A Refined Model of Ill-definedness in Project-Based Learning

Arthur Rump
Computer Science
University of Twente
The Netherlands
a.h.j.rump@student.utwente.nl

Vadim Zaytsev
Formal Methods and Tools
University of Twente
The Netherlands
vadim@grammarware.net

ABSTRACT

Project-based courses are crucial to gain practically relevant knowledge in modelling and programming education. However, they fall into the "ill-defined" domain: there are many possible solutions; the quality of a deliverable is subjective and not formally assessable; reaching the goals means designing new artefacts and analysing new information; and the problem cannot always be divided into independent tasks. In this paper, we refine the existing two-dimensional (verifiability and solution space) classification of ill-defined classes of problems, contemplate methods and approaches for assessment of projects, and apply the model to analyse two study units of two different computer science programmes.

CCS CONCEPTS

• **Social and professional topics → Computing education**.

KEYWORDS

learning objectives

ACM Reference Format:
Arthur Rump and Vadim Zaytsev. 2022. A Refined Model of Ill-definedness in Project-Based Learning. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3550356.3556505

1 INTRODUCTION

Project-based learning is a student-centred form of learning based on the constructivist ideas that learning is context-specific, that students learn best when they are actively involved in the learning process and that learning happens through social interaction and the sharing of knowledge [9]. Although not every implementation of a project means that project-based learning is applied, it is also difficult to use projects purely for summative assessment because students will almost always learn new things when working on a complex project. This means that some of the benefits of project-based learning are likely to occur to some degree in any course that uses a project.

A project in project-based learning has two essential characteristics: there is a *driving question*, often in the form of a problem to be solved, and the learning activities result in a set of artefacts or products, which represent students' solutions and reflect their knowledge [4, 10]. The question and activities can be determined by students or teachers, but it is important that the question is not so constrained that the outcomes are predetermined. Blumenfeld et al [4, p.372] note that "students' freedom to generate artefacts is critical, because it is through this process of generation that students construct their knowledge."

This freedom in generating artefacts that become the solution, eventually delivered to the teachers for grading, leads to many issues in assessment of that solution. Such problems are called "ill-defined", and this ill-definedness of a project is crucial to let students construct their knowledge and thus learn. We will recall the definition of ill-defineness and link it to our situations in § 2. Then, in § 3, we will introduce two study units of two different programs at our university, each being the first opportunity for corresponding students to model an entire software system by themselves by applying principles of object-oriented design. In § 4 we crystallise lessons learnt from analysing these two into concrete refinements on the existing model of ill-definedness. In § 5, we apply the resulting framework to the two study units we just introduced. The paper ends with § 7 which draws some conclusions.

2 ILL-DEFINEDNESS

A word that is commonly used for problems that do not have a definite solution is **ill-defined**. Such problems have an *indefinite endpoint*, meaning that determining if the goal has been reached is complex and imprecise, and it is one of three criteria Simon [15] (as cited in [5], we lack access to the original) describes for calling a problem ill-defined. The other two features they find are an *indefinite starting point*, meaning that the problem description is vague or incomplete, and *unclear strategies* for finding a solution. Wherever a project description mentions "good" solutions or talks about a free selection of additional features, we see ill-definedness. Additionally, software engineering has, in general, no single strategy for finding a solution: it always involves some creativity that starts to manifest at use case diagrams and persists till the last line of code.

Lynch et al describe five features of an ill-defined domain [13]:

- *There are multiple solutions, and which one is better is partly subjective.* This is certainly the case for project-based learning: every group will likely have very different solutions, and which follows a better style is to some degree a matter of taste.
- *There is no formal theory for determining a problem's outcome and testing its validity.* There is no formal theory of modelling and programming that can derive a correct and valid program

MODELS '22 Companion, October 23–28, 2022, Montreal, QC, Canada

from a problem description. As noted before, programming involves some creativity.

- *The task-structure is ill-defined: it involves the design of new artefacts or analysis with incomplete information.* The goal of projects is the design of a new artefact, so the task structure is ill-defined.
- *The domain involves abstract concepts without absolute definitions.* The project descriptions are written in natural language, the meaning of which is not absolutely defined.
- *The problem can not be subdivided into smaller independent subproblems.* While engineering does involve subdividing tasks into smaller subproblems, these smaller parts are often not independent. Separation of features in classes, for example, can always be done in a number of different ways, and the resulting system will still function correctly, even though components of one system will be interdependent with other components of the same system is a different way than that would happen in a different implementation.

Note how Lynch et al look at a whole domain when considering ill-definedness. They do not see a distinction between domains and problems and use the word "domain" to emphasise that the goal of learning is general domain knowledge rather than an answer to a specific problem. Fournier-Viger et al [5] argue that "the domain dimension is debatable, especially from the point of view of ITS researchers, because the main goal of building an ITS is to provide support to the learner at the level of individual tasks. Since a domain can contain both ill-defined and well-defined tasks, it does not seem relevant to discuss the ill-definedness of a domain as a whole."

Similarly, we argue that classification of a project as a whole is not relevant since a project is assessed based on many criteria. Instead, we should consider the ill-definedness of each criterion separately. Teachers do not just look at a project and grade it on a scale from 1.0 to 10.0 because that is indeed very difficult with an ill-defined problem. Instead, they use a list of criteria which are (hopefully) less ill-defined, or at least smaller in scope, so that the assessment task becomes more manageable. In § 3, we will take a closer look at the criteria in our projects.

One might ask: if ill-defined problems, like projects, are so difficult to assess, then why do we use ill-defined problems in education? (And why do we assess them directly, since some authors claimed we should assess gained insights instead of deliverables [1]). While ill-definedness is not a great feature when it comes assessment, it can be very beneficial for the learning process to work on complex problems. The class of ill-defined problems is very similar to the class of wicked problems [12], which are commonly used in challenge-based learning to teach real-world problem solving skills [6]. Project-based learning can be seen as a well-controlled, focused and contained form of challenge-based learning.

3 CASE STUDIES

The following descriptions are based on course materials, like the course manual, project description, and assessment forms. The first course we consider is the programming part of the module **Software Systems** (PSS from now on) for the Bachelor of Technical Computer Science. It is the second module in the study programme, and students are introduced to software modelling (conceptually in general and concretely in UML) and programming (in Java) following objected-oriented principles, building on a short introduction to imperative programming with Python in the first module. The core of the module consists of a design part and a programming part, each weighing half of the grade, totalling 12 EC. Each part is assessed through a written exam and a project. The final project takes place in the final four weeks of the module and is executed in teams of two students. According to the intended learning outcomes (ILOs from now on), students should be able to implement, document and test software of average size (10–20 classes).

The goal of the project is to develop a client-server multiplayer game. The game varies from year to year, but it is usually a variation on a board game where players take turns: one example would be three-dimensional tic-tac-toe. Students get a description of the game rules, the protocol that their client and server should use to communicate and a reference server and client to test their implementations. The client should have a textual user interface (TUI) that shows the game state to the user and allows them to make moves. The client should also include a computer player that can make moves automatically or to give hints to the user.

Besides the correct implementation of this functionality, the project is also assessed on the design and quality of the code, (automated) tests, documentation, and the explanations and reflection in the report. Students are able to get bonus points by implementing extra features like a chatbox or authentication, by handing in a minimal sufficient version a week before the deadline and by scoring high in a tournament of computer players.

The second example is the 3 EC **Algorithms for Creative Technology** course (ACT) that is part of the fourth module of the Bachelor of Creative Technology. In previous modules, students have got some experience in programming with the Processing language (https://processing.org) and the Arduino environment. In this course, students continue with Processing and learn more about object-oriented structure in larger programs, as well as algorithms to represent physical phenomena in their animations, like flocking and mass-spring-damper systems. The students are assessed through a project and oral examination about their project. The project is an individual assignment, though students are allowed to collaborate as long as they can explain every part in an oral examination. The project is executed in week 8 and 9 of the course, followed by oral exams in the final week.

For this project, there are few functional requirements. Students can choose their own topic for the project, as long as they combine at least three topics covered in the course. The requirements instead focus on the complexity of the program: there should be enough interaction with the user, there should be at least two (non-trivial) classes, etc. The code is also assessed on programming style and code quality. Besides the code, the project is assessed on the appearance of the running program, documentation and the student's explanations during an oral exam.

While these projects are quite different in the freedom students get to set their own goals, both projects give students a large degree of freedom in the way they implement their solution. This is one of the most important properties of projects when it comes to automated assessments: even if the functional requirements are well-specified, like in the PSS project, no two students will create the same program, let alone if they are given the freedom to choose their own functional requirements. In a project, students have so many choices to make that it becomes infeasible to anticipate every choice in a tool.

3.1 Requirements and Rubrics

With both projects students get a description of what they are expected to create, what the expected functionality is and how the project will be assessed. Both descriptions contain a list of requirements that the project should meet. The PSS project description gives, for example, some detailed functional requirements [17, p.22]:

- When the server is started, it will ask the user to input a port number where it will listen to. If this number is already in use, the server will ask again.
- When the client is started, it should ask the user for the IP-address and port number of the server to connect to.
- When the client is controlled by a human player, the user can request a possible legal move as a hint via the TUI.

The ACT project description specifies, for example, how much original work should be in a student's project [7, p.23]:

- At least two (nontrivial) classes have to be written by yourself (not from a tutorial or elsewhere from the web.
- Flocking and particles should not both be from a tutorial or other sources on the web, at least one of them self written.
- Most code of significant complexity must be self-written.

Both projects also use a rubric for the grading process. Because the main goal for the projects is summative assessment, these grading rubrics also describe the primary assessment criteria. For the Software Systems project, this rubric is nine pages long, five of which list requirements for the code and the other four describe the requirements for the report. The rubric is used to grade different aspects of the project, like game logic, networking, structure and design patterns. For each aspect, there are five columns ranging from poor to excellent, each of which contains one or more requirements that should be fulfilled to receive the corresponding grade. For the game logic, the levels are described as follows [17, p.134]:

- 1 (poor/missing): Common execution flows in the game logic are incorrect: basic game rules are not implemented correctly, gameover conditions are not applied correctly, etc.
- 3 (subpar): One of the following is applicable:
  - Game logic is almost correct except for corner cases.
  - Implementation supports only a few games or only one game simultaneously.
- 6 (sufficient): All game rules are correctly implemented. Many games can be played independently.
- 8 (good): As *sufficient* and one of the following:
  - Game logic methods do not crash with unchecked exceptions from incorrect usage by other classes (invalid moves, negative indices, etc).
  - Game logic is thread-safe (no race conditions even when used from multiple threads)
- 10 (excellent): As *good* and both points apply.

From this line in the rubric, we can extract a few assessment criteria, like "game rules are implemented correctly" (with three levels: fully correct, corner cases permissing, incorrect).

The ACT project also uses a rubric, which fits on a single page and looks like an evaluation form: it contains a list of requirements which the assessor can mark to be met on an exceptional, acceptable, amateur or unsatisfactory level. Unlike a normal rubric, there are no more specific criteria for each of these levels. The form is split in two parts: the first part lists the features the project could incorporate (like randomness, mechanics, particles etc), while the second part describes quality and correctness criteria, like "pixels not hardcoded in classes" and "no unused variables in code". To summarise, the assessment criteria are formulated like requirements, but some of them have an associated minimal grade, while others influence the final grade depending on the level.

Because of these similarities, we will treat "criteria" and "requirements" as synonymous, even though conceptually they have a subtly different meaning: a requirement is given apriori to guide the implementation, while a criterion is used to assess a solution after it has been created. In practice these two often coincide. Both the PSS and ACT project descriptions urge students to read the assessment criteria before starting the project, turning them into de facto requirements.

3.2 Grade Forming

One can convert judgement to grades in multiple ways. Biggs and Tang [3, p.238] describe three options:

1. Quantitative marking: counting how many points have been made, or how many criteria have been met. Here it does not matter which items are correct, as long as enough points are accumulated (generally, half of them is sufficient).
2. Analytic assessment: the assignment is reduced to independent components or aspects, which are each assessed qualitatively according to a rubric. Each aspect receives a grade and the final grade is determined as a (weighted) average of all subgrades.
3. Holistic assessment: the task is viewed as a whole and graded directly, based on the criteria. This is more similar to the acceptance process for peer reviewed publications: an editor or programme chair makes a judgement based on the value of a paper and the advice given by reviewers. The result is not determined by averaging the decisions of each reviewers, but by taking into account their arguments and making a holistic decision. For example [3, p.258], such a scheme can give the highest grade to students who have clearly met all the ILOs and went beyond established practice; somewhat lower grade when ILOs have been met well, even lower if they are met only satisfactorily, and a failing grade only if some ILOs have not been met at all (or in case of plagiarism/non-submission). Notable is that, typically, in a holistic assessment all criteria have to be met sufficiently, whereas marking and analytic assessment allow students to compensate the weak spots of their performance by scoring higher on other aspects. An holistic grading scheme could also allow that, of course, if that were desirable according to the ILOs.

The PSS project uses analytic assessment: each aspect of the project is graded according to a row in the rubric and the final grade is determined by taking the average of all those subgrades. In ACT, a more holistic approach is used, though the criteria for awarding a certain grade are not clearly specified. Instead, the assessment team relies mainly on their own expertise and ability to judge quality, while ensuring consistency by carrying out assessment in pairs with changing formations. The criteria on the evaluation form are used to judge the student's work and give feedback, and then deciding on a grade by taking into account those judgements. It is also the case that students are not able to compensate for a clearly insufficient aspect by performing strongly on other aspects: a program with bad structure will be awarded with an insufficient grade, even if it is quite good in all other aspects.

These different approaches mean that criteria are structured in many different ways. A list of criteria is sufficient for marking, while analytic assessment requires criteria to be listed in a rubric where each row has an importance indicated by its weight in the average. In that case the criteria could be specific to a certain level (column) of the rubric, or a criterion can itself be judged with a certain level as "output". This also holds for holistic assessment, where rubrics are typically used to judge each aspect, but here the rubric does not contain any information about the grading process or importance of each aspect.

Marzano and Miedema [14] also recommend using rubrics, though they directly assess the learning outcomes rather than certain criteria. The goal of this rubric is not describe an ILO in more detail, but rather to describe the different levels of mastery of that ILO. In their scale from score 0 to 4, they recommend setting the ILO at level 3, then level 4 could be about exceeding expectations, level 2 about partial accomplishment, level 1 about questionable conformance and level 0 means the lack of understanding. Note the similarity of this structure with the *Structure of the Observed Learning Outcome* (SOLO) taxonomy [2, 3]. This structure could also be applied to a criterion: in that case the rubric describes multiple levels of mastery for a single criterion, rather than criteria for different grades of an aspect.

3.3 Properties and Categories of Criteria

We have found five general properties of assessment criteria. *First*, a criterion relates to a certain aspect of the artefact (e.g., for PSS these include networking functionality, clean code and structure). *Second*, a criterion is more or less ill-defined. We have already seen that projects as a whole are ill-defined, which is why we use more granular assessment criteria to assess a project. Each of these criteria can also be ill-defined to some degree, which we will consider in the next section. *Third*, the criteria are structured in some way (e.g., bundled in a group or a rubric). For instance, in PSS: some requirements are listed as the primary functional requirements, and other criteria in the rubric describe what it means to score grade 10 on the client functionality and grade 6 on the structure. *Fourth*, a criterion can have one of three types of "output values": binary (e.g., "the project contains 3–5 activity diagrams"), scaled or numeric ("the game rules are implemented correctly" from the example below with its levels of accomplishment), or counting violations (e.g., if the criterion is "do not hide user interaction in classes", the assessment can be based on how many times this does happen in a program). *Fifth*, a criterion can include some grading information. In the case of analytic assessment, each criterion or aspect has a weight to be used in the weighted average, while in a marking scheme each criterion is worth a certain number of points. In holistic assessment the grading policy is described separate from the criteria.

The criteria for our projects fall into three categories, namely the three aspects regarding which the code is typically assessed in a programming project:

- **Correctness**: is the specification implemented correctly? (Whatever that specification might be.)
- **Scope**: is the project sufficiently large in scope, i.e. are enough features implemented?
- **Quality**: do the design and code follow good practices w.r.t. style, structure and standards?

These are not necessarily the only aspects in which a project is assessed: the quality of a report, the ability to answer questions about the code in an oral examination, successful teamwork and planning, and creativity in the addressed problem could, among others, also be considered. These aspects, however, are not necessarily apparent in the models and the code.

The line between correctness and scope is not always clear-cut: if a feature that should be implemented according to the project description is not there, you could argue that the program does not follow the specification and is thus lacking in correctness, or that the program does not implement the baseline of required features and is therefore inadequate in scope. We will consider this case to be lacking in scope: checking correctness is limited to features that are implemented. Whether a specification is part of a feature or a feature on its own, depends on the project description.

Assessing each of these types of criteria requires a different way of looking at the program. To check correctness the program is often executed and tested, either manually or using automated tests. An experienced corrector could also scan the code or diagrams for common mistakes. The scope is assessed on a holistic level, checking for the presence of certain classes or functions, or certain elements in the program UI. Style is assessed by reading the code, but considering the way the code is written, rather than how the code works. This means that an automated assessment tool requires different views of the program to assess these different types of criteria.

This categorisation of criteria seems to align with assessments reported in the literature. Thompson [16] describes a set of assessment criteria based on the SOLO taxonomy, related to the amount of functionality delivered, to what degree the program operates without errors, and to the quality of the code regarding programming and UI design standards, code structure and documentation. The first two clearly map to scope and correctness, while the latter maps in part to quality (it also includes UI design standards, which are visible in the running program, but not clearly a property of the code itself).

Kao et al [8] give a rubric for scoring programming projects in middle school, in which they assess three dimensions: holistic review, programming fundamentals and programming style. The holistic review includes five categories: program correctness, usability, project scale, project complexity and creativity. In the programming fundamentals dimension the use and mastery of basic constructs like variables and loops are assessed. The programming style dimension considers the use of comments and readability of

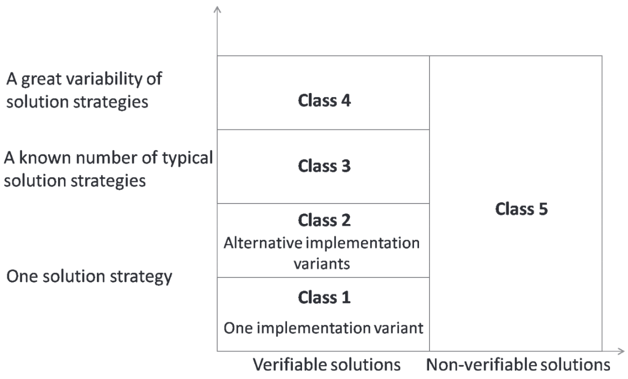
A great variability of solution strategies
A known number of typical solution strategies
One solution strategy
Class 4
Class 3
Class 2
Alternative implementation variants
Class 1
One implementation variant
Class 5
Verifiable solutions
Non-verifiable solutions

Figure 1: The 5 class scale of ill-definedness [11, p.262]

the code. The holistic review contains categories related to two of our aspects: the program correctness category clearly maps to correctness, while project scale and project complexity are part of the scope aspect. The programming fundamentals dimension also splits across correctness and scope: assessing mastery of constructs involves checking if they are used correctly, while mere usage falls in the scope bucket, because the project should be sufficiently large in scope that multiple constructs are used. The programming style dimension maps to the quality aspect, of course. This leaves two categories from the holistic review: usability and creativity. Usability is a property of the running program, not of the code, and thus goes beyond our categorization. Creativity can be viewed in many ways. As mentioned before, creativity in the problem that was chosen or creativity in properties of the running program are not part of the code assessment, even though they could well be valid assessment criteria for the project. In this rubric, creativity was restricted to the code, as can be seen from the guidance for scoring [8, p.1136], talking about projects being near-clones, extensions or non-duplicates of learnt components. This definition of creativity would place it under the scope aspect, as it is reflecting the scope of the work the student has done themselves.

With these three classes of criteria, we start to see some structure in the long list of criteria for our projects. The distinction is also relevant when considering automated assessment: assessing the correctness of a program is quite different from assessing quality. Criteria of the first type care about the behaviour of the program, while those of the latter care just about the models and code. When assessing behaviour, many stylistic differences in the models and code should be normalised away, because they do not influence behaviour, while those details can be very important in assessing the quality.

4 MODELS OF ILL-DEFINEDNESS

Now that we have seen how assessment criteria are defined for our projects, the next question to answer is how well- or ill-defined these criteria are. In this section we will introduce a classification to help us make more useful distinctions than just marking every criterion ill-defined or not. This classification will be applied to the criteria of PSS and ACT in the next section.

A great variability of solution strategies				
A known number of typical solution strategies				
One solution strategy, alternative implementation variants				
One solution strategy, one implementation				
	Specified requirement			Free choice of req
	Objectively assessable	Subjectively assessable with a rubric	Subjectively assessable, no rubric	

Figure 2: Our extended classification of ill-definedness, using seven levels of verifiability

Fournier-Viger et al [5] argue that "there are no clear boundaries between ill-defined and well-defined domains. Rather, there is a continuum ranging from well-defined to ill-defined". Le et al [11] take this notion and consider the solution space of a problem based on three characteristics: (1) the number of *alternative solution strategies*: the different approaches that could be used to solve a problem; in programming, these often correspond with different algorithms; (2) the *implementation variability*: the different ways in which a strategy could be implemented, for example, using more or less intermediate variables in a calculation or simply using different variable names; (3) the *solution verifiability*: whether a solution can objectively be verified as correct; this is not the case when, for example, aesthetics or usefulness are evaluated.

Based on these characteristics, they define five classes of problems (see Figure 1). In this classification, one axis is formed by the alternative solution strategies and implementation variability characteristics, which we will call the *variability* axis. The other axis is the solution verifiability, here shortened to the *verifiability* axis.

Class 1 problems have one correct solution which is also verifiable. Modelling exercises in this class could be small fill-in-the-blanks exercises. **Class 2** contains problems where there may be multiple ways to execute one solution strategy. Modelling problems in this category often are templated or templatable, with the exact number of entities in a model and exact nature or their associations being debatable. **Class 3** contains problems for which there are a known number of solution strategies. For a modelling exercise, this could mean choosing among Visitor, Observer and Interpreter design patterns, where each option leads to a different solution with pros and cons, but all are acceptable. If the number of different solution strategies is very low, it is even possible to write them all down as grading instructions or for auto-grading. **Class 4** problems have a great variability of solution strategies, where their number is so large that it becomes infeasible to enumerate the all. These problems still have an objectively verifiable solution. In programming, this can be done with test cases or with invariants, and it is much trickier to automate in modelling (unless some advanced tooling can, for instance, ensure that all use cases are covered by the interactions diagrams). **Class 5** contains problems for which the solutions are not objectively verifiable. It is the only class on the right side of the verifiability axis, covering the full range of the variability axis, though these problems are likely to have multiple solution strategies.

Le et al use the classification to group intelligent tutoring and automated assessment systems based on the class of problems these tools can be used with. They find that different approaches work well to deal with problems of different classes, which is why they recommend that tool creators use the classification to describe its capabilities more precisely.

We believe there is room for extension in this classification by differentiating more in the verifiability axis. Some criteria can be verified objectively, others have agreed upon rubrics that guide more subjective verification. There are also cases where students are allowed to choose their own verification criteria. From this observation, we define four levels of verifiability:

- The requirement is specified and objectively assessable.
- The requirement is specified and subjectively assessable following an agreed upon rubric.
- The requirement is specified, it is subjectively assessable and there is no agreed upon rubric.
- Students are free to choose their own requirements.

Combined with the four levels of variability defined by Le et al [11], this results in the classification shown in Figure 2.

In the process of defining this classification, we considered some alternatives. For example, we considered a middle ground between specified and free choice requirements: a set of given requirements from which the students can choose. This type of requirement is found in both of our example projects: in PSS students can choose to implement some additional features (like a leaderboard or encryption of the client-server communication), and in ACT students can choose (with some restrictions) which elements of the course content are included in their project. While this certainly makes a difference for the students, there is little difference in terms of assessment: the requirement still has to be verified to determine if the student implemented it, which is no different from requirements that are fully specified. The case could be made that there is a difference in choosing to implement a requirement and doing so correctly, but that is also the case for mandatory requirements: students may still decide not to implement them all, for example due to time constraints.

Another aspect we considered in extending the classification is explainability. Intuitively, there is a difference between decisions that have an answer and decisions for which this answer can be explained, in the same sense as "explainable artificial intelligence" is different from "normal" AI. Explainability, however, is a property of the tool (or process) that makes the decision, not of the decision itself and thus is not a factor of the verifiability of a solution.

Also note that a specified requirement does not necessarily have to be shared with the students. While it is generally considered good practice to be transparent about assessment requirements, not sharing certain requirements (or not sharing all details) does not change the verifiability of a solution regarding that requirement. Guidelines such as "if the class diagrams contains fewer than five classes, reject it" also usually are shared with teaching assistants, but not with students because such a numeric limit does not guide their learning into any useful direction.

5 APPLICATION TO CASE STUDIES

We have seen the three different types of criteria the deliverable in a software project is typically assessed on and a classification of how ill-defined a criterion is. We will put them to the test by applying them to the projects introduced in § 3. For this, we consider the project descriptions and rubrics or assessment forms used during assessment. We only considered the categories of criteria concerned with the assessment of the code and the models, ignoring the report and the oral exams. However, some criteria not directly related to models and code remained: for example, the "appearance: attention for details" in the ACT project or the "usability of the client" in the PSS project. For completeness, these have been included in the fourth *Non-code* category.

Our main motivation for introducing this classification is to investigate ways to automatically assess such projects. We already discussed that different types of criteria need different approaches, and a similar argument can be made on the degree of ill-definedness: objective criteria are assessed differently from subjective criteria and thus require a different approach. Therefore, we will be looking at the lowest, most specific level of criteria.

This poses a challenge for working with rubrics: one of the levels in the classification is *subjectively assessable with a rubric*, but it is quite common for the levels of a rubric to contain different criteria. In some cases, these are specifications about the quantity or quality with which a criterion is met, for example, from "no documentation comments are present at all" to "all classes and methods have documentation comments". This is a single criterion, which is subjectively assessable with a rubric, and while "none" and "all" are quite objective, the levels in between are less clear cut. In other cases, the levels contain very different criteria. For example, from "there are protocol violations" to "the networking code is thread-safe". These are clearly two different criteria, and we will treat them as such. This means we use a narrower definition of what constitutes a *rubric* than is common in practice at courses like PSS and ACT: we only consider it a rubric when judgement is required as to the level to which a criterion is met. The different levels in a line in the rubric describe a different level of meeting the criterion, based on quality or quantity (or both).

Our findings are shown in Figure 3 for the PSS project and in Figure 4 for the ACT project. Remember that the size of a bubble is not representative of the grade weight carried by criteria in that category, but just of the number of criteria in that category. Also, note that the bubble size between the images is not on the same scale, as the ACT project has shorter assessment guidelines and fewer criteria overall than PSS. These figures show that there is some hope for automated assessment of projects: there are many objectively assessable criteria, even in the lower ranges of variability (where it is feasible to describe all different approaches). For the PSS project, this includes some requirements on the lowest level of variability, like "all fields except constants are `private`". In § 4 we noted that these Class 1 problems often take the form of fill-in-the-blanks exercises, of which this is a variation: the student has to fill in the access modifier for a field, and if it is not a constant, there is only one correct answer.

On the other hand, the ACT project only has a single correctness criterion: the program should be functionally correct. This creates

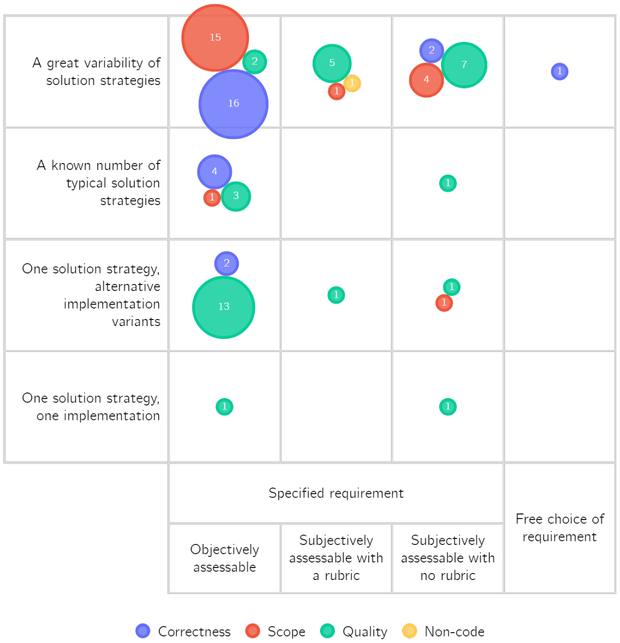

Figure 3: Classification of assessment criteria for PSS

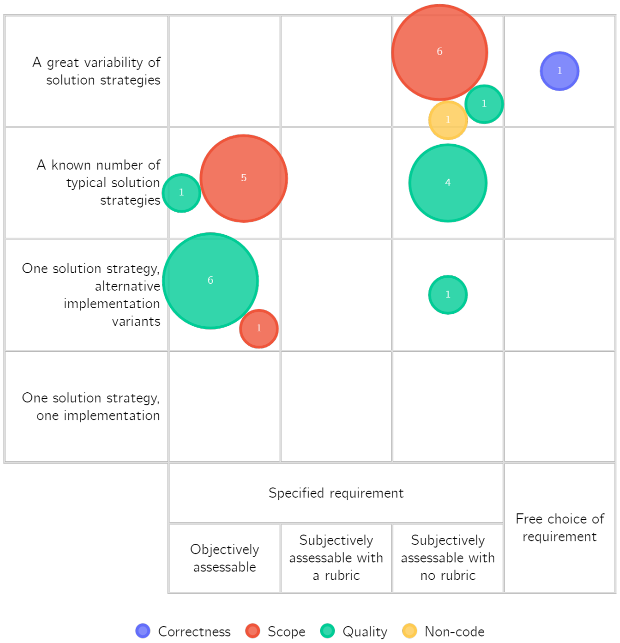

Figure 4: Classification of assessment criteria for ACT

a free choice of requirements, because students decide what the functionality of their program should be. A free choice requirement will generally be at the top of the classification, as there are a lot of strategies to implement whatever requirements the students come up with. Their specific requirements may have fewer solution strategies, but that is unknowable unless those requirements are specified.

The correctness criteria form the main difference between these projects: in the PSS project, most correctness criteria are well specified and objectively assessable, while in the Algorithms for Creative Technology project, students get a lot more freedom in this respect. In scope and quality, the projects have a lot more in common: these criteria spread around the specified area, with both objective and subjective assessments required, and a variation in the variability of solutions.

6 REFLECTION ON THE MODEL USEFULNESS

Besides getting an overview of what types of assessment a supporting tool would have to work with, the classification and visualisation applied here could also be used in course design. It gives an overview of how well-defined the assessment criteria for that course are and how much freedom students get in what the results of their projects would be. In this context, some modifications could be helpful.

We already noted that there is a middle ground between specified requirements and free choice: namely, a choice from a given set. We argued that for automated assessment the distinction between specified and choice from a set is not relevant, but looking through the lens of course design it would be: a project where students are allowed to choose which requirements to implement is very different from a project with strict requirements. For this context, we should extend the classification with three categories along the verifiability axis: between the specified requirements and free choice of requirements, we add a category *choice out of a given set of requirements* with the subcategories *objectively assessable*, *subjectively assessable with a rubric* and *subjectively assessable with no rubric*. This gives more insight into the freedom students have to decide the goal of their project.

We noted before that the bubbles in the visualisation represent the number of criteria rather than the importance of those criteria. For automated assessment we would care about the implementation of checks for these, so we care about their number and less about the influence on a grade of each criterion, which is simply a matter of configuration. However, for course design this is definitely relevant. If an analytic assessment approach is used, it would be more useful to scale bubble size with the total weight of the criterion in the final grade. This would give a better overview of the distribution across the classification from a grading perspective. With holistic assessment there is no predetermined weight, but an approximate relative importance could be assigned to each criterion or category of criteria for the purposes of this visualisation.

The amount of criteria in a circle of certain weight is still relevant, as it also tells us something about how strictly defined these aspects are. This could be included in the visualisation by changing the opacity of a bubble based on the criteria density or by showing each criterion as a point inside the bubble. We would expect this density to softly correlate with the verifiability axis, but including it in the visualisation could be useful to spot irregularities in this regard.

Finally, the categories *correctness, scope* and *quality* are chosen specifically for the assessment of models and code in a programming project and are interpreted in this strict sense. In a broader meaning of their terms, these categories may well apply to other types of assignments, but it seems likely that other categories could be added.

7 CONCLUSION

In this paper we have considered many issues around assessment of projects in introductory software engineering courses, where students already familiar with the basics of programming, are exposed to modelling, teamwork and agile processes. First, what are the assessment tasks used in such a software engineering course? In our examples, two types of tasks are used: PSS uses a written exam, and both courses use a project. Given the ILOs for both courses, the projects seem to be the most aligned task, so that is what we focus on in our research. The project is assessed according to certain assessment criteria.

Next, how are the assessment criteria defined? In general, the criteria are formulated as requirements that a good project should fulfil. Some of these requirements are simply listed out in a project description, for example if they are absolutely required. Other requirements can be written in a rubric: in this case a project should fulfil those requirements to receive a certain grade. In the detailed specification of the PSS project, most requirements are binary in nature: the project fulfils the requirement or it does not. Others can be fulfilled to a certain degree, in which case the assessor has to grade the level to which a requirement is fulfilled. Those levels may be clarified in the form of a rubric, but that is not necessarily the case in practice.

In terms of the structure for those criteria, we found three grading methods in the literature that influence the structure of the criteria: marking, analytic assessment and holistic assessment. When marking is applied, each criteria is worth some number of points, which are awarded as criteria are fulfilled. Analytic and holistic assessment are both qualitative methods, assessing aspects according to a rubric or list of criteria. In analytic assessment, these independent assessments are averaged to determine a final grade, whereas the holistic approach uses that data to argue for a grade based on the value of the whole artefact. In our courses the latter two methods are used. This means that our criteria are structured to correspond to certain aspects of the artefact.

Then, how should the assessment criteria be defined? Given the assessment method used, the criteria should be qualitatively assessable. In general, a rubric is the preferred way to describe the different levels of quality that may be encountered and also clarify what is considered insufficient, sufficient and good.

Finally, how well-defined are the assessment criteria? Even though students get a lot of freedom to make design decisions in their projects, we still find a large number of criteria that are more on the well-defined side of the spectrum for both projects. In the ACT project, about half of the criteria are classified as objectively assessable with up to a known number of solution strategies. In the PSS project, this is the case for about one third of the criteria, with another 40% objectively assessable but with a great variability of strategies. This means that an automated assessment tool should potentially be able to cover a large part of the assessment criteria quite well. Other criteria for both courses are more ill-defined, occupying the upper right corner of the diagram. Covering these criteria in an automated fashion could prove to be challenging to the point of impossibility.

In conclusion, how is the achievement of ILOs assessed in a software engineering course? Starting from the ILOs, teachers define assessment criteria and set tasks for the students, which generally results in a deliverable artefact. Given that the ILOs of a software engineering course often use verbs that indicate designing and writing code, these artefacts are mostly programs. Teachers then assess the artefacts according to the criteria.

We hope this investigation into ill-definedness is useful for other educators working in project-based learning, and that our descriptive model could serve as a convenient stepping stone to further research ways to assess students' contributions reliably, fairly and transparently.

REFERENCES

[1] Anya Helene Bagge, Ralf Lämmel, and Vadim Zaytsev. 2015. Reflections on Courses for Software Language Engineering. In *Proceedings of the MoDELS Educators Symposium (EduSymp'14)*, Birgit Demuth and Dave Stikkolorum (Eds.), Vol. 1346. CEUR-WS.org, 54–63.

[2] John Biggs and Kevin Collis. 1982. *Evaluating the Quality of Learning: The SOLO Taxonomy.* Academic Press, Inc.

[3] John Biggs and Catherine Tang. 2011. *Teaching for Quality Learning at University* (4th ed.). McGraw-Hill/SRHE/Open University Press.

[4] Phyllis C. Blumenfeld, Elliot Soloway, Ronald W. Marx, Joseph S. Krajcik, Mark Guzdial, and Annemarie Palincsar. 1991. Motivating Project-Based Learning: Sustaining the Doing, Supporting the Learning. *Educational Psychologist* 26, 3-4 (June 1991), 369–398. https://doi.org/10.1080/00461520.1991.9653139

[5] Philippe Fournier-Viger, Roger Nkambou, and Engelbert Mephu Nguifo. 2010. Building Intelligent Tutoring Systems for Ill-Defined Domains. In *Studies in Computational Intelligence.* Springer, 81–101. https://doi.org/10.1007/978-3-642-14363-2_5

[6] Silvia Elena Gallagher and Timothy Savage. 2020. Challenge-Based Learning in Higher Education: An Exploratory Literature Review. *Teaching in Higher Education* (Dec. 2020), 1–23. https://doi.org/10.1080/13562517.2020.1863354

[7] Marcus Gerhold et al. 2021. *Algorithms for Creative Technology manual.* Technical Report. Creative Technology, University of Twente.

[8] Yvonne Kao, Irene Nolan, and Andrew Rothman. 2020. Project Scoring for Program Evaluation and Teacher Professional Development. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education.* ACM. https://doi.org/10.1145/3328778.3366959

[9] Dimitra Kokotsaki, Victoria Menzies, and Andy Wiggins. 2016. Project-Based Learning: A Review of the Literature. *Improving Schools* 19, 3 (July 2016), 267–277. https://doi.org/10.1177/1365480216659733

[10] Joseph S. Krajcik and Namsoo Shin. 2014. Project-Based Learning. In *The Cambridge Handbook of the Learning Sciences*, R. Keith Sawyer (Ed.). Cambridge University Press, 275–297. https://doi.org/10.1017/cbo9781139519526.018

[11] Nguyen-Thinh Le, Frank Loll, and Niels Pinkwart. 2013. Operationalizing the Continuum between Well-Defined and Ill-Defined Problems for Educational Technology. *IEEE Transactions on Learning Technologies* 6, 3 (July 2013), 258–270. https://doi.org/10.1109/tlt.2013.16

[12] Johanna Lönngren. 2017. *Wicked Problems in Engineering Education: Preparing Future Engineers to Work for Sustainability.* Ph.D. Dissertation. Chalmers, Sweden. https://publications.lib.chalmers.se/records/fulltext/250857/250857.pdf

[13] Collin F. Lynch, Kevin D. Ashley, Vincent Aleven, and Niels Pinkwart. 2006. Defining Ill-defined Domains: A Literature Survey. In *Intelligent Tutoring Systems (ITS 2006): Workshop on Intelligent Tutoring Systems for Ill-Defined Domains.*

[14] Robert J. Marzano and Wietske G. Miedema. 2018. *Leren in 5 dimensies: Moderne didactiek voor het voortgezet onderwijs* (7th ed.). Koninklijke Van Gorcum.

[15] Herbert A. Simon. 1978. Information-Processing Theory of Human Problem Solving. In *Handbook of Learning and Cognitive Processes (Volume 5)*, William Estes (Ed.). Psychology Press. https://doi.org/10.4324/9781315770314-14

[16] Errol Thompson. 2007. Holistic Assessment Criteria: Applying SOLO to Programming Projects. In *Proceedings of the Ninth Australasian Conference on Computing Education - Volume 66 (ACE '07).* Australian Computer Society, 155–162. https://doi.org/10.5555/1273672.1273691

[17] Tom van Dijk et al. 2021. *Software Systems manual.* Technical Report. Technical Computer Science, University of Twente. Version of 18 January 2022.