Construction of a knowledge graph for exercise selection

Construction of a knowledge graph for exercise selection

Lex Bijlsma[1], Cornelis Huizing[2], Arjan Kok[1], Ruurd Kuiper[2], Harrie Passier[1], Erik Scheffers[2], Stefano Schivo[1], and Tanja Vos[1]

[1]Open Universiteit, Faculty of Science, Department of Computer Science, P.O. Box 2960, 6401 DL Heerlen, The Netherlands
[2]Eindhoven University of Technology, Faculty of Mathematics and Computer Science, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

October 14, 2021

Abstract

This paper describes the construction of an open-source repository to store and select exercises, whose architecture and design are usable for many knowledge areas. The repository has been built and is operational for a collection of Java programming exercises, independent of any specific teaching approach. Each exercise is equipped with a set of knowledge items to clearly show which programming knowledge is needed. Consequently, a tag-based search enables students to find those exercises that match their knowledge level. In order to make the process of tagging and searching efficient, the exercise tags have been structured into a foreknowledge graph. In this graph, each node is a tag while edges describe dependency relations between them. An interactive online tool is provided that supports the submission of and search for exercises.

1 Introduction

There is a long-standing, general demand for a repository that provides a substantial quantity of good quality programming exercises. Current large student numbers as well as an increase in on-line education, ranging from regular university courses to self-study oriented MOOCs, make this all the more topical.

This paper presents a theoretical foundation for, and an implementation of, an open source exercise repository that enables one to build, use and maintain a pool of exercises. Specifically, the repository enables teachers and students at different institutions to contribute and select exercises that provide training in programming. In this paper Java is used as the programming language, but the repository can equally well be used for other languages. The architecture and design of the searchable repository are, in fact, usable for all knowledge domains where prior knowledge plays an important role, yet different pathways through the subject matter are possible.

The main challenge is to match exercises to the knowledge level of a student in a manner that is independent of specific teaching approaches (for instance we accommodate 'objects first'

and 'objects late' courses equally well). The central idea is that an exercise may train several things, like *while*, *method* and *object*, and require foreknowledge like a thorough understanding of boolean conditions and a glancing acquaintance with *main* as occurring in template start-up code. The challenge is then, how to match, independently from teaching approaches, such an exercise to training desires. To achieve this aim, firstly each exercise is tagged with the knowledge it trains. Of course, the idea of adding metadata to learning materials to enhance findability is nothing new [10], but the level at which the tags apply, and what they exactly capture is quite specific for the repository purpose.

Secondly, a foreknowledge graph that captures precise dependencies between tags is introduced.

Thirdly, an interactive search function is developed that combines the information in the tags and the foreknowledge graph to identify exercises that match a training desire.

The exercises are stored in a database, together with information to search the database. An interactive search tool is provided that supports the submission of, and search for, exercises. (The system also provides automated testing of students' solutions, but in this paper we concentrate on the search tool.) The search tool is described in an abstract manner, implementation issues are deferred to a companion paper.

In Sections 2 to 8 the considerations that shaped the repository are presented in an incremental fashion, identifying subgoals and choices made. Throughout, this stepwise refinement is motivated by showing that the earlier, less complicated definitions would lead to a search tool that has undesirable properties for practical use. The search tool corresponding to the final definition has been implemented and is now in use at two universities with very different programming courses. As the difficulty with systems like this tends to lie in providing the tags, it is worth emphasizing that the approach explained in this paper has indeed led to a practically useful implementation. It can be viewed at `https://serf.win.tue.nl`.

After the exposition, in Section 9, Use cases are provided. In Section 10, Related work, connections are made to the study of object-oriented programming concepts and, more specifically, to work by Meyer and Petroni [8], and by Hubwieser [4].

The research reported on is performed in the context of SURF project "a Structured Exercise Repository with automated Feedback" (SERF). This project also concerns automated feedback on solutions, on-line availability of the repository, collaborative use and maintenance of the repository, and evaluating the impact on student performance and satisfaction.

2 Search by knowledge items – tags

Exercises are usually ordered from the perspective of a teaching approach, following the development of acquired knowledge in the approach. However, for an exercise set to be used by teachers and students at different institutions, using different teaching approaches, orderings may differ. Therefore it is proposed to accompany each exercise by information that determines at what point in different teaching approaches it will be useful.

An exercise trains the application of certain knowledge. Such knowledge consists of several knowledge items, each of which is trained by solving the exercise. As an example, an exercise

might be intended to train the knowledge items *while*, *method* and *object*. The set of all knowledge items in the system will be denoted by *Items*.

A submitter (usually a teacher) submits an exercise e to the exercise set *Exercises* of the repository. The submitter also provides a set of knowledge items that the exercise is intended to train. These will be called the *tags* of the exercise.

Definition 2.1 *The* tag set *of an exercise is the subset of Items consisting of the knowledge items the exercise is tagged within the repository. The tag set of exercise e is denoted by $Tags(e)$.*

We reserve the word tags for knowledge items that are attached to an exercise. For now, we assume that the submitters choose a tagging that for each exercise e gives a tag set, based on experience and professional judgement. For an example exercise E, let $Tags(E) = \{while, method, object\}$. (In what follows we will use the capital E when referring to this particular example.) In Sections 3 and 5 we present formal criteria for taggings.

A searcher for exercises (this can be a student as well as a teacher) wants to select exercises that train some desired knowledge item(s). So searchers indicate the knowledge items they want to train as a set of knowledge items, for example, $\{method, object\}$.

Definition 2.2 *A* training set *is a set of knowledge items a user submits to the system in order to be provided with exercises that train these knowledge items.*

A straightforward definition of a search function is to return the set of exercises for which the exercise tag set is the same as the training set.

The first approximation of the search function is therefore as follows: For any set of knowledge items *Train*, let $search_1$ be the set of exercises that have exactly *Train* as their tag set.

Definition 2.3 *Mapping $search_1$ from sets of knowledge items to sets of exercises is defined by*

$$search_1(Train) \stackrel{\text{def}}{=} \{e \in Exercises \mid Train = Tags(e)\}$$

Earlier we considered an example exercise, E, whose tag set was $(\{while, method, object\})$. This E is in the return set of $search_1(\{while, method, object\})$.

However, E is not in the return set of $search_1(\{method, object\})$, because E also trains *while*. The exercise tag set indicates the training an exercise provides, so if for an exercise the training set is a subset of the exercise tag set, that exercise provides *at least* the desired training, so it is useful to return that exercise as well. For instance, the example exercise that trains *while*, *method* and *object* can for some student at some stage in an approach be used to train *method* and *object* or in another case just *object*. However, it cannot be used for these purposes when the student does not yet have any knowledge of *while*. This point will be elaborated in the next section.

The second approximation of the search function is therefore as follows: for any set of knowledge items *Train*, the set of exercises that each have *Train* as subset of their tag set is selected by $search_2$.

Definition 2.4 *Mapping $search_2$ from sets of knowledge items to sets of exercises is defined by*

$$search_2(Train) \stackrel{\text{def}}{=} \{e \in Exercises \mid Train \subseteq Tags(e)\}$$

The example exercise, E, is still in the return set of $search_2(\{while, method, object\})$ and now also in the return set of $search_2(\{method, object\})$.

Remark 2.1 *Various choices have been made for already this first approximation. Because of the novelty of the approach, some motivation of choices made and mention of alternatives is provided.*

To keep the search function simple, and also to make it easy for students to estimate what has been trained when they complete an exercise, the choice has been made that each *exercise selected, taken on its own, provides the requested training rather than that the selected exercises together provide the requested training.*

To keep tagging simple, the choice has been made to use only a set of knowledge items for an exercise rather than allow combinations of tags with, e.g., boolean connectives. The exercise tag set induces an implicit 'and': an exercise trains all *knowledge items in its tag set.*

To keep search simple, the choice has been made to use only a set of training tags rather than to use a query language. The training tag set also induces an implicit 'and': an exercise gets selected if it trains (at least) all *knowledge items in the training tag set.*

Judicious use of $search_2$ still provides some, quite intuitive, flexibility in searching. Often training for a single knowledge item is desired: then the training set is a singleton. To search for all *exercises that train a knowledge item from a set of knowledge items, i.e., 'or' over the items in the set instead of the implicit 'and' over the items in a training set, separate searches can be performed for each item in the set. For example, $search_2(\{while, method\})$ selects the exercises that* each *train (at least)* both *while* and *method; $search_2(\{while\})$ and $search_2(\{method\})$* together *will also select the exercises that train both, but will* additionally *select the exercises that train while but* not *method, and method but* not *while, respectively.*

3 Limiting search by missing knowledge – negative tags

To include exercises that train more knowledge items than just the ones indicated, $search_2$ returns all exercises that train at least the desired knowledge items. However, exercise tags that were not in the search set may represent knowledge items that the searcher was just not interested to train but able to use, which is fine, but they may also represent knowledge items that the searcher has not mastered yet, which is problematic. For example, $search_2(\{method\})$ returns E, as $Tags(E) = \{while, method, object\}$. That E also trains *while* may for many students not be a problem, but that it also trains *object* may for quite some students mean that the exercise is beyond their level. Therefore search has to be adapted to enable to exclude exercises that require knowledge above a particular searcher's level.

The searcher indicates the above level knowledge items as a set of so-called negative tags.

Definition 3.1 *A* negative tag set *is a set of knowledge items that are not desired to be trained.*

For the example, $Negative = \{object\}$ is a possible negative tag set.

The third approximation of the *search* function will therefore introduce an extra argument, as follows.

Definition 3.2 *For training set Train and negative tag set Negative,*

$$search_3(Train, Negative) \stackrel{\text{def}}{=} \{e \mid Train \subseteq Tags(e) \ \wedge \ (Negative \cap Tags(e) = \emptyset)\}$$

For ease of notation, *Negative* is an optional argument. When it is not specified, that means $Negative = \emptyset$.

The example exercise, E, is not in the return set of $search_3(\{method\}, \{object\})$ – justifiedly so, as $Tags(E) = \{while, method, object\}$ and *object* was indicated as being above level. An exercise with tag set $\{while, method\}$ would be in the return set – justifiedly so, as *while* was not indicated as being above level.

Next it will be shown that $search_3$ is a generalization of $search_1$ and $search_2$ in the sense that it is at least as expressive as $search_1$ and $search_2$.

For any set of knowledge items T, the set of exercises that each have precisely T as their tag can be selected by $search_3$.

Proposition 3.1 *For each subset* $T \subseteq Items$,

$$search_3(T, Items \setminus T) = \{e \in Exercises \mid T = Tags(e)\}$$

In other words, $search_3(T, Items \setminus T) = search_1(T)$.

Proof:

$search_3(T, Items \setminus T) =$

$\{e \mid T \subseteq Tags(e) \wedge (Items \setminus T) \ \cap Tags(e) \ = \emptyset\} =$

$\{e \mid T \subseteq Tags(e) \wedge \ Tags(e) \subseteq T \ \} =$

$\{e \mid Tags(e) = T\}$.

□

For any set of knowledge items T, the set of exercises that each have T as subset of their tag set can also be selected by $search_3$.

Proposition 3.2 *For each subset* $T \subseteq Items$,

$$search_3(T) = \{e \in Exercises \mid T \subseteq Tags(e)\}$$

In other words, $search_3(T) = search_2(T)$.

Proof:

$search_3(T) =$

$search_3(T, \emptyset) =$

$\{e \in Exercises \mid T \subseteq Tags(e) \wedge (\emptyset \cap Tags(e) = \emptyset)\} =$

$\{e \in Exercises \mid T \subseteq Tags(e)\}$.

□

Apart from providing the functionality of $search_1$ and $search_2$, mapping $search_3$ also enables to select sets of exercises that neither $search_1$ nor $search_2$ can select, namely, by using *Negative* to deselect exercises that train knowledge items that are not required by $Train$. For example, if the exercise set contains exercises tagged by $\{while, method\}$, $\{do, method\}$ and $\{for, method\}$, the current definition enables to exclude just the exercises labelled with *for*, namely by $search_3(\{method\}, \{for\})$, which the naive versions cannot express with one search action.

An important issue to take into account is that *Train* is easy to describe for the searcher, but *Negative* less so. Searchers know which knowledge they want to train, but by the nature of 'above-level knowledge' are likely not explicitly aware of what that entails, making it difficult to provide a set *Negative* that captures their 'above-level knowledge'.

This issue is tackled the following way. When performing a $search_3$, to enable the seacher to define *Negative*, the tool displays for each exercise in the result its tags. This enables the searcher to assess whether he or she is familiar with that knowledge, and if this is not the case, to put the unfamiliar tag in *Negative*. Then when the searcher repeats $search_3$ with the updated *Negative*, exercises that required the above level knowledge are excluded from the result. For example, $search_3(method)$ has E in the return set and displays for E its tag set $\{while, method, object\}$. The searcher then, before attempting to solve this exercise, is alerted that knowledge about *while* and *object* is required for the solution, recognizes that *object* is above his level but *while* is not, and adds *object* to *Negative* for the next search – which will not return E anymore. This makes the search an interactive and iterative process.

Because just one negative search tag that is in the tag set of an exercise already means that the exercise is above level and hence should be excluded, it is efficient to search again after one or a limited number of exercises have provided (some) negative tags: in a new search with these negative tags all other exercises with those tags, also those having additional unfamiliar tags, are then excluded.

Of course, no tags should be in both the training set and the negative set: this would pose contradictory demands (having and not having a tag in the exercise tag set) on the exercises searched for and hence yield an empty search result.

4 Tagging with all knowledge items – $Needs(e)$ tagging

The intuitive aim for searching exercises is that with appropriate search terms, a search can return the set of all exercises that train desired knowledge items. Whether this is possible with $search_3$ depends on the tagging of the exercises.

The dependency on the choice of $Tags(e)$, which so far has been decided by the problem author without any constraints, means that it is not ensured that the set of exercises that trains *any* set of knowledge items can be returned. For example, say a searcher wants to further select, from the exercises that train {*while*, *method*, *object*}, the ones that also train console I/O. This can only be done if a tag relating to console I/O was considered relevant by the tagger and attached to the corresponding exercises. Therefore a specific tagging, by *all* knowledge items needed for solving an exercise, is defined.

Definition 4.1 *The* needs tag set *for an exercise e, $Needs(e) \subseteq Items$, is the set of all knowledge items needed to solve it.*

It is a reasonable assumprtion, one borne out in practice, that a consensus about what is needed among different taggers and searchers is feasible. A more formal definition in terms of, e.g., syntactic language elements or concepts like polymorphism used in a preferred solution is left for later research.

For any set of knowledge items T, the set of exercises that train T can be selected by $search_3$.

Proposition 4.1 *If each exercise e satisfies $Tags(e) = Needs(e)$, then for each subset $T \subseteq Items$,*

$$search_3(T, Items \setminus T) = \{e \in Exercises \mid T = Needs(e)\}$$

Proof:

Direct from Proposition 3.1 with $Tags(e)$ replaced by $Needs(e)$.

□

For any set of knowledge items T, the set of exercises that train T and possibly more can be selected by $search_3$.

Proposition 4.2 *If each exercise e satisfies $Tags(e) = Needs(e)$, then for each subset $T \subseteq Items$,*

$$search_3(T) = \{e \in Exercises \mid T \subseteq Needs(e)\}$$

Proof:

Direct from Proposition 3.2 with $Tags(e)$ replaced by $Needs(e)$.

□

Remark 4.1 *Note that the searcher can choose the Negative argument to be a smaller set than in Proposition 4.1 to not exclude some exercises that do not only train the desired knowledge of Train but also some already mastered items.*

5 Tagging with relevant knowledge items – the prior-knowledge graph

The role of tagging has now changed with respect to the initial version. Originally, the submitter provided tags to indicate what competencies an exercise was intended to train; however, the introduction of *Needs* added knowledge items that are necessary for the solution, without the exercise being considered good training for these. As Proposition 4.1 states, tagging with $Needs(e)$ in principle enables for any set of knowledge items to select the set of exercises that provides the training for these. However, it turns out that to be usable in practice, deeper insight in the role of tags and the relations between them is necessary.

The most important practical problem is over-tagging. For example, when tagging with $Needs(e)$, a very basic knowledge item like *assignment* will be a tag for almost all exercises. Because of the, well-motivated, inclusive use of tag sets in a search, $Train \subseteq Tags(e)$, a novice searching with *assignment* will get almost all exercises returned, most of them way beyond his knowledge level, and has to remove these beyond level exercises in further searches by selecting many negative tags from the displayed very large set of knowledge items. This is impractical and off-putting. Similarly, tagging with all $Needs(e)$ tags is laborious for the tagger.

Therefore, we need to prune the $Needs(e)$ tagging.

The idea is that an exercise should not be tagged with a knowledge item that is just prior knowledge for another knowledge item that the exercise needs. This suggests limiting the needs tags for an exercise to those knowledge items that are highest, 'top', in some hierarchy of prior knowledge. For the example, an exercise that has needs tag *while* should not have tag *boolean-expression* that is prior knowledge for the *while*.

Therefore, the needs relation on knowledge items is defined. (This relation should not be confused with *Needs*, which is a function on exercises.)

Definition 5.1 *The* needs *relation between knowledge items expresses that ItemA* needs *ItemB if mastery of ItemA must be preceded by mastery of ItemB.*

From the definition it follows that needs is transitive. The name needs is chosen for this relation to indicate that all knowledge items that have a needs relation from a knowledge item indicated by a tag of an exercise, are also needed knowledge items for that exercise.

The needs relation is an irreflexive partial ordering on knowledge items; it is not necessarily linear. Exercises will generally have several tags in their needs tag set that are not in a needs relation. For example, an exercise may train *recursion* and *extends*. Both these tags can then be included in the needs tag set, neither is prior knowledge for the other.

Knowing which knowledge item is and is not needed by other knowledge items would pose unrealistic demands on the tagger, and even more so on a less experienced searcher. Furthermore, there would likely be inconsistencies between individuals about this relation. Therefore, and for use in the search function, a prior-knowledge graph is developed.

Definition 5.2 *The* prior-knowledge graph *is the smallest directed acyclic graph with as nodes the knowledge items that has the property that there is a path from ItemA to itemB if and only if ItemA* needs *ItemB. If in this graph there is an arrow (directed edge) from ItemA to itemB, we shall denote this by ItemA* $\rightarrow$ *ItemB. The graph is visualized with the arrows pointing downward.*

A smallest directed acyclic graph exists and is unique because needs is an irreflexive partial ordering. An equivalent form of this definition is that for knowledge items *ItemA* and *ItemB* we have

$$ItemA \text{ needs } ItemB \;\Leftrightarrow\; ItemA \rightarrow^{+} ItemB$$

where $\rightarrow^{+}$ denotes the transitive closure of $\rightarrow$.

The minimality of the graph means that if *ItemA* $\rightarrow$ *ItemB* and *ItemB* $\rightarrow$ *ItemC* , hence also *ItemA* needs *ItemC*, the graph will not contain an arrow from *ItemA* to *ItemC*, because the corresponding needs relationship is already implied by *itemA* $\rightarrow$ *ItemB* and *itemB* $\rightarrow$ *ItemC*.

Rather than expecting the tagger to provide every exercise with its full needs tag set, we shall see that it is sufficient to provide a limited set of knowledge items from which the needs tag set can be generated by the paths in the graph.

The graph codifies what the knowledge items and their names are, i.e., the set *Items*, and what the prior-knowledge relation between them is. It is the crucial ingredient for practical tagging and searching, providing information to the tagger and to the searcher.

For an exercise e, the knowledge items in $Needs(e)$ that are relative sources, i.e., that cannot be reached by a path from other elements of $Needs(e)$, are the most difficult knowledge items needed for solving e. These we will call the *tops* of the needs tag set. In terms of the partial ordering these are minima; because we draw the graph with arrows pointing downward, they appear near the top of the page – hence the name. Using *only* tops as tags for the exercise remedies finding too many exercises as caused by over-tagging.

The second problem is under-tagging: not having enough exercise tags to find the exercise. This can be solved by tagging with *all* tags in the needs tag set that are tops.

The tops tagging is defined more formally as follows.

Definition 5.3 *The* tops tagging *for an exercise e,* $Tops(e)$*, is*

$$Tops(e) \stackrel{\text{def}}{=} \{t \in Needs(e) | \neg(\exists u \in Needs(e) : u \rightarrow t)\}$$

The original needs set of e may be recovered from $Tops(e)$ as follows.

Proposition 5.1

$$Needs(e) = \{x \in Items \mid (\exists t \in Tops(e) \;:\; t = x \;\vee\; t \text{ needs } x)\}$$

Proof:

To see this, take any element x of $Needs(e)$. If there is a $t \in Needs(e)$ with $t \to x$, walk up to t, and repeat the process until there is no such $\to$-predecessor available. This will eventually be the case because the graph is finite and acyclic. The invariant of this process is

$$t = x \ \vee \ t \ \mathsf{needs} \ x$$

Finally, after 0 or more steps, we have arrived at a $t \in Needs(e)$ without a suitable $\to$-predecessor, so

$$\neg(\exists u \in Needs(e) : u \to t)$$

By definition, this t is a member of $Tops(e)$. This proves left-to-right inclusion. The other direction is a consequence of the transitivity of needs.

□

This corresponds to the following tagging rule. The idea is that, according to Definition 4.1, for an exercise e the tagging with $Needs(e)$ provides *all* knowledge items needed to solve it. The graph then enables to prune this set down to tags that are tops in the graph.

For exercise e:

1. Determine $Needs(e)$.
2. Remove tags from $Needs(e)$ that are prior knowledge of another tag in $Needs(e)$ to obtain the *Tops* tagging.

The third problem is that tagging with only relative top tags has an undesirable consequence for negative tagging. For instance, if a student searches for recursion and gets an exercise labeled with, among others, the tag *extends*, this concept may be so far beyond his knowledge level that it is confusing rather than helpful. The student might know that *class* is beyond his level, and already have added that as a negative tag. However, although *class* is prior knowledge for *extends*, a negative tag *class* should but does not exclude exercises tagged with the *extends* tag but without the tag *class*. So for negative tags in a search *all* prior knowledge tags of the exercise tags must to be considered.

The search function now can be adapted so as to let negative tags exclude exercises e that have these tags in the $Needs(e)$ set of their prior knowledge: $search_4$ uses the information from the prior-knowledge graph to expand the tag sets of the exercises.

The fourth approximation of the *search* function is therefore as follows.

Definition 5.4 *For a train tag set Train and a negative tag set Negative,*

$$search_4(Train, Negative) =$$
$$\{e \in Exercises \mid Train \subseteq Tops(e) \wedge$$
$$Negative \cap Needs(e) = \emptyset\}$$

For any set of knowledge items T, the set of exercises that each have T as subset of their Tops-tag set can be selected by $search_4$.

Proposition 5.2 *If each exercise e satisfies $Tags(e) = Tops(e)$, then for each subset $T \subseteq Items$,*

$$search_4(T) = \{e \in Exercises \mid T \subseteq Tops(e)\}$$

Proof:

$search_4(T) =$

$search_4(T, \emptyset) =$

$\{e \in Exercises \mid T \subseteq Tops(e) \wedge \emptyset \cap Needs(e) = \emptyset\} =$

$\{e \in Exercises \mid T \subseteq Tops(e)\}.$

□

Note that this set is a defendable choice of tag set for an exercise, but that maybe some manual fine-tuning might be advisable. Tagging e with $Tops(e)$ gives a tag set that is reduced as far as possible while retaining access to all needed knowledge items. But for pragmatic reasons it may be preferable to retain some more tags. For example, an exercise may train a relative top knowledge item, but some knowledge that is one needs-arrow removed may be a more intuitive training aim for a student. An experienced tagger may also include this knowledge item as a tag.

'Pruning' is used to make the tagging easy to describe: in practice, rather than first establishing all needs-tags for an exercises, a tagger can with an eye on the graph avoid adding non-relative top tags from the start, and check at the end that all relative tops that are needed for solving the exercise are present.

This takes care of the practical tagging as far as tagging with needs tags is concerned.

These are the essential ingredients for a conceptually quite intuitive and simple, yet practicable, repository.

What has been achieved is that the tagging $Tags(e)$ as described in Section 2 as 'based on experience and professional judgement' has been replaced by a tagging $Tops(e)$. Here $Tops(e)$ aims to provide the same characterization of exercises by the knowledge items the exercises trains, but for $Tops(e)$ tagging guidelines based on a prior-knowledge graph are available. Furthermore, several strategies for searchers exist. A searcher may search by intuition only and add (negative) search tags when needed, or consult the knowledge graph first and then use more precise search requests to find the exercises of his liking.

In the next two sections we add two somewhat more advanced extensions to further refine the approach.

Remark 5.1 *Studying the graph enables the tagger to see whether all prior knowledge is covered beneath the given tags: tag as high as possible ('high' meaning minimal with respect to $\rightarrow$), without overshooting (overshooting means: the exercise does not train the tag-subject!). The graph also enables a searcher to know what low-level tags exclude exercises tagged with higher tags only. However, in practice this is mainly useful in case the searcher is a teacher selecting exercises for his course; for beginning students, a graph full of as yet unknown concepts will not be informative.*

Remark 5.2 *Studying the graph also helps the teacher to partially order exercises/lecturing: according to the needs relation.*

Remark 5.3 *In the use of search$_4$, negative tagging can* not *be used to remove exercises that (in addition to the search topics) train already mastered non-tops topics that the student is not interested in re-training: adding these to the negative set would also remove higher-level elements that the student does want to train.*

6 Useful prior knowledge – the uses relation

For the needs tags, given in Definition 4.1, needed knowledge was intended as *mastered* knowledge. For many exercises there also is useful knowledge of which *glancing acquaintance* rather than mastery is sufficient. For instance, anyone writing a Java program with a *main* function will encounter the concepts *public* and *static*, without necessarily understanding their full meaning. Such knowledge is indicated by a separate relation: *uses*.

Definition 6.1 *The* uses tag set *of an exercise e, $Uses(e)$, is the set of knowledge items of which glancing acquaintance is required to solve it.*

Note that being a uses tag for an exercise is approach independent.

For the **needs** relation, given in Definition 5.1, needed prior knowledge was described as *needing to be mastered*. For many knowledge items there also is useful knowledge of which *glancing acquaintance* rather than mastery to understand the knowledge item is sufficient. Such knowledge is indicated by a separate relation: uses.

Definition 6.2 *The* uses *relation between knowledge items*[1] *is that ItemA* uses *ItemB if mastery of ItemA needs glancing acquaintance of ItemB.*

Note that the uses relation between knowledge items is approach independent.

The characteristics of relation uses differ from that of needs in that it is not transitive; neither are we interested in its transitive closure. Indeed, using a knowledge item just means being aware that it should be mentioned in certain circumstances, without the necessity to know more about it. Printing information by calling on *System.out* is possible at a very early stage and does not require the realization that *System* is a public final class in the *java.lang* package, nor what any of these words mean.

Detecting the items directly used by an exercise can be automated easily: the tool merely needs to scan an example solution. For the tool to display all subjects for which a glancing acquaintance is required for an exercise e, it should compute

$$ShowUses(e) \stackrel{\text{def}}{=} \{u \in Items \mid u \in Uses(e) \vee (\exists t \in Needs(e) : t \text{ uses } u)\}$$

[1] Observe the difference with Definition 6.1: the earlier definition is concerned with properties of an exercise, the present one with relations between knowledge items.

for which the graph is employed.

7 Abstracting topics – the isa relation

The graph is not only used by the search function, but also to help taggers and searchers in their understanding. It is therefore important that the graph is conceptually manageable.

One issue is that there may be a proliferation of needs arrows. In Figure 1 tags A, B and C all need the same prior knowledge X, Y and Z. This makes for a spaghetti-like structure. (In fact, it is a well-known example from graph theory that this structure cannot be drawn without arrows crossing.) Abstractions and the isa relation will now be introduced in order

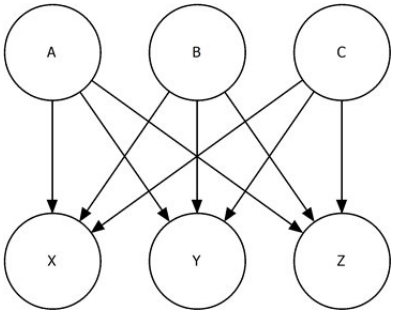

Figure 1: A cluttered graph

to simplify the graph.

Definition 7.1 *An* abstraction *is a set Q of knowledge items with the property that all elements of Q have some prior knowledge needs in common. Formally: there is a set $R_Q \subseteq Items$ such that*

$$\forall A \in Q \; \forall X \in R_Q \; : \; A \text{ needs } X$$

For an abstraction Q we write A isa Q instead of $A \in Q$, and Q needs X instead of $X \in R_Q$. If A isa Q we shall call A an instance *of Q.*

Next we modify the graph by adding these abstractions as extra nodes and extra arrows representing the isa relation. The isa relation is drawn as an arrow with an open head, while the needs relationships have a closed head. Definition 7.1 shows that there will be isa arrows going into an abstraction node and needs arrows going out from it. The direct arrows from the abstraction's instances to its needs are removed from the graph, as these are implied by the isa and needs arrows passing through the abstraction.

For example, *public* and *private* are instances of the abstraction *access*, which needs *class*. Hence *public* and *private* also need *class* without this being explicitly drawn in the graph. This yields the much less complicated graph of Figure 2.

This change in the graph necessitates a corresponding change in the computation of $Needs(e)$. Where formerly we started at $Tops(e)$ and added all items reachable via paths in the graph

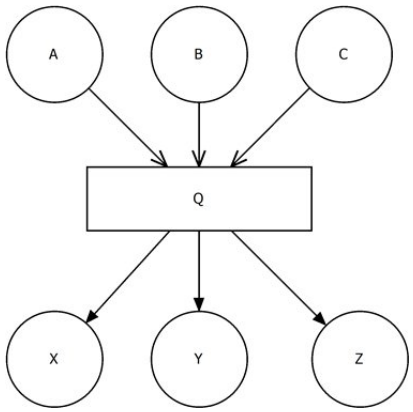

Figure 2: A less cluttered graph

composed of needs arrows, now we need to also take into consideration paths of needs and isa arrows that do not end with an isa arrow. Formally, Definition 5.1 must now be replaced by

$$\begin{aligned} &\mathit{Needs}(e) = \\ &\{x \mid (\exists t \in \mathit{Tops}(e) : t = x \vee \\ &\qquad\qquad (\exists u \ : \ t(\mathsf{needs} \cup \mathsf{isa})^* u \wedge u \rightarrow x))\} \end{aligned}$$

The bound variables x and u range over the graph's nodes, i.e., the set *Items*, which now includes abstractions. Relation $(\mathsf{needs} \cup \mathsf{isa})^*$ denotes the reflexive-transitive closure of the relation between parentheses. It is necessary to take this closure, because otherwise we would get only paths that consist exclusively of isa or needs arrows, not, as required, paths of mixed composition. Note that $search_4$ is still defined as in Definition 5.4, but with the new formula for *Needs*.

Observe that the introduction of abstractions does not change the information contained in the graph: it merely makes it easier for the human reader to visualise and understand.

8 Abstractions as first-class knowledge items

Observe that, when used purely in the fashion of the previous section, the incoming arrows of an abstraction are all of type isa. However, once we have decided to introduce such abstractions as a summary of more concrete knowledge items, it turns out that these can also be useful to indicate that for some exercise or knowledge item the prior knowledge consists of an understanding of concepts rather than any specific implementation of these. For instance, an exercise involving algorithm design will need the concept of repetition, but it is generally immaterial whether this should be expressed by *while* or *for* or another form. One should not tag such an exercise with *while* because there is no reason to prefer this above, say, *for*. One should certainly not tag it with both *while* and *for* because it does not need both. In such a case it is preferable that the exercise should be tagged with *repetition* rather than with an overspecific syntactic construct.

Definition 8.1 *An exercise is* tagged with an abstraction *when it requires understanding of that concept, but does not mandate the use of any particular syntactic construct to express the concept.*

This does *not* mean that an exercise tagged with *repetition* should require knowledge of *all* forms of repetition, but merely of the concept of repetition and of some way to implement this. Do not confuse this with the fact that the needs of an abstraction are shared by all its instances.

This tagging of exercises with an abstraction is intended for a very common class of exercises, in particular the ones that occur in the part of a course that deals with algorithm design. In this course part we get exercises such as, for example: given an array a, determine indices r and s with $r < s$ such that $a[r] * a[s]$ is as small as possible. Such questions are intended for students who have passed through the initial introduction of specific grammatical constructs for repetition and are now involved in the design of nontrivial programs that involve some sort of repetition. As to which grammatical construct they use their algorithm, we do not care at all.

In a similar spirit one may introduce needs arrows in the graph pointing to abstractions. (As we observed above, these do not occur with the use of abstractions for pure decluttering purposes.) For instance, reference types and primitive types are instances of the abstraction *type*, but it is less demanding to state that *assignment* needs prior knowledge of the abstraction *type* than prior knowledge of all available types.

In this way promoting abstractions to first-class citizens in the graph enhances expressivity. This expressivity is already useful in case of programming languages, but all the more so for domains with inherently abstract concepts, such as software engineering. For instance, the concept *subclassing* is an abstraction because *extends* and *implements* are instances of subclassing. Exercises may be tagged with *subclassing* when one does not care whether the students use an interface or an abstract class to achieve some purpose. However, *subclassing* is directly needed by *protected-access* and *dynamic-binding*: we simply cannot explain what dynamic binding is without discussing subclasses first.

Definitions 2.2 and 5.1 remain valid if we now interpret the term 'knowledge items' to include abstractions.

Note that in this case it should also be allowed to use abstractions in the negative tags of Definition 3.1. However, this removes only the exercises where the abstractions appear in the extended closure, hence are really required to understand the solution, not the exercises that merely use an instance of the abstraction. So putting *repetition* in a negative tag does not remove exercises that are explicitly intended to practice the *for* statement. Abstractions will appear in a negative tag set when a student encounters them as a consequence of teacher labeling; if we want the student to exclude them proactively, the graph should be supplied to the students as well as the teachers. Putting abstractions into a negative tag set allows the student to filter out the more advanced, high-level exercises; putting all the instances into the negative tag set but not the abstraction will produce just the advanced ones.

In the final version of the knowledge graph, three different types of arrows occur, corresponding to the needs, isa and uses relations. Figure 3 shows an example of their use.

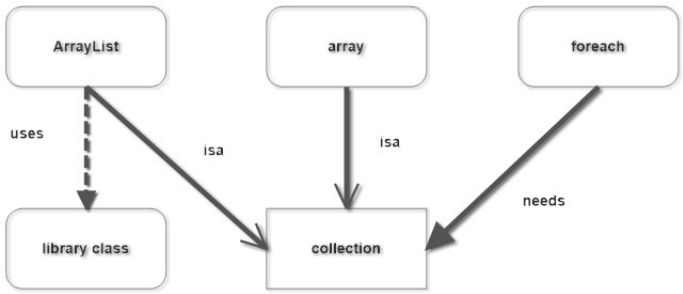

Figure 3: The different types of arrows

Remark 8.1 *One might ask whether all this could be simplified by ignoring the difference between the* isa *and* needs *relations. Is not* isa *merely a special kind of need? The answer is no. Consider the example of the primitive type* int*. Now clearly* int *is a type, but requiring a student to write some simple programs containing integer expressions is very different from expecting the student to have a deep understanding of the principles involved in the Java type concept.*

9 Use cases

In this section we describe the use of the tool in submitting exercises or solutions, and in retrieving suitable exercises for a given training purpose. The tool enables submitting solutions and receiving automated feedback.

9.1 Submitting an exercise

Primary actor: teacher.

Step 1 Upload the exercise text that will ultimately be presented to students attempting the exercise.

Step 2 Tag the exercise with knowledge it is intended to train.

Step 3 Upload an envisaged standard solution.

Step 4 The system automatically generates a set of syntax tags collected from the standard solution.

Step 5 Decide which of the generated syntax tags are needed and which are merely used in the sense of Definition 6.2. Store the latter.

Step 6 Decide whether any of the syntax tags must be combined with alternatives, or replaced by an abstraction connected to it by an isa relation in the knowledge graph. For instance, the system may have labeled an exercise with *for*, because that occurs in the standard solution, but the teacher may decide that a more proper tag is *repetition*.

Step 7 Add tags to indicate concepts that must be understood for semantic reasons in order to solve the exercise.

Step 8 Remove tags that are prior knowledge for other tags already present. This produces the tops tag set in the sense of Definition 5.3.

Step 9 Calculate and store the needs tag set. This step uses the graph and is performed automatically by the system.

Step 10 Provide a unit test to be used in evaluating student solutions.

9.2 Finding an exercise

Primary actor: either teacher or student.

Step 1 Choose one or more tags that indicate the subject for which the exercise is to provide a practice opportunity.

Step 2 Indicate tags for related subjects that have not been mastered and hence should not be required for the exercise. For instance, a query might have search tag *repetition* and negative tag *array*. This step is optional, but it will reduce the work in the next steps.

Step 3 The system retrieves a set of exercise titles whose tag set contains at least one of the search tags of Step 1.

Step 4 The system removes exercises from the set where needs tag set contains any negative tags listed in Step 2.

Step 5 The system displays a list of exercises together with their prior knowledge needs and uses tags.

Step 6 Repeat from Step 2, using a larger set of blocked knowledge items, until an exercise is found that has no unwanted prior knowledge.

9.3 Submitting a solution

Primary actor: student.[2]

Step 1 The student uploads a program text that is intended to solve the exercise.

Step 2 The system provides feedback. This is composed of the result of the unit test provided by the teacher at the time the exercise was added to the system, as well as comments from style checking software.

10 Related work

The IEEE LOM standard for learning object metadata enables one to find and reuse learning materials, learning objects, by tagging these [10]. The tagging presented in the current paper applies at a different level: inside one learning object (an exercise repository). The tags, and even more so the needs and use relations between them and the ensuing search function, are the contribution of the approach: they capture the intricate foreknowledge relations between concepts in the domain of learning, enabling to find small items like exercises matching the knowledge of a specific student.

Ivanović and others [6] report on experiences with teaching materials developed and jointly used at various institutions. Their approach to cope with different teaching approaches is that two sets of teaching materials, roughly corresponding to Objects Late and Objects First styles respectively, have been developed. Exercises are mostly tightly coupled to other teaching materials such as textbooks or presentations. Their experience supports the claim that aiming for compatibility with all approaches entails limiting the materials to exercises only. This is feasible, provided there is automated support for selecting exercises.

Armstrong [1] presented a taxonomy of elemental object-oriented concepts. Not surprisingly, these concepts resemble the knowledge items present in the system presented here. However, the paper does not go into relations like the ones in this system's knowledge graph.

A knowledge graph superficially similar to the one described here was proposed by Hubwieser [4]. In particular, his distinction between hard and soft prerequisites resembles the needs and uses relations. However, his graph has a different purpose: it is intended to identify and order knowledge for curriculum design rather than for matching student skills and exercise demands. Because of this difference in purpose, Hubwieser's knowledge items are much larger than the ones this paper aims at: they do not represent a single concept, but correspond roughly to chapters in a course. Moreover, as Hubwieser's graph is not intended for automatic retrieval of exercises, no attempt is made to remove cyclic dependencies.

Because Hubwieser's graph is intended to be used to support the large-scale design of a course, it does not concern itself with small differences in presentation order. On the other hand, we aim for a system that is usable with any pre-existing set of teaching materials. One point

[2] This use case does not depend on the graph and can be considered irrelevant from the point of view of this paper. It is included to give a more complete picture of the system's use.

where this ambition influences the graph is around the concept *method*, where it was necessary to introduce quite fine-grained tags such as *method with parameters*.

It would be interesting to investigate connections between Hubwieser's work and the present authors': a good question is whether we can show that the system developed in this paper fits all curricula that satisfy the prerequisites in Hubwieser's graph. Furthermore, it could be investigated how the structure of a curriculum based on these prerequisites could be used to, possibly automatically, order a set of tagged exercises to fit that curriculum.

Hubwieser and Mühling [5] present an approach to assess student competencies for OO programming. This addresses more than a prior knowledge characterisation and would be an interesting extension to pursue for more detailed student/exercise matching – with maybe some emphasis on the student side.

The theory of knowledge spaces [2] was developed for the purpose of assessing the knowledge of individuals. To this end, a body of knowledge is formalized as a set of 'notions' that are comparable to the knowledge items in this paper. However, there are important differences. One is that Doignon and Falmange do not assume that every notion has a unique set of predecessors, while the needs relation does have this property, a reflection of the highly hierarchical nature of the subject area being modelled. Another difference is that the notions are identified with problem types, which leads to a great proliferation of notions – hundreds even in the case of elementary subjects, according to Falmange, Cosyn, Doignon and Thiéry [3]. As the examples in the latter paper show, the resulting graphs are too complicated for humans to understand or memorize. That is not problematic for their envisaged use in knowledge assessment, but for the present purpose human comprehension is essential (see Use case A). Because we use graph nodes for concepts rather than exercise types, there are fewer of them: the whole collection of programming exercises at both universities can be described by only 75 nodes. Finally, in the literature on knowledge spaces there is only one type of precedence relation, masking the distinction between mastery and glancing acquaintance, which we have found necessary for realistic and useful tagging of exercises. These differences do not constitute an implicit criticism of knowledge spaces (or, indeed, of our own work), but follow from the different purposes for which both approaches were developed.

Meyer and Pedroni [7, 8] present a wide ranging approach to structuring knowledge of teaching domains, notably OO programming: the emphasis is on the structuring. The approach considers three types of knowledge units (in increasing level of granularity): notions, trucs (Testable, Reusable Units of Cognition), and clusters. In addition, it defines several types of relationships between the entities. Their *notion* with the *requires* relation corresponds closest to *knowledge item* with the needs relation. The relation *refines* resembles the isa relation. A tool, TrucStudio, is provided in [9] that supports course management based on this structure.

It would be interesting to investigate how matching students to exercises could be improved using these knowledge units and relations, and the TrucStudio tool – with maybe some emphasis on the exercises side.

The site CodingBat (`https://codingbat.com/java`) also provides a collection of exercises, with very good feedback, but in limited number and scope: mostly for training simple algorithmics and methods. This collection does not have a skills-matching search process such as we have outlined.

11 Conclusions

This paper presented a course independent repository of Java exercises, with a search method and tool to match student skills to exercise demands.

In how far the aim that the repository fits any Java curriculum succeeded is being investigated empirically in the SERF project: the repository is being used with different curricula, and the results will be evaluated. This is ongoing research, at present conducted in the SURF project SERF.

The repository aims to be filled with exercises by its users: we will support and encourage such use.

In the current version, the student/exercise match is based on knowledge items required to solve the exercise. The emphasis in this project was how to use such information for searching. Several of the results mentioned in Section 10 suggest how this information can be employed for other purposes such as curriculum design and the assessment of individual competences.

An exercise should be such that it is clear what it trains: either because a preferred solution is clear from the specified functionality, or because it is specified as an extra requirement, like that the solution should (or should not) use recursion.

To keep the definitions in this paper as well as the implementation of the tool simple, we have deliberately not included some straightforward improvements of user friendliness. Examples of this are the following.

A mistake like putting the same tag in the training set and in the negative tags set results in an empty exercise set. This is an obvious mistake and the tool could warn for this situation or even prevent the searcher to enter it.

The exercise tags are displayed with the exercises with the aim to enable removal of exercises that, additionally, train topics for which the student does not have the necessary knowledge, yet do not appear in the negative tags set – possibly because the student was unaware of the existence of these topics. By definition, the search tags are among the exercise tags; different fonts or colours or grouping could be used to distinguish between these.

The tool can help the teacher in specifying the tag set of an exercise by computing the minimal (i.e., pruned) set. However, it could be the intention of the teacher to specify a non-minimal tag set, so these variant uses should be supported.

Acknowledgements

This work has been done in the context of the SERF project that has been funded by SURF in the ‘stimuleringsregeling open en online onderwijs’ 2018.

References

[1] Deborah Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, 2006.

[2] Jean-Paul Doignon and Jean-Claude Falmagne. Spaces for the assessment of knowledge. *International journal of man-machine studies*, 23(2):175–196, 1985.

[3] Jean-Claude Falmagne, Eric Cosyn, Jean-Paul Doignon, and Nicolas Thiéry. The assessment of knowledge, in theory and in practice. In *Formal concept analysis*, pages 61–79. Springer, 2006.

[4] Peter Hubwieser. Analysis of learning objectives in object oriented programming. In Roland T. Mittermeir and Maciej M. Sysło, editors, *Informatics Education - Supporting Computational Thinking*, pages 142–150. Springer, 2008.

[5] Peter Hubwieser and Andreas Mühling. What students (should) know about object oriented programming. In *Proceedings of the Seventh International Workshop on Computing Education Research*, ICER '11, page 77–84. Association for Computing Machinery, 2011.

[6] Mirjana Ivanović, Zoran Budimac, Anastas Mishev, Klaus Bothe, and Ioan Jurca. Java across different curricula, courses and countries using a common pool of teaching material. *Informatics in Education*, 12(2), 2013.

[7] Bertrand Meyer. Testable, reusable units of cognition. *Computer*, 39(4):20–24, 2006.

[8] Michela Pedroni and Bertrand Meyer. Object-oriented modeling of object-oriented concepts. In *International Conference on Informatics in Secondary Schools-Evolution and Perspectives*, pages 155–169. Springer, 2010.

[9] Michela Pedroni, Manuel Oriol, Bertrand Meyer, Enrico Albonico, and Lukas Angerer. Course management with TrucStudio. *ACM SIGCSE Bulletin*, 40(3):260–264, 2008.

[10] Devshri Roy, Sudeshna Sarkar, and Sujoy Ghose. A comparative study of learning object metadata, learning material repositories, metadata annotation & an automatic metadata annotation tool. *Advances in Semantic Computing*, 2(2010):103–126, 2010.

Appendix: Graph

In these graphs the nodes that do not represent syntax items are called concept nodes. The reason for this distinction is that syntax nodes can be identified automatically, but concepts nodes are not. All abstractions in the sense of this paper are concept nodes, but the opposite is not true. For instance, *recusrion* is a concept node, but it is not an abstraction, as it is not the target of nay isa arrows.

Figure 4: prior-knowledge graph imperative

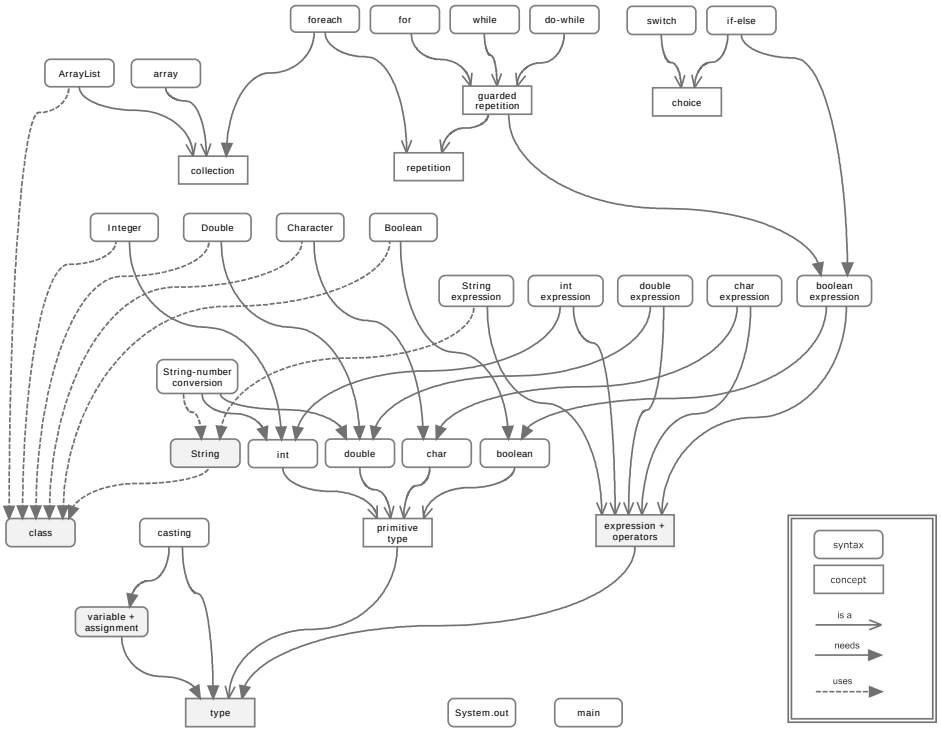

Figure 5: prior-knowledge graph class

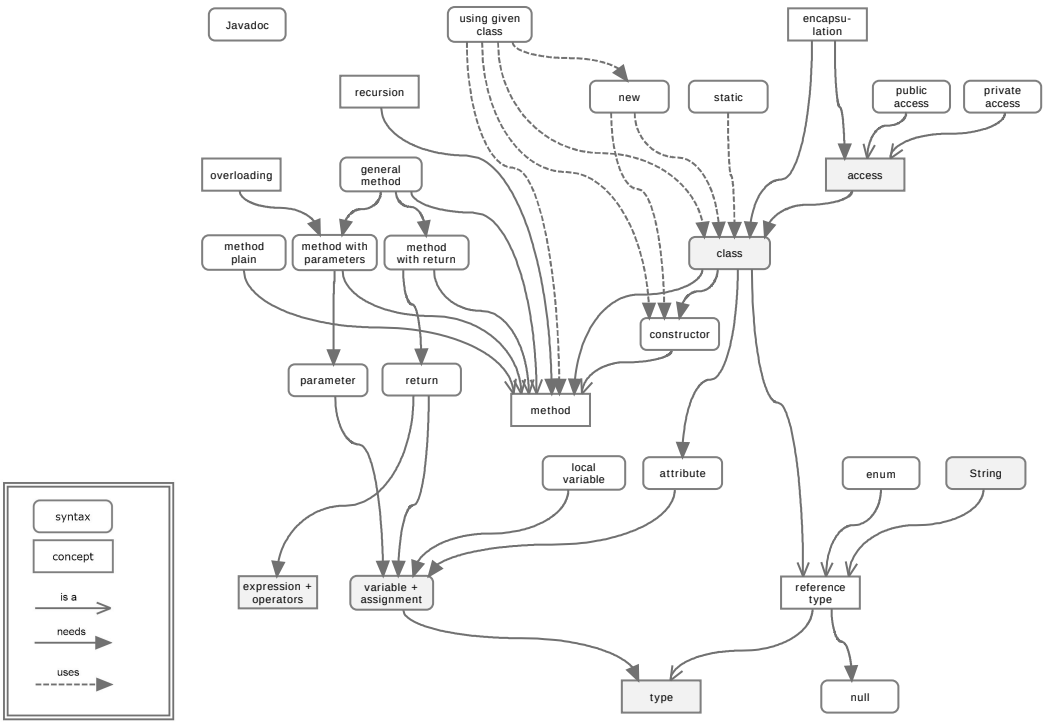

Figure 6: prior-knowledge graph inheritance