generalization: 87.50% Realization: 100% Dependency: 86.79% Aggregation: 75.00%	Generalization: 92.85% Realization: 90.00% Dependency: 89.13% Aggregation: 83.33%

Table 7
Evaluation of ReSECDI on EA diagram images.

	Recall	Precision
Class	100%	96.03%
Relationship	Overall 93.42%	Overall 89.90%
Detail	Generalization: 100% Realization: 100% Dependency: 88.67% Aggregation: 71.42%	Generalization: 86.36% Realization: 90.00% Dependency: 87.23% Aggregation: 80.00%

Table 8
Evaluation of ReSECDI on Rational Rose diagram images.

	Recall	Precision
Class	97.93%	97.89%
Relationship	Overall 90.80%	Overall 87.87%
Detail	Generalization: 93.75% Realization: 100% Dependency: 84.90% Aggregation: 75.00%	Generalization: 86.66% Realization: 84.61% Dependency: 88.88% Aggregation: 83.33%

From the evaluation on the StarUML diagram images in Table 6, we can see that the recall and precision of the class recognition are 98.96% and 100%. Table 7 shows that the class recognition on EA images has a recall of 100% and a precision of 96.03%. Table 8 shows the evaluation on Rational Rose images has a recall of 97.93% and a precision of 97.89%. We checked the classes that are not recognized by ReSECDI, and find that they either have too many attributes (about 30 or more) but no methods (the method rectangle is empty), or have too many methods (about 30 or more) but no attributes (the attribute rectangle is empty). We find that when the sizes of small rectangles within a class vary a lot, i.e., the size of the attributes' rectangle is over 30 times bigger than the size of name or methods' small rectangles, the small ones are usually considered as part of the bigger one. This impacts the text recognition, e.g., the name cannot be recognized, and eventually it causes the loss of recall. However, the information of recognized classes is correct with a precision over 90%. We believe that this is mainly due to StarUML's diagram style, i.e., its default font presents big and clear letters and modifier characters which are good for recognition.

For the relationship recognition on StarUML diagram images, the overall recall is 87.35% and the precision is 89.47%. If we focus on each of the relationship types, we can see that the recognition of *realization* is the best, with a recall of 100% and a precision of 90.0%, and the recognition of *aggregation* is the worst, with a recall of 75.0% and a precision of 83.33%. We should note that the overall result on the four types of relationships is calculated after their type symbols are determined, so we can compare this result with Img2UML. Img2UML's result on relationship type symbols

is 85%, while our overall result on the relationship recognition is 87.35%. So both Img2UML and ReSECDI perform well on the relationship recognition, while the precision of ReSECDI is just a little higher. For the other two tools, the overall performance on relationships recognition on EA images is about 93.42% of recall and 89.90% of precision, and 90.80% of recall and 87.87% of precision on Rational Rose images. We can also find a similar phenomenon with StarUML images: The performance of *Realization* relationship recognition are the best. The performance on other relationships recognition is moderate and great with a recall of 84.0%–100% and a precision of 84.61%–92.85%. These results show the generality of our approach on different UML modeling tools.

We dig into the images and would like to explore the reason for the performance of our approach on these semantic elements.

For classes, a good value of recall has been achieved on the images of EA, while there are some losses with the images of Rational Rose and of StarUML. The style of *Class* on Rational Rose is similar with StarUML, which also has small empty rectangles when attributes or methods are empty. The images of EA do not have the small empty rectangle. When a class of EA only has name and attributes, it just consists of two rectangles. Also, the default size of the name's rectangle in EA is big. So, when we draw a class with over 20 attributes and no methods in these tools, StarUML and Rational Rose will draw a *Class* consisting of one big rectangle and two small rectangles (the default size of the empty rectangle is similar to that with only a name), while EA will draw a *Class* consisting of one big rectangle and one not-too-small rectangle with its name. Thus, the recognition of images on both StarUML and Rational Rose are affected by various sizes within classes, while EA is not.

However, the information of recognized classes in EA and Rational Rose are not as correct as for StarUML. For EA, we find that the modifier characters are so small, and they are far away from the modified methods. For Rational Rose, we find that some letters are slightly overlapped (maybe caused by its default font). We think these are the reasons that impact their precision.

Moreover, the recalls of realization are all 100% for the three tools. The realization relationship is composed of dashed lines and triangles. The dashed lines, especially the dashed polygonal lines, are hard to recognize for the reason that the connections of segments are often affected by low resolution as we stated in Section 4.1.3. The relationship type symbols are also hard to recognize because of the flaws caused by low resolution. The results show that both the method for addressing the noise points (in Section 4.1.3) and the double-recognition-approximation method for recognizing relationship types (in Section 4.1.4) address the dash-line relationship problem well.

The recognition of aggregation results in low recall and precision for the three tools. This is because the relationship type symbol of aggregation is more complex than other relationship type symbols, e.g., the symbol has more edges to be recognized, and the symbol has two styles of solid and hollow. In fact, the aggregation symbol is not always presented as a standard rhombus. Random loss of corners occurs frequently in real cases, which makes it difficult to recognize, even though we approximate the shape twice.

5.3. Addressing RQ2

5.3.1. Experiment design

Image selection To address RQ2, we need to collect UML class diagram images from open-source communities firstly. In order to address the diversity of the selected models, we first sort all projects that contain class diagrams by LOC decreasingly, then select class diagrams from top to bottom one by one. The reason why we sort the projects is that we believe larger projects are more likely to have complex designs than smaller ones, which may contain class diagrams with more elements. For each candidate diagram, we manually check if the diagram contains at least three classes. We iterate this process manually until we get 50 diagrams.

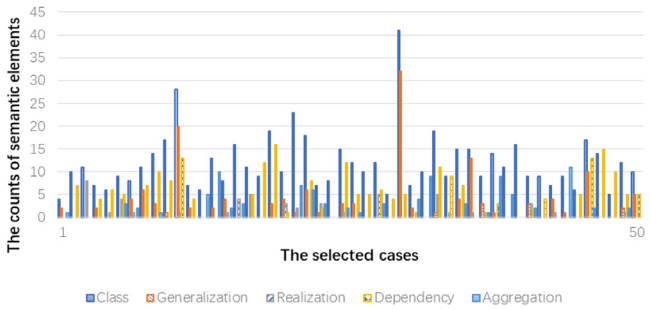


Fig. 15. The distribution of the semantic elements in the selected UML class diagram images.

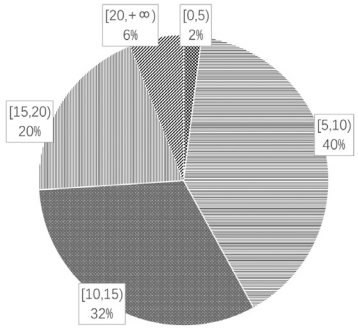


Fig. 16. Distribution of the selected diagram images by the number of semantic elements.

Table 9
Distribution of editors in the selected diagrams.

Tools	Number (Ratio)
StarUML	12 (24%)
EA	12 (24%)
Rational Rose	11 (22%)
ObjectAid	7 (14%)
Visio	2 (4%)
Papyrus	2 (4%)
IDEA Ultimate	2 (4%)
ArgoUML	1 (2%)
PowerDesigner	1 (2%)

We count the number of each semantic element (classes and different types of relationships) in all the selected cases, and illustrate the statistics in Fig. 15. The horizontal axis means the cases from 1 to 50, and the vertical axis means the counts of each semantic element. The total number of classes in all selected diagrams is 598, and the total number of relationships is 533. Fig. 16 shows the distribution of the number of semantic elements in the selected cases. If a selected diagram contains four classes and six relationships, we say this diagram contains a total of ten semantic elements. 92% of the selected diagrams contain 5 $\sim$ 20 semantic elements. We also manually identify the involved editors according to the diagram styles. The distribution of editors for the 50 diagrams is shown in Table 9. The total ratio of StarUML, EA, and Rational Rose is 70%.

Measurement We use the same metrics as those in the first experiment (Section 5.2.2).

Table 10
The evaluation of our ReSECDI on the open source diagrams.

	Recall	Precision
Class	97.15%	98.62%
Relationship	Overall 90.26%	Overall 93.64%
Detail	Generalization: 85.71% Realization: 100% Dependency: 91.73% Aggregation: 90.72%	Generalization: 91.96% Realization: 92.30% Dependency: 94.44% Aggregation: 94.38%

5.3.2. Results and analysis

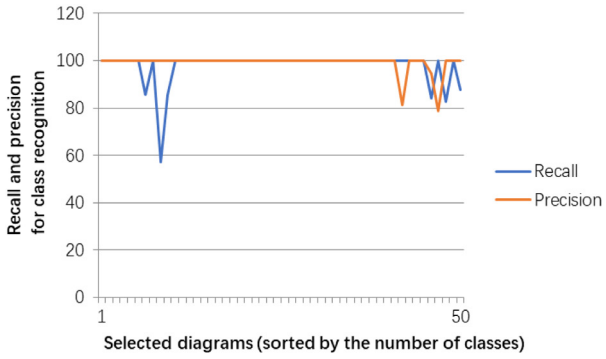
The recognition results are listed in Table 10. We can see that the recall of class recognition is 97.15%, and the precision is 98.62%. The results are all over 95%, however, a little lower than the results for StarUML. We believe the main reason causing metric loss is the various styles and resolutions of diagrams from the open-source community. Some styles are different to StarUML, EA, or Rational Rose, e.g., with background colors in the class rectangles, or blur texts, etc. The unexpected conditions can cause fault rectangle recognition and text recognition. For example, when the background color of a class's rectangle is gradually changed from light to dark, the solidness of this rectangle is also changed. The recognition algorithm may wrongly recognize it as one rectangle with light color and another one with dark color.

The overall relationship recognition recall is 90.26%, and the precision is 93.64%. In Img2UML, its precision ranges from 80% to 97% with an average of 89%. Compared to Img2UML in this scenario, although our approach cannot reach a precision as high as 97%, it has a stable performance on different diagram styles. To take a deep inspection into the results, we can see that the highest recall is realization recognition (100%), and the highest precision is dependency recognition (94.44%). These show that our approach performs well on dash-line relationships. The lowest recall and precision are all generalization recognition, which shows that our approach still needs some improvement on the solid-line relationship recognition. In fact, there are many cases where the polygonal lines of two or more generalizations are merged together. It is difficult to find the source and target class for each generalization in the merged line group. This may be a big challenge for the recognition that needs more attention in the future.

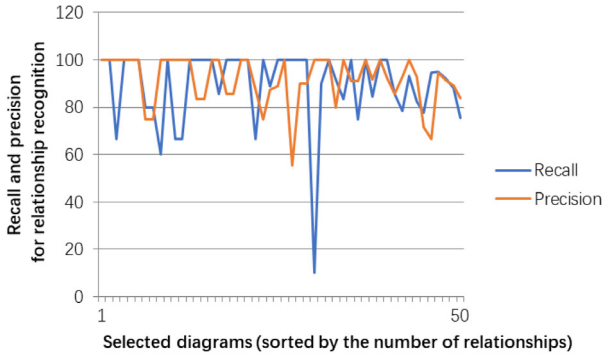
Intuitively, the more complex the model, the worse the automated recognition due to more inference. Therefore, we would like to explore the performance of our approach on processing the diagram images with different scales of classes and relationships. We analyze the overall counts of relationships rather than each type of relationship because all of them, distributed in the same space, are composed of lines and arrows with very high similarity, resulting in the interference of each other on the recognition. Besides, not all diagrams contain all of the relationship types. The statistics of the class numbers and the relationship numbers are shown in the subfigures of Fig. 17.

In Fig. 17(a), the horizontal axis means the selected 50 diagrams sorted by the number of classes (ranged from 4 to 41, as shown in Fig. 15), and the vertical axis means the values of recall and precision for each diagram. In Fig. 17(b), the horizontal axis means the selected 50 diagrams sorted by the number of relationships (ranged from 0 to 37, as shown in Fig. 15), and the vertical axis means the same as Fig. 17(a).

From Fig. 17(a), we can see that the precision of class recognition is stable, with a little fluctuation for diagrams with classes over 40. The curve of recall is similar to the curve of precision, while few cases drag down the results. The worst case is shown in Fig. 18 with a recall of 57.14%. This diagram has gradually changed colors in its class rectangle, which impacts the recognition most.



(a) The results by the number of classes.



(b) The results by the number of relationships.

Fig. 17. The evaluation results of our approach by the distribution of selected diagrams.

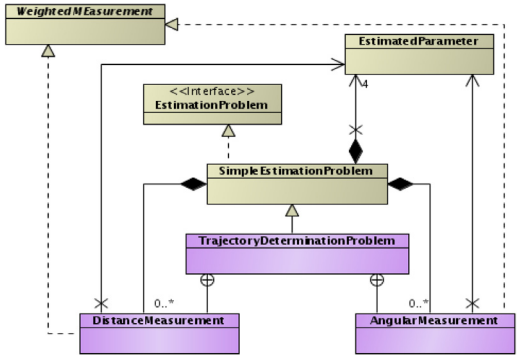


Fig. 18. The case with worst class recognition results (the "gradual color" situation). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

From Fig. 17(b), we can see that the results of relationship recognition are not as stable as those of class recognition. However, nearly half of the recalls are around 80% to 100%, while the other half of the recalls are around 60% to 80%. The majority of precisions are around 80% to 100%. The worst case is shown in Fig. 19, which has some texts and unknown symbols (such as small squares) on the relationship lines. The unexpected contents on lines can cause line gaps in the line recognition, which eventually impact the relationship recognition. It shows that the factors that impact the recognition most are some situations such as "text in line" or low-resolution problem.

As we said in Section 5.2.2, a class is correctly recognized only if all the class name, attributes, and methods are correctly

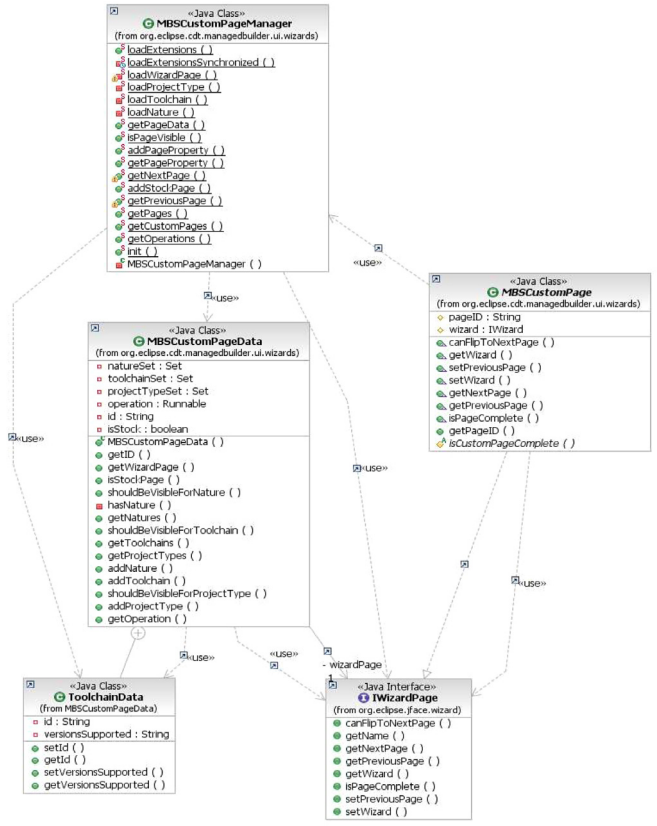


Fig. 19. The case with worst relationship recognition results.

recognized. This indicates that the results of class recognition include the text recognition. From Tables 6–10, we can see that all the recalls and precisions of class recognition are above 96%. This means that the text recognition has at least a recall and precision of 96%. Thus, the text recognition performs well. However, it has some limitations on recognizing special characters, such as "©" in the class name or "Ⓘ" in the interface name.

5.4. Discussion

5.4.1. Unexpected situations that may cause recognition errors

The procedure of our approach introduced in Section 4 has actually been optimized several times from the initial version. At first, we only consider the rectangle recognition, solid and dashed line recognition, triangle recognition, etc. During each step, we encounter many unexpected situations which cause recognition errors. Some of them are already addressed as stated in Section 4, while there are still many other situations that are hard to overcome. We list the situations found in the 50 selected cases as follows, with the number of cases that the situations occur (some cases may contain more than one situation):

(a) An oblique line is blurred, so that unexpected gaps in the line result in a wrongly recognized dashed line or not a line. (16 cases)
(b) Some lines' colors are too light. The recognition method cannot recognize these lines. (8 cases)
(c) Some relationship lines go across classes, which makes the class recognition or relationship recognition inaccurate. (4 cases)
(d) Some images' background colors gradually change, which causes problems on the grayscale step in pre-process that lead to recognition error. (4 cases)

Table 11
Recognized semantic elements of our approach.

	Recognized by our approach
Class	✓
Class's properties	✓
Generalization	✓
Realization	✓
Dependency	✓
Aggregation	✓

(e) Some relationship lines are too short to be recognized by our approach. Although this situation can be intuitively addressed by changing the threshold for recognizing the shortest lines in an image, a too-small threshold can cause more problems since there will be "too many short lines" with wrong results. (2 cases)
(f) Some classes are too close to each other, tending to wrongly recognize them as one class. (2 cases)
(g) Some texts are overlapped with relationship lines, which break one line into separated segments, and eventually affect the relationship recognition. (2 cases)

Some of the above situations can be dealt with by parameter adjustments. For example, we partially address the "short line" situation in (e) by adjusting the threshold for recognizing the shortest line in an image to a very small number, which in our case is 29 pixels. It is difficult to automatically determine the best threshold since the length of a line can be affected by the image's width or height, the line style, and the drawing style of the image's creator, which may be a research direction in future works. We also argue that the situations which are mainly image storage problems are not considered optimization in this work, such as the "blur line" situation in (a) and the "too light color" situation in (b).

Some situations are hard to be dealt with automated techniques, such as the "line over class" situation in (c), the "gradual color" situation in (d), the "too close" situation in (f), and the "text in line" situation in (g). However, we do not expect to solve all of the problems. For these situations, we can give an alert on the possible identified errors to help engineers adjust the specific identified elements easily. These situations account for 20% of all the unexpected situations, while the overall results (both the value of precision and recall) on other cases are still around 90%. In this way, our approach is helpful and feasible.

5.4.2. Capability of our approach

The main goal of our approach is to help the existing consistency check approaches by recognizing the essential semantic elements of a class diagram image. The semantic elements used by current consistency check approaches are already listed in Table 1. In this section, we would like to evaluate the capability of our approach. Table 11 shows our approach's capability of recognizing semantic elements of UML class diagrams. We can recognize classes, class properties (including attributes and methods), and four types of relationships (generalization, realization, dependency, and aggregation).

Compared with Table 1, all existing consistency check approaches use the class and dependency relationships, while our approach can recognize them with a precision around 90%. Four approaches (that account for 80%) use the generalization and realization relationship, which our approach can recognize with a precision of over 85%. Only one approach uses the aggregation relationship, which is also supported by our approach. We conclude that our approach can help the existing consistency check approaches to get a semantic model from UML class diagrams in most cases.

5.5. Threats to validity

There are three kinds of potential threats to the validity in our work, based on the taxonomy proposed by Wohlin et al. (2012). They are threats to external validity, threats to construct and internal validities.

5.5.1. Threats to external validity

According to Wohlin et al. (2012), external validity concerns the applicability of the proposed approach in a more general context. We try our best to make the samples representative. We used the latest and biggest repository of UML diagrams (modelsdb) and crawled around 90% the class diagrams based on their URLs of GitHub projects. After filtering the invalid URLs, we get 18 054 projects with a total of 68 313 links to UML diagrams (including 37 872 UML class diagrams). We conducted stratified random sampling from these repositories to produce the samples used in our experiment. For those who want to check our data more deeply, a bundle of all artifacts we produced during this work was prepared. Since the collection is not tool-oriented, it can cover many styles of UML class diagrams drawn by different tools. We are confident that this can help make our results more applicable in practice.

5.5.2. Threats to construct and internal validities

According to Wohlin et al. (2012), construct validity concerns the generalizability of the constructs under study, while internal validity concerns the validity of the methods employed to study and analyze data.

The first threat to construct and internal validity is from the dataset construction. To alleviate subjectivity, for calculating the ratio of different diagram types in open-source projects, we attempt to build a comprehensive collection of different diagrams with the query combining the diagram name, abbreviations and synonyms. However, queries for component diagrams (e.g., "architect*" and "module*") may lead to wrong results, such as high-level class diagrams which represent the architecture or modules of a system. We manually check the query results to eliminate the bias. However, we admit that this threat to validity still remains as we cannot manually check all of the diagrams. For the manual popular tools identification in open-source diagrams, we randomly collect 200 samples through a stratified sampling (introduced in Section 3.1) according to the project LOC. In general, larger projects have more complex design models. However, there might be a selection bias. Small projects may use various types of tools, and some of these tools may be rarely seen in projects. We admit that our stratified sampling may miss such rarely used tools and lead to incompleteness of the editor distribution estimation.

Besides, we manually annotate the semantic elements from the selected images and evaluate the automated identification. The wrong annotation must impact the evaluation results. To mitigate this threat, we go over the UML 2.5 Specification (Selic et al., 2015) carefully before the annotation although we are familiar with them and use UML models during our routine work. During this process, we make sure to be crystal clear about every related element in this present study.

To alleviate the threats from the survey, we carefully design the questionnaire with clear target (complementary to the survey in GitHub) and focus on the elicitation of design model usage. The questions within the same category are arranged in logic order. Besides, we select diverse participants including academic researchers from different countries and IT engineers from different groups. During the answer collection, we only select complete answers and filter out the incomplete ones, in order to improve the validity of answers.

To reinforce the construct and internal validity of our observation, the experiments for evaluation were executed over two sessions. The first experiment focuses on the comparison between our method and Img2UML on the major UML tools, while the second experiment focuses on the performance evaluation on the examples from practice. We also discussed the major problems that may have influences on the recognition, with the purpose of helping future studies tackle these challenges.

To eliminate the bias in the case selection, we do the stratified sampling process and select 10 diagrams in each category of projects according to their LOC. In order to select diagrams which are not too simple for the evaluation, we manually check if each selected diagram has at least three classes. Finally, we get a total of 50 diagrams in all categories of projects.

6. Conclusion

Software design model diagrams (such as UML class diagrams, component diagrams, etc.) are of vital importance during software development and maintenance. The usage of design diagrams in the open-source communities and industry are different because of their own features, so in this work, we conduct a survey to grasp a picture of the current status of the design diagram usage in the open-source communities and industry. The results show that the UML class diagram is the most commonly used standard model diagram in both open-source communities and industry, while the form of model storage (mostly images) makes it difficult to use.

Since the daily usage and maintenance of UML class diagrams may be very difficult, we propose an approach for recognizing semantic elements in UML class diagrams based on Img2UML, in order to help practitioners analyze models with other tools. The main contribution of our method is recognizing more types of relationships with polygonal lines, and suiting more situations of UML class diagrams drawn by various tools with different resolutions. The results of the experiment on 30 diagrams show that our method performs well on three major UML tools, as well as on 50 diagrams collected from open-source communities. Our approach can also cover many types of semantic elements that are commonly used by consistency check approaches.

For future work, we will promote our approach to addressing more kinds of class diagrams drawn by different tools, and perform additional experiments on more diagrams from different sources in order to explore the robustness of our approach. We will also compare our method with more types of recognition methods, in order to inspire better UML class diagram recognition approaches.

CRediT authorship contribution statement

Fangwei Chen: Data curation, Formal analysis, Methodology, Software, Writing – original draft. **Li Zhang:** Conceptualization, Funding acquisition, Writing – review & editing. **Xiaoli Lian:** Funding acquisition, Formal analysis, Investigation, Methodology, Validation, Writing – review & editing. **Nan Niu:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the National Science Foundation of China Grant No. 61732019 and 62102014. It is also supported by Key Laboratory of Software Development Environment, China No. SKLSDE-2021ZX-10.

References

Adersberger, J., Philippsen, M., 2011. ReflexML: UML-based architecture-to-code traceability and consistency checking. In: European Conference on Software Architecture. Springer, pp. 344–359.

Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W., 2003. TIMES: a tool for schedulability analysis and code generation of real-time systems. In: International Conference on Formal Modeling and Analysis of Timed Systems. Springer, pp. 60–72.

Barinova, O., Lempitsky, V., Kholi, P., 2012. On detection of multiple object instances using hough transforms. IEEE Trans. Pattern Anal. Mach. Intell. 34 (9), 1773–1784.

Beltrametti, M.C., Massone, A.M., Piana, M., 2013. Hough transform of special classes of curves. SIAM J. Imaging Sci. 6 (1), 391–412.

Best, N., Ott, J., Linstead, E.J., 2020. Exploring the efficacy of transfer learning in mining image-based software artifacts. J. Big Data 7 (1), 1–10.

Biehl, M., Löwe, W., 2009. Automated architecture consistency checking for model driven software development. In: International Conference on the Quality of Software Architectures. Springer, pp. 36–51.

Buckley, J., Mooney, S., Rosik, J., Ali, N., 2013. JITTAC: a just-in-time tool for architectural consistency. In: 2013 35th International Conference on Software Engineering (ICSE). IEEE, pp. 1291–1294.

Caracciolo, A., Lungu, M.F., Nierstrasz, O., 2015. A unified approach to architecture conformance checking. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture. IEEE, pp. 41–50.

Chandan, G., Jain, A., Jain, H., et al., 2018. Real time object detection and tracking using deep learning and opencv. In: 2018 International Conference on Inventive Research in Computing Applications (ICIRCA). IEEE, pp. 1305–1308.

Chavez, H.M., Shen, W., France, R.B., Mechling, B.A., Li, G., 2015. An approach to checking consistency between UML class model and its java implementation. IEEE Trans. Softw. Eng. 42 (4), 322–344.

Chen, F., 2022. Open science materials of the paper "Automatically recognizing the semantic elements from UML class diagram images". http://dx.doi.org/10.5281/zenodo.6717688.

Chen, F., Zhang, L., Lian, X., 2020. An improved mapping method for automated consistency check between software architecture and source code. In: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS). IEEE, pp. 60–71.

Chudasama, D., Patel, T., Joshi, S., Prajapati, G.I., 2015. Image segmentation using morphological operations. Int. J. Comput. Appl. 117 (18).

Deng, S., Huang, Y., 2017. Fast algorithm of dilation and erosion for binary image. Comput. Eng. Appl. 53 (5), 207–211.

Dogra, A., Bhalla, P., 2014. Image sharpening by gaussian and butterworth high pass filter. Biomed. Pharmacol. J. 7 (2), 707–713.

Duan, D., Xie, M., Mo, Q., Han, Z., Wan, Y., 2010. An improved hough transform for line detection. In: 2010 International Conference on Computer Application and System Modeling (ICCASM 2010), Vol. 2. IEEE, pp. V2–354.

El-Salamony, M., Guaily, A., 2020. Enhanced modified-polygon method for point-in-polygon problem. In: Recent Advances in Engineering Mathematics and Physics. Springer, pp. 47–61.

Girshick, R., 2015. Fast r-cnn. In: Proeedings of the IEEE International Conference on Computer Vision. pp. 1440–1448.

Gosala, B., Chowdhuri, S.R., Singh, J., Gupta, M., Mishra, A., 2021. Automatic classification of UML class diagrams using deep learning technique: Convolutional neural network. Appl. Sci. 11 (9), 4267.

Gravino, C., Tortora, G., Scanniello, G., An empirical investigation on the relation between analysis models and source code comprehension. In: Proceedings of the 2010 ACM Symposium on Applied Computing. pp. 2365–2366.

Hains, G., Li, C., Atkinson, D., Redly, J., Wilkinson, N., Khmelevsky, Y., 2015. Code generation and parallel code execution from business uml models: A case study for an algorithmic trading system. In: 2015 Science and Information Conference (SAI). IEEE, pp. 84–93.

Haitzer, T., Zdun, U., 2012. DSL-based support for semi-automated architectural component model abstraction throughout the software lifecycle. In: Proeedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures. pp. 61–70.

Hammond, T., Davis, R., 2006. Tahuti: A geometrical sketch recognition system for uml class diagrams. In: ACM SIGGRAPH 2006 Courses. pp. 25–es.

Hassanein, A.S., Mohammad, S., Sameer, M., Ragab, M.E., 2015. A survey on hough transform, theory, techniques and applications. arXiv preprint arXiv:1502.02160.

Hebig, R., Quang, T.H., Chaudron, M.R., Robles, G., Fernandez, M.A., 2016. The quest for open source projects that use UML: mining GitHub. In: Proeedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. pp. 173–183.

Ho-Quang, T., Chaudron, M.R., Samúelsson, I., Hjaltason, J., Karasneh, B., Osman, H., 2014. Automatic classification of UML class diagrams from images. In: 2014 21st Asia-Pacific Software Engineering Conference, Vol. 1. IEEE, pp. 399–406.

Hoshen, Y., Peleg, S., 2016. Visual learning of arithmetic operation. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 30.

Illingworth, J., Kittler, J., 1988. A survey of the hough transform. Comput. Vis. Graph. Image Process. 44 (1), 87–116.

Javed, M.A., Zdun, U., 2014. A systematic literature review of traceability approaches between software architecture and source code. In: Proeedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. pp. 1–10.

Karasneh, B., Chaudron, M.R., 2013a. Extracting UML models from images. In: 2013 5th International Conference on Computer Science and Information Technology. IEEE, pp. 169–178.

Karasneh, B., Chaudron, M.R., 2013b. Img2uml: A system for extracting uml models from images. In: 2013 39th Euromicro Conference on Software Engineering and Advanced Applications. IEEE, pp. 134–137.

Karasneh, B., Chaudron, M.R., 2013c. Online Img2UML repository: An online repository for UML models. In: EESSMOD@ MoDELS. Citeseer, pp. 61–66.

Kirillov, A., Aforge. net framework. Retrieved September 25th from http://Www.Aforgenet.Com 68, 47–52.

Koschke, R., Simon, D., 2003. Hierarchical reflexion models. In: WCRE. Vol. 3. Citeseer, pp. 186–208.

Laroca, R., Severo, E., Zanlorensi, L.A., Oliveira, L.S., Gonçalves, G.R., Schwartz, W.R., Menotti, D., 2018. A robust real-time automatic license plate recognition based on the YOLO detector. In: 2018 International Joint Conference on Neural Networks (IJCNN). IEEE, pp. 1–10.

Machado, G., Alali, A., Hutchinson, B., Oluwatobi, O., Marfurt, K.J., 2015. Improving fault images using a directional Laplacian of a Gaussian operator. In: SEG Technical Program Expanded Abstracts 2015. Society of Exploration Geophysicists, pp. 1851–1855.

Mithe, R., Indalkar, S., Divekar, N., 2013. Optical character recognition. Int. J. Recent Technol. Eng. (IJRTE) 2 (1), 72–75.

Moreno, V., Génova, G., Alejandres, M., Fraga, A., 2020. Automatic classification of web images as UML static diagrams using machine learning techniques. Appl. Sci. 10 (7), 2406.

Mukhopadhyay, P., Chaudhuri, B.B., 2015. A survey of hough transform. Pattern Recognit. 48 (3), 993–1010.

Murta, L.G., Van Der Hoek, A., Werner, C.M., 2006. ArchTrace: A tool for keeping in sync architecture and its implementation. In: Brazilian Symposium on Software Engineering (SBES), Tools Session, Florianópolis, Brazil. Citeseer.

Nam, M., Ahuja, N., 2012. Learning human preferences to sharpen images. In: Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012). IEEE, pp. 2173–2176.

Osman, M.H., Ho-Quang, T., Chaudron, M., 2018. An automated approach for classifying reverse-engineered and forward-engineered UML class diagrams. In: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, pp. 396–399.

Ott, J., Atchison, A., Linstead, E.J., 2019. Exploring the applicability of low-shot learning in mining software repositories. J. Big Data 6 (1), 1–10.

Ozkaya, M., Erata, F., 2020. A survey on the practical use of UML for different software architecture viewpoints. Inf. Softw. Technol. 121, 106275.

Passos, L., Terra, R., Valente, M.T., Diniz, R., Mendonça, N., 2009. Static architecture-conformance checking: An illustrative overview. IEEE Softw. 27 (5), 82–89.

Purkait, P., Zhao, C., Zach, C., 2017. SPP-Net: Deep absolute pose regression with synthetic views. arXiv preprint arXiv:1712.03452.

QIN, K.-h., WANG, H.-y., ZHENG, J.-t., 2010. A unified approach based on hough transform for quick detection of circles and rectangles. J. Image Graph. 1.

Reggio, G., Leotta, M., Ricca, F., 2014. Who knows/uses what of the UML: A personal opinion survey. In: International Conference on Model Driven Engineering Languages and Systems. Springer, pp. 149–165.

Ren, S., He, K., Girshick, R., Sun, J., 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. Adv. Neural Inf. Process. Syst. 28, 91–99.

Robles, G., Ho-Quang, T., Hebig, R., Chaudron, M.R., Fernandez, M.A., 2017. An extensive dataset of UML models in GitHub. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). IEEE, pp. 519–522.

Selic, B., Bock, C., Cook, S., Rivett, P., Rutt, T., Seidewitz, E., Tolbert, D., 2015. Omg Unified Modeling Language (Version 2.5). Tech. Rep..

Sharaf, M., Abusair, M., Eleiwi, R., Shana'a, Y., Saleh, I., Muccini, H., 2019. Modeling and code generation framework for iot. In: International Conference on System Analysis and Modeling. Springer, pp. 99–115.

Shcherban, S., Liang, P., Li, Z., Yang, C., 2021. Multiclass Classification of Four Types of UML Diagrams from Images Using Deep Learning. In: Proc. of the 33rd International Conference on Software Engineering & Knowledge Engineering (SEKE). KSI. pp. 57–62.

Sinkala, Z.T., Herold, S., 2021. InMap: automated interactive code-to-architecture mapping. In: Proeedings of the 36th Annual ACM Symposium on Applied Computing. pp. 1439–1442.

Smith, R., 2007. An overview of the tesseract OCR engine. In: Ninth International Conference on Document Analysis and Recognition (ICDAR 2007). 2, IEEE, pp. 629–633.

Smith, R.W., 2013. History of the tesseract OCR engine: what worked and what didn't. In: Document Recognition and Retrieval XX, Vol. 8658. International Society for Optics and Photonics, 865802.

Szegedy, C., Toshev, A., Erhan, D., 2013. Deep neural networks for object detection.

Tavares, J.F., Costa, Y.M., Colanzi, T.E., 2021. Classification of UML Diagrams to Support Software Engineering Education. Tech. rep., EasyChair.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in software engineering. Springer Science & Business Media.

Xie, G., Lu, W., 2013. Image edge detection based on opencv. Int. J. Electron. Electr. Eng. 1 (2), 104–106.

Ya-jie, L., 2012. Edge detection based on mathematical morphology dilation and erosion of magnetic resonance image. Biomed. Eng. Clin. Med. 1.

Zarándy, A., Stoffels, A., Roska, T., Chua, L.O., 1998. Implementation of binary and gray-scale mathematical morphology on the CNN universal machine. IEEE Trans. Circuits Syst. I 45 (2), 163–168.

Zhao, Z.-Q., Zheng, P., Xu, S.-t., Wu, X., 2019. Object detection with deep learning: A review. IEEE Trans. Neural Netw. Learn. Syst. 30 (11), 3212–3232.

Fangwei Chen is a Ph.D. candidate in Beihang University in Beijing, China. His research interests are in software consistency checking. To be specific, he focuses on the heuristic IR-based approaches of consistency checking between software design and source code, including monolithic and microservice-based architecture, in order to address the inconsistency problem, such as recognizing design models from specific documents, establishing trace links between design and source code, and detect inconsistencies.

Li Zhang received her Bachelor, Master and Ph.D. degrees in School of Computer Science and Engineering, Beihang University, Beijing, China, in 1989, 1992 and 1996, respectively. She is now a professor in School of Computer Science and Engineering, at Beihang University, where she is leading the expertise area of System and Software Modeling. She has around 20 years of experience of conducting industry-oriented research in various application domains such as Avionics and Ships, and Communications in several countries including America, Norway, and France. Her main research area is software engineering, with specific interest in requirements engineering, software/system architecture, model-based engineering, model-based product line engineering and empirical software engineering and has more than 100 publications on the above fields. She is also a committee member of Software Engineering in CCF (China Computer Federation), vice chair of Education Committee in CCF and PC member for several international academic conference, such as ICSSP, IESA, APRES, and etc.

XiaoLi Lian is an assistant research fellow in Beihang University in Beijing, China. Her research interests are in software requirements and software architecture. She focuses on designing heuristic rule-based approaches and application of deep learning and natural language processing technologies to address the problems on requirements, design and source code, such as mining software requirements from multiple domain documents, mining design rationale from software repository, and identifying the conflicts between software requirement specification.

Nan Niu is an Associate Professor in the University of Cincinnati's Department of Electrical Engineering and Computer Science. His research interests include requirements engineering, scientific software development, and human-centric computing. He received the Ph.D. degree in computer science from the University of Toronto.