Object-oriented modeling of Object-Oriented Concepts

A Case Study in Structuring an Educational Domain

Michela Pedroni and Bertrand Meyer

Chair of Software Engineering, ETH Zurich, Switzerland
{michela.pedroni|bertrand.meyer}@inf.ethz.ch

Abstract. Teaching introductory object-oriented programming presents considerable challenges. Some of these challenges are due to the intrinsic complexity of the subject matter — object-oriented concepts are tightly interrelated and appear in many combinations. The present work describes an approach to modeling educational domains and reports on the results for object-orientation. It analyzes the dependency structure of object-oriented concepts and describes the implications that the high interrelatedness of concepts has on teaching introductory programming.

1 Introduction

One of the strengths of the object-oriented mode of software development is to provide us with a set of powerful and expressive concepts, so powerful and expressive indeed that they can serve beyond their original target area — programming. These concepts, such as classes, message passing, single and multiple inheritance, were initially programming concepts; but they are in fact useful for a far more general purpose: designing systems, modeling systems, and more generally *thinking about* systems. The systems at hand are not even necessarily *software* systems: they can be human and artificial systems of many different kinds. In this work we apply the concepts to a human-centered problem: teaching. We show that it is possible and useful to take ideas originally developed for programming and apply them to the modeling of teaching and learning activities.

Partly by coincidence, the pedagogical target area — the topics for which we hope to support and improve teaching — is programming, and indeed the very form of programming whose results serve as inspiration for the teaching methods and tools: object-oriented programming. The work is then about *object-oriented techniques* for teaching *object-oriented programming.*

Teaching introductory programming is a difficult endeavor. On the side of the learner, programming is a complex activity that involves skills and mental models that many novices struggle to develop during programming courses. On the side of the instructor, teaching programming presents considerable difficulties and has been described as one of the seven grand challenges in computing education [1].

Since the mid 1990s, object-oriented programming has entered the classrooms of introductory programming courses. Many schools have since then adopted an "objects-first" or "objects-early" approach for their CS1 courses, and researchers as well as educators have proposed numerous tools, approaches, and strategies.

It has been asserted that for object-oriented programming "the basic concepts are tightly interrelated and cannot be easily taught and learned in isolation" [2]. This complexity is intrinsic to object-orientation and cannot be removed making it important to develop appropriate tools and processes to handle the resulting challenges.

In programming courses, it seems natural to expose students first to single programming language features (matching the first stage of Linn's "chain of cognitive accomplishments from computer programming instruction" [3]). For object-oriented programming, it is difficult to isolate single language features and to find an initial sequence of single language features (a phenomenon known as "Big Bang problem"). In addition, the tight interrelatedness of O-O concepts results in a higher number of elements to teach, since the instructor must examine not only the elementary concepts but also their possible combinations. This makes it harder to ensure that the teaching sequence meets the prerequisites at all times and that a course covers all facets of a concept.

This work describes a modeling approach and the supporting tool for modeling educational domains through their main concepts and the relations between these concepts, and its application to the educational domain of introductory programming.

2 Truc framework

A course will never be specified as precisely and rigorously as, for example, a computer program. Still, applying modeling techniques partly imitated from software and other engineering disciplines can help meet some of the challenges of course design, in particular for object-oriented programming.

The Truc framework [4] used in this work models educational domains and identifies structural dependencies between concepts. It extends the idea of Truc (Testable, Reusable Unit of Cognition) [5] by adding two additional types of knowledge units. The final model then uses three types of knowledge units (in increasing level of granularity): *notions*, *trucs*, and *clusters*. In addition, it defines several types of relationships between the entities.

At the highest level, a *cluster* is a collection of trucs and other clusters representing a particular knowledge area. A truc belongs to exactly one cluster; the set of clusters forms a hierarchical structure in a directed acyclic graph.

At the medium level, a *truc* is "a collection of concepts, operational skills and assessment criteria" [5]. Its description follows a standardized scheme with sections on technical properties (for example, its role in the field, benefits of applying it, and a summary) and pedagogical properties (such as common confusions and sample exam questions). To help instructors check that their teaching material addresses the misconceptions of students, we have extended the original "common confusions" section [5] with *recommendations* applicable to teaching material such as slides.

The most elementary unit, *notion*, "represents a single concept or operational skill or facet of a concept" [4]. Since the key unit of granularity of the model is truc, every notion belongs to exactly one truc. A truc may have a central notion, which then bears the same name. In our example pedagogical domain, examples of notions within a "feature call" truc are: the central notion "feature call" (capturing the general idea of a method call instruction), "multi-dot feature call" (calls of the form o1.o2.o3.f), and "unqualified feature call" (method calls without an explicit target).

To capture the dependency structure of the knowledge units, the Truc framework defines two types of relations between notions. A *requires* link captures that understanding a notion requires knowing another notion. This relation is comparable to the client relationship between classes in object-oriented systems. A *refines* link expresses that a notion is a specialization of another notion; it is comparable to the inheritance mechanism in object-oriented systems. A refined notion implicitly inherits all the *requires* links from its ancestor, but may also introduce additional ones. For simplicity, the methodology prohibits *refines* links across truc boundaries.

Dependencies at the notion level contribute to dependencies at the truc level: a truc A *depends* on another truc B if any of its notions *requires* a notion of B. Since each truc contains a set of notions, the trucs and notions define a two-layered graph. The graph provides the domain model for the modeling of courses and their associated lectures as a sequence of covered notions. Figure 1 shows an extract of the **truc-notion graph**[1] for object-oriented programming. It includes the direct dependencies of truc Feature call and their notions. The textual description of an example truc is available in Appendix A.

The TrucStudio[2] [6] Pedagogical Development Environment supports the Truc approach. It automatically deduces the truc dependencies from the notion requirements; it provides a graphical representation of the domain model (such as the one produced for the truc *Feature call* shown in Figure 1) and a view of courses as diagrams. Additionally, it offers a customizable output generation mechanism to produce Web pages and ontology files, supports the analysis of transitive dependencies and cycles on notion and truc level, and reports prerequisite violation within a course.

3 Model of object-oriented programming

Several articles and standards have guided the work of selecting concepts and skills that can serve as a starting point for defining the trucs of OOP. In particular, the article on "the quarks of object-oriented development" [7] identifies *inheritance*, *object*, *class*, *encapsulation*, *method*, *message passing*, *polymorphism*, and *abstraction* as "quarks". Except for *abstraction*, all of these quarks appear

[1] To prevent misunderstandings related to the entity type "cluster", we use the name *truc-notion graph* instead of *clustered notions graph* as found in an earlier article [4].

[2] Available at `http://trucstudio.origo.ethz.ch`

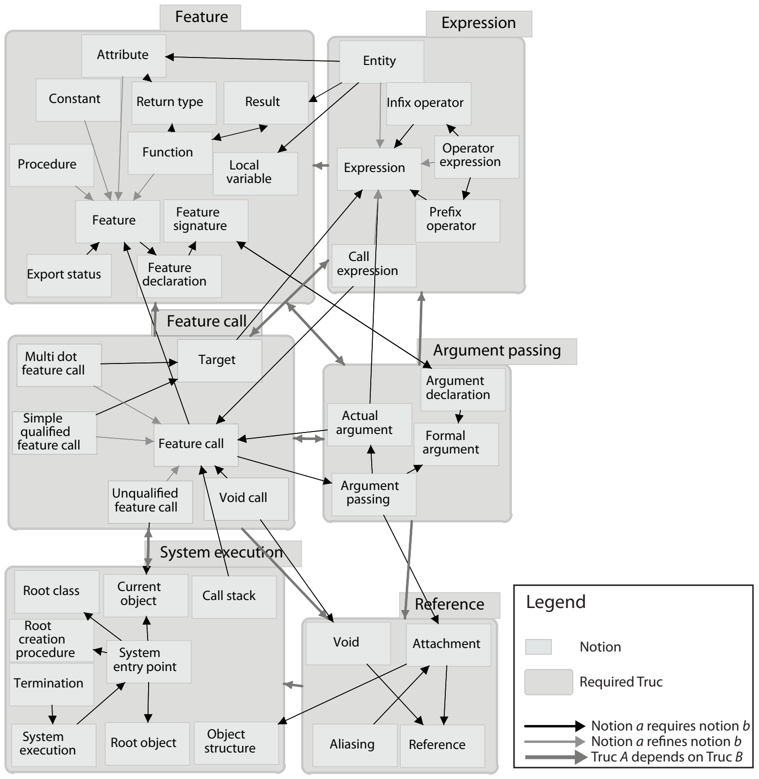


Fig. 1: Dependencies of the “Feature call” truc

as trucs (*encapsulation* under the name *information hiding* and *message passing* as *feature call*).

An experiment by Sanders et al. [8] contrasts the expert view of the quarks with the view of students who recently had studied object-oriented programming. They asked them to draw concept maps [9] that summarize their knowledge of OOP. The most commonly mentioned concepts are *class*, *method*, *instance*, *variable*, and *object*. Other commonly found concepts (implicitly or explicitly) are *data/attribute/instance variable*, *inheritance*, and *encapsulation*. The developed trucs contain all of these concepts; *instance* is integrated in the *object* truc and *data/attribute/instance variable* in the *feature* truc.

Schulte and Bennedsen [10] carried out a study in 2006 where they asked computer science teachers from high schools, colleges, and universities in various countries to rate the difficulty, relevance, and cognitive level of 28 programming topics. They refer to a set of other studies [11,12,13] that helped develop the list of topics. The topics with highest relevance are *selection and iteration*, *simple data structures*, *parameters*, *scope*, *object and class* and *syntax*. The trucs in our model cover these topics, except for *syntax*.

The ACM curricular initiative CC2001 [13] defines the body of knowledge of computer science by specifying 14 knowledge areas ranging from Discrete

Structures, Programming Fundamentals, to Social and Professional Issues. Each knowledge area contains a set of units, which hold a set of topics. It also defines six curricular models for introductory courses and proposes a syllabus and set of units for each variant. The syllabi and description of knowledge units have also guided the selection of concepts covered by trucs.

The model we have developed for object-oriented programming contains the two clusters *Object-oriented programming* and *Data structures* with 28 trucs: *Algorithm, Argument passing, Array, Assignment, Class, Conditional, Deferred class, Design by Contract, Dynamic binding, Expression, Feature, Feature call, Genericity, Hash table, Information Hiding, Inheritance, Instruction, Linked list, Loop, Multiple inheritance, Object, Object creation, Polymorphism, Primitive type, Recursion, Reference, Stack, System execution.* The trucs cover concepts ranging from imperative to object-oriented programming and simple data structures. They contain 147 notions with 196 *requires* and 39 *refines* links. These links result in 85 direct dependencies between trucs. The entire model is available as Web pages and as a TrucStudio project at `http://se.ethz.ch/people/pedroni/trucs`.

4 Analysis of the dependency structure

The first part of this section analyzes the transitive dependencies and cycles as present in our developed domain model. As the domain model represents our view of object-oriented programming and is influenced by our context (in particular the programming language we use, Eiffel), we present a comparison to a model developed by another instructor using Java in the second part.

4.1 Transitive dependencies

The analysis of the dependency structure relies on the transitive (direct and indirect) dependencies resulting from the truc-notion graph of our model for object-oriented programming. The discussion distinguishes between outgoing and incoming links. The analysis of outgoing links organizes the trucs according to the number of their dependencies (prerequisites). This gives an intuition of a truc's place in a course; trucs with many dependencies are likely to appear towards the end, while trucs with few dependencies will probably appear at the beginning. With the incoming links of trucs, the focus shifts to the number of trucs that rely on a given one. This gives an indication of a trucs' importance; if many trucs rely on it, then it is probably central to teaching programming and will reappear throughout a course. Table 1 presents the trucs grouped by their transitive outgoing dependencies: if a set of trucs shares their dependencies, then they are listed in one row.

Outgoing links. The first row of the table shows a *core group* of trucs. They constitute a minimal set of requirements for all 28 trucs appearing in the model.

Table 1: Overview of transitive truc dependencies

Truc \ Prerequisite	*Core group*	Algorithm	Array	Assignment	Conditional	Deferred class	DbC	Dyn. binding	Genericity	Hash table	Inf. hiding	Inheritance	Instruction	Linked list	Loop	Multiple inh.	Polymorph.	Prim. type	Recursion	Stack	# Outgoing links
Core group: Argument passing, Class, Expression, Feature, Feature call, Object, Object creation, Reference, System execution	x																				9
Assignment, Inheritance, Primitive type	x																				9
Deferred class, Genericity, Multiple inh.	x											x									10
Conditional, Instruction, Loop	x			x	x								x		x			x			14
Algorithm	x			x	x								x		x			x			14
Design by Contract	x											x						x			11
Polymorphism	x			x					x			x									12
Dynamic binding	x			x					x			x					x				13
Information hiding	x			x				x	x			x					x				14
Array, Linked list	x	x		x	x				x			x	x		x			x			17
Hash table	x	x	x	x	x				x			x	x		x			x			18
Stack	x	x	x	x	x				x			x	x	x	x			x			19
Recursion	x	x	x	x	x				x			x	x	x	x			x		x	20
# Incoming links	28	5	3	12	9	0	0	1	8	0	0	12	9	2	9	0	2	10	0	1	

The core group contains nine trucs: *Argument passing*, *Class*, *Expression*, *Feature*, *Feature call*, *Object*, *Object creation*, *Reference*, and *System execution*. Every member of the core group depends on itself and on all other members. This is an indication for cycles in the domain model (see 4.2). The trucs *Assignment*, *Inheritance* and *Primitive type* share the dependencies of the core group. They are not part of the core group, because they are not all mutually dependent.

The second set of trucs with cyclic dependencies consists of *Conditional*, *Loop*, and *Instruction*. They are recursively dependent on each other. Additionally to the core trucs, they depend on *Assignment* and *Primitive type*. *Algorithm* has the same dependencies, but does not recursively depend on itself. This group mostly contains trucs associated to imperative programming.

All remaining trucs require *Inheritance* besides the nine core trucs. This is the only supplemental requirement for *Deferred class*, *Genericity*, and *Multiple inheritance*. *Design by Contract* additionally relies on *Primitive type*, while *Polymorphism* requires *Assignment* and *Genericity* in addition to *Inheritance* and the core trucs. *Dynamic binding* depends on *Polymorphism* and thus includes all its dependencies; similarly, *Information hiding* relies on *Dynamic binding* and shares all its requirements. This group mostly contains advanced object-oriented concepts related to inheritance.

The trucs representing knowledge about data structures combine the dependencies of the imperative programming group with some of the object-oriented group. *Linked list* and *Array*, for example, depend on *Algorithm*, *Conditional*, *Instruction*, *Loop*, and *Primitive type*, as well as on *Inheritance* and *Genericity*. *Hash table* additionally requires *Array*; *Stack* requires both *Array* and *Linked list*; and *Recursion* depends on *Stack*.

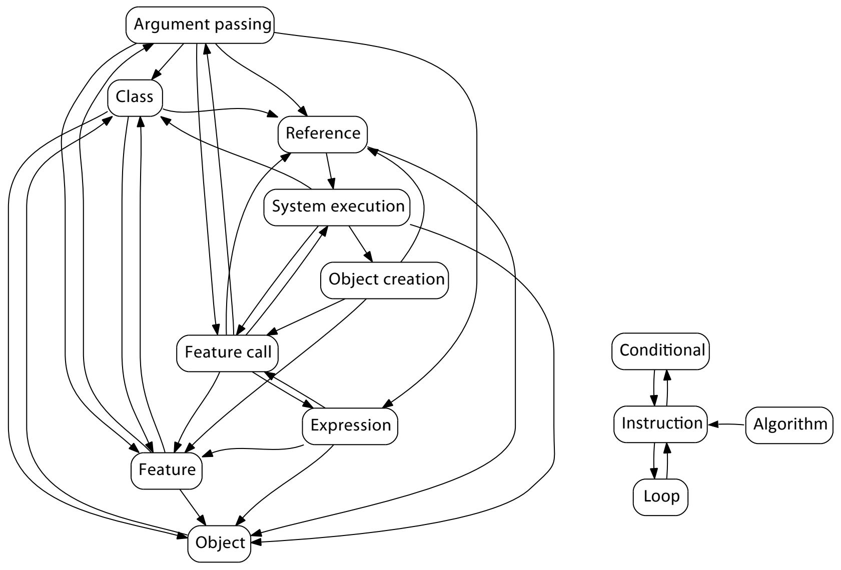
Incoming links. The nine core trucs are a prerequisite for all the trucs of the domain model. This makes them fundamental for teaching object-oriented programming. The second group containing *Assignment*, *Inheritance* and *Primitive type* are required by 12 respectively 10 other trucs (almost half of all trucs) and *Genericity* is a requirement for eight trucs. The second cyclic group containing *Conditional*, *Loop*, *Instruction*, and *Algorithm* provides a basis for nine respectively five trucs. *Polymorphism*, *Dynamic binding*, *Array*, *Linked list*, and *Stack* are prerequisite to one to three trucs. The trucs *Deferred class*, *Multiple inheritance*, *Design by Contract*, *Information hiding*, *Hash table* and *Recursion* do not appear as a requirement for any truc in the model.

Transitive dependencies of notions. The transitive dependencies between notions exhibit characteristics similar to those of the trucs. Ten notions, out of 147, form a core group such that all notions in the model transitively require them. The core group contains the notions *Argument declaration*, *Class*, *Feature*, *Feature declaration*, *Feature signature*, *Formal argument*, *Generating class*, *Instance*, *Object*, and *Type*. Additionally, over half of all notions in the model transitively require the notion *Expression*. 55 notions are not needed by any other notions.

4.2 Cycles

On the notion level, the domain model exhibits five circular dependencies, of which three involve the truc *Argument passing*. One of these cycles is a mutual dependency between the notions *Argument declaration* and *Feature signature*; another cycle consists of the notions *Argument passing*, *Actual argument*, and *Feature call*; and the third cycle contains the notions *Formal argument*, *Type*, *Class*, *Feature*, *Feature declaration*, *Feature signature*, and *Argument declaration*. The fourth cycle on the notion level involves the trucs *Class* and *Object* and illustrates their close interrelatedness via a path through the notions *Class*, *Object*, *Instance*, and *Generating class*, back to notion *Class*. The fifth cycle shows the close connection between *Function* and *Result*.

As indicated in 4.1, two groups of trucs contain cycles in their lists of dependencies. Figure 2(a) shows an extract of the graph with the direct dependencies of the core trucs *Argument passing*, *Feature call*, *Feature*, *Class*, *Expression*, *Object creation*, *Object*, *Reference*, and *System execution*. This subgraph exhibits high interrelatedness between its concepts; in particular, there are multiple pairs of mutual dependencies (such as between *Class* and *Object*, and *Feature call* and *Expression*). Figure 2(b) shows the second group of trucs with mutual dependencies that connect *Instruction* to, separately, *Conditional* and *Loop*.



(a) Core group (b) Imperative group

Fig. 2: Cyclic dependencies

4.3 Comparison with another model

Our use of Eiffel to teach programming has some bearing on the model of object-oriented programming. The choice of trucs and notions, the relationships between notions, and the descriptions of the trucs reflect this particular choice. The model may not, as a result, reflect a generally accepted image of object-orientation and it is not the only form of object-oriented programming.

To find out which properties, in particular cyclic and transitive dependencies, might be artifacts of our course's choices, we asked another instructor teaching introductory object-oriented programming with Java to model parts of his teaching. His domain model includes the entities required to represent the first three lectures of his introductory Java course. It contains a cluster *Programming* with the three trucs *Programming language, Memory management*, and *Program* and a second cluster *Object-orientation in Java* with the 12 trucs *Data type, Object, Method, Variable, Polymorphism, Compilation unit, Instruction, Expression, Access modifier, Conditional, Loop*, and *Identifier*. The trucs contain 67 notions with 19 *refines* links and 33 *requires* links. Appendix B shows the model as a truc-notion graph. The notion dependencies are incomplete.

A comparison of this model to ours shows that they cover similar notions, but their distribution amongst trucs varies. For example, the Java truc *Method* combines sets of notions from our trucs *Feature* and *Argument passing* with single notions of *System execution*. A similar pattern is visible for other trucs, such as the Java truc *Object*, which subsumes our truc *Object* and includes single notions found in our trucs *Feature call*, *Reference*, and *Object creation*. Our model has no truc conforming to the Java truc *Access modifier*, but its notions are integrated in truc *Feature*. Additionally, certain trucs have different names although covering similar notions. For example, the Java truc *Polymorphism* conforms to our truc *Inheritance* and *Compilation unit* conforms to *Class*.

Analysis of the transitive dependencies of the Java model results in a core group containing the trucs *Data type*, *Method*, *Object*, and *Variable*. These trucs are suppliers for about half of the trucs in the model. In particular, they are a prerequisite to themselves and to the trucs *Instruction*, *Polymorphism*, *Memory management*, and *Access modifier*.

There are no cycles on the notion level in this model, but four cycles exist at the truc level: *Method* and *Data type* as well as *Object* and *Variable* are mutually dependent; additionally, there is one cycle containing the trucs *Method*, *Variable*, and *Object*, and one containing *Method*, *Data type*, and *Object*.

A few differences exist between the two models. In the Java model, truc *Compilation unit* containing the *Class* notion is not part of the cyclic group, while in our model *Class* is part of the core group. On the other hand, the truc *Data type* conforming to our truc *Primitive type* is part of the core group. This is due to the notion *Command line argument* in the *Method* truc requesting the notion *Array* of truc *Data type*.

Another difference is that most of the other parts of the model are unconnected. This is probably due to the incomplete nature of the model.

The Java model exhibits similar characteristics with respect to transitive dependencies and cycles as ours. The most striking similarity is the existence of a group of core trucs that are mutually dependent and that are fundamental for a large portion of the remaining trucs. This suggests that our model has a broader reach than just our course.

5 Implications for teaching

Instructors face many challenges when designing courses or textbooks. Besides pedagogical finesse for presenting material adapted to students' skills and interests, they must demonstrate the ability to structure the material in a sound sequence. This task is particularly difficult if the domain of teaching is object-oriented programming, due to the high interrelatedness of its concepts [14,15], what Caspersen calls "one of the most challenging inherent complexities of object-orientation" [2, p. 78]. It complicates finding a starting point where no prerequisites are necessary, and raises the challenges of how to cover the entire subject area and how to order the concepts without prerequisite violations.

Analysis of the truc and notion graphs in this article confirms Caspersen's observation, but narrows it down to a core group of nine closely interrelated concepts. This group contains the trucs *Argument passing*, *Class*, *Expression*, *Feature*, *Feature call*, *Object*, *Object creation*, *Reference*, and *System execution*. Their transitive and recursive dependencies show that they belong in the initial phase of an objects-first course.

On the notion level, the ten notions *Argument declaration*, *Class*, *Feature*, *Feature declaration*, *Feature signature*, *Formal argument*, *Generating class*, *Instance*, *Object*, and *Type* have similar properties with respect to their dependencies as the nine core trucs. With the exception of *Argument passing* and *Formal argument*, which can be omitted if only features without arguments are covered, it seems necessary to introduce them together.

The circular dependencies in the truc and notion model indicate that teaching object-oriented programming requires a spiral model; "A curriculum as it develops should revisit these basic ideas repeatedly, building upon them until the student has grasped the full formal apparatus that goes with them" [16]. The detected cycles also confirm the existence of the "Big Bang problem" [2].

The Inverted Curriculum [17,18] approaches the Big Bang problem by using a large body of supporting software that, through information hiding and inheritance, allows the initial examples and exercises used in class to rely on advanced mechanisms without introducing them formally yet. For example, it first introduces feature calls on predefined objects inherited from an ancestor class. The difficulty of the Inverted Curriculum approach is that the preparation of the software framework and all examples and exercises needs to happen before the students receive the software. This produces the need for more planning and may lead to a restricted set of possible exercises.

Another approach to handling the Big Bang problem is to use an "example-driven" approach, where the "progression in a course is defined by increasing complexity of class models rather than being dictated by a bottom-up ordering of language constructs" [19]. For introducing association, for example, this approach first uses recursive 0..1 association (a *PERSON* class having an attribute *married_to*), then it covers 0..$*$ associations (extending *PERSON* with attribute *friends*), and finally it shows associations between different classes. The language constructs and concepts required for understanding the examples are introduced when they are needed (such as collections for recursive 0..$*$ associations).

The concept interrelatedness also makes it difficult to ensure that an existing course is compatible with the prerequisites and covers all the concepts. We have modeled our Introduction to Programming course using TrucStudio and detected one critical prerequisite violation (the *Result* entity is used before introducing *Function*) and five notions missing in the course slides (*Multi-dot feature call*, *Manifest constant*, *Constant*, *Precursor*, and *Polymorphic creation*).

The complexity of O-O concepts also leads to misconceptions in students' understanding when they first learn about them. The created trucs and their common confusions sections help check whether the teaching material addresses these misconceptions. In a first analysis of our teaching material, we have found that it addresses only a small part of the misconceptions. The conjecture is that this phenomenon is not specific to our material, but more general: although a large body of studies on novices' misconceptions is available, it rarely influences actual teaching.

6 Conclusions

This article has presented a modeling approach for educational domains and reported on its application to object-oriented programming. The approach uses knowledge units at three levels: clusters (describing knowledge areas), trucs (describing skills and concepts following from a central idea), and notions (describing single facets of a concept). It also includes links between the entities to capture the dependency structure of a domain.

The resulting domain model for object-oriented programming consists of 28 trucs and 147 notions. The analysis of the dependency structure confirms that the basic object-oriented concepts are tightly interrelated. It identifies a core group of trucs containing *Argument passing*, *Class*, *Expression*, *Feature*, *Feature call*, *Object*, *Object creation*, *Reference*, and *System execution*. The core trucs are mutually dependent; all other trucs in the model rely on them. The core group of trucs indicates that the associated concepts are mostly responsible for the Big Bang problem — the problem of finding a proper order of introduction for the basic concepts of object-oriented programming.

The developed domain model is influenced by our context (in particular by the programming language we use, Eiffel). To ensure that the findings from our domain model apply to a more general context, we have analyzed a second model of object-oriented programming developed by an instructor using Java for his introductory programming course. In spite of differences in the trucs and in links between notions, his model also contains a mutually dependent core group, on which approximately half of all trucs rely. This indicates that the model for Eiffel is similar to the one for Java and that the results are likely to apply to yet other settings. In the future, we would like to develop a more complete model for Java (possibly based on the Eiffel trucs) and to investigate whether mechanisms from mathematics and computing can help teach tightly coupled notions (as proposed by a reviewer of this article).

Acknowledgements
We thank D. Herding from RWTH Aachen for testing TrucStudio and providing a partial model of object-oriented programming in Java. We are grateful to the anonymous referees for many useful comments.

References

1. McGettrick, A., Boyle, R., Ibbett, R., Lloyd, J., Lovegrove, G., Mander, K.: Grand Challenges in Computing: Education – A Summary. The Computer Journal **48**(1) (2005) 42–48

2. Bennedsen, J., Caspersen, M.E., Kölling, M.: Reflections on the Teaching of Programming. Springer Berlin/Heidelberg (2008)
3. Linn, M.C., Dalbey, J.: Cognitive consequences of programming instruction. In: Studying the Novice Programmer. Lawrence Erlbaum Associates (1989) 57 – 81
4. Pedroni, M., Oriol, M., Meyer, B.: A framework for describing and comparing courses and curricula. SIGCSE Bull. **39**(3) (2007) 131–135
5. Meyer, B.: Testable, reusable units of cognition. IEEE Computer **39**(4) (2006) 20–24
6. Pedroni, M., Oriol, M., Meyer, B., Albonico, E., Angerer, L.: Course management with TrucStudio. In: ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education, New York, NY, USA, ACM (2008) 260–264
7. Armstrong, D.J.: The quarks of object-oriented development. Commun. ACM **49**(2) (2006) 123–128
8. Sanders, K., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J.E., Thomas, L., Zander, C.: Student understanding of object-oriented programming as expressed in concept maps. SIGCSE Bull. **40**(1) (2008) 332–336
9. Novak, J.D., Cañas, A.J.: The theory underlying concept maps and how to construct them. Technical report, IHMC CmapTools, Florida Institute for Human and Machine Cognition (January 2006)
10. Schulte, C., Bennedsen, J.: What do teachers teach in introductory programming? In: ICER '06: Proceedings of the second international workshop on Computing education research, New York, NY, USA, ACM (2006) 17–28
11. Milne, I., Rowe, G.: Difficulties in learning and teaching programming - views of students and tutors. Education and Information Technologies **7**(1) (March 2002) 55 – 66
12. Dale, N.: Content and emphasis in CS1. SIGCSE Bull. **37**(4) (2005) 69–73
13. The Joint Task Force on Computing Curricula: Computing Curricula 2001 (final report). Technical report, ACM and IEEE (December 2001) Available online at: http://www.acm.org/sigcse/cc2001.
14. Gries, D.: A principled approach to teaching OO first. SIGCSE Bull. **40**(1) (2008) 31–35
15. Shultz, G.: Using a restricted subset of Java in the first part of CS1. J. Comput. Small Coll. **23**(1) (2007) 212–218
16. Schwill, A.: Fundamental ideas in computer science. Bulletin European Association for Theoretical Computer Science **53** (1994) 274 – 295
17. Meyer, B.: The outside-in method of teaching introductory programming. In Broy, M., Zamulin, A.V., eds.: Ershov Memorial Conference. Volume 2890 of Lecture Notes in Computer Science., Springer (2003) 66–78
18. Pedroni, M., Meyer, B.: The inverted curriculum in practice. In: SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education, New York, NY, USA, ACM Press (2006) 481–485
19. Bennedsen, J., Caspersen, M.: Model-Driven Programming. In: Reflections on the Teaching of Programming. Springer-Verlag, Berlin, Heidelberg (2008) 116–129

Appendix A – An example truc: Feature call

Alternative names	Method invocation, Message passing
Dependencies	Feature, Object, Argument
Notions	Feature call, Multi dot feature call, Simple qualified feature call, Target, Unqualified feature call, Void call

Summary. Feature call is the mechanism of applying a feature to a target object [Meyer, 2009]. The target may be explicitly defined through an expression, which at run time will be attached to a certain object. If no explicit target is given, the target is the Current object. Feature calls may contain arguments.

Role. Feature call is the "basic mechanism of object-oriented computation". In an object-oriented software system, all computation is achieved by calling features on objects and no software element will ever be executed except as part of a feature call [Meyer, 1997].

Applicability. Need to modify an object or access object data or state.

Benefits.

– Fundamental to create running programs.
– Favors reuse of code by outsourcing a set of instructions into a feature and replacing them by a single feature call.

Pitfalls. Using non-pure queries (queries that change the state of an object) in feature calls may produce side effects. Side effects may lead to mysterious failures that are difficult to locate and fix.

Examples. Consider class *COORDINATE* with attributes *x*: *REAL* and *y*: *REAL* and procedures *set_x_y* (*u*,*v*: *REAL*) and *translate* (*a*,*b*: *REAL*), and a query *distance* (*other*: *COORDINATE*): *REAL*. Assuming that *p1* and *p2* are entities of type *COORDINATE* and have been instantiated, sample feature calls could be:

```
p1.set_x_y(17.2, 3.7)
p2.set_x_y (p1.x, 0)
p1.translate (-0.5, p1.distance (p2))
```

Common confusions.

– **Static-text execution.** Many novices have an incorrect model of control flow for feature calls. It is difficult for them to understand that a feature call results in the suspension of the calling feature (caller), a transfer of the execution control and sometimes data to a *new and unique specimen* of the called feature (callee), then a transfer of control and data back to the caller after the callee has finished. This knowledge is especially important when recursion is introduced. [George, 2000]
– **Availability of features.** "Calling a non-existent feature" seems a recurring mistake [Ng Cheong Vee et al., 2006]. Writing feature calls demands thorough knowledge of what features are available for an entity. This requires looking up the type of the entity and the sufficiently exported features.

- **Wrong target.** Various errors in connection with feature calls originate from specifying wrong targets. For example, students use an unqualified feature call where a qualified feature call is necessary, or they explicitly specify the target to be `Current` for features that are only available for unqualified feature calls. [Ng Cheong Vee et al., 2006]
- **Object state.** Novices have difficulties in understanding that feature call instructions modify object state. [Ragonis and Ben-Ari, 2005]
- **Query as command.** It seems difficult for novice programmers to distinguish between queries and commands for feature call instructions. In certain programming languages it is possible to use a query (with a resulting value/object) as a statement, but the result is then lost. This is a common mistake. [Hristova et al., 2003]

☐ Examples show control flow of feature calls.
☐ Examples include lookup of available features.
☐ Examples include feature calls on composite targets.
☐ Examples show when qualified or unqualified calls are appropriate.
☐ Examples include feature calls that modify object state.
☐ Examples show invalid (or unwanted) use of expression as instruction.

Sample questions. Consider a class *WORD* that has an attribute *word*: *STRING*, a procedure *set_word*(*s*: *STRING*), and a procedure *print* that displays the word on a console. It also has a function *substring*(*i*, *j*: *INTEGER*): *WORD*, which returns a new *WORD* object containing the part of the original word defined through the indices *i* and *j*. Given is an entity *w*: *WORD*. Write an instruction that sets the *word* entity of *w* to "summertime". Then write an instruction that uses the *substring* function to extract "time" from *w*.

Feature Call Bibliography

[George, 2000] George, C. E. (2000). Erosi – visualising recursion and discovering new errors. *SIGCSE Bull.*, 32(1):305?309.

[Hristova et al., 2003] Hristova, M., Misra, A., Rutter, M., and Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. *SIGCSE Bull.*, 35(1):153–156.

[Meyer, 1997] Meyer, B. (1997). *Object-Oriented Software Construction.* Prentice-Hall, 2nd edition.

[Meyer, 2009] Meyer, B. (2009). *Touch of class: Learning to program well with objects and contracts.* Springer.

[Ng Cheong Vee et al., 2006] Ng Cheong Vee, M.-H., Meyer, B., and Mannock, K. L. (2006). Empirical study of novice errors and error paths in object-oriented programming. In *7th Annual HEA-ICS conference*, Dublin, Ireland.

[Ragonis and Ben-Ari, 2005] Ragonis, N. and Ben-Ari, M. (2005). On understanding the statics and dynamics of object-oriented programs. *SIGCSE Bull.*, 37(1):226–230.

Appendix B – Java model of object-oriented programming

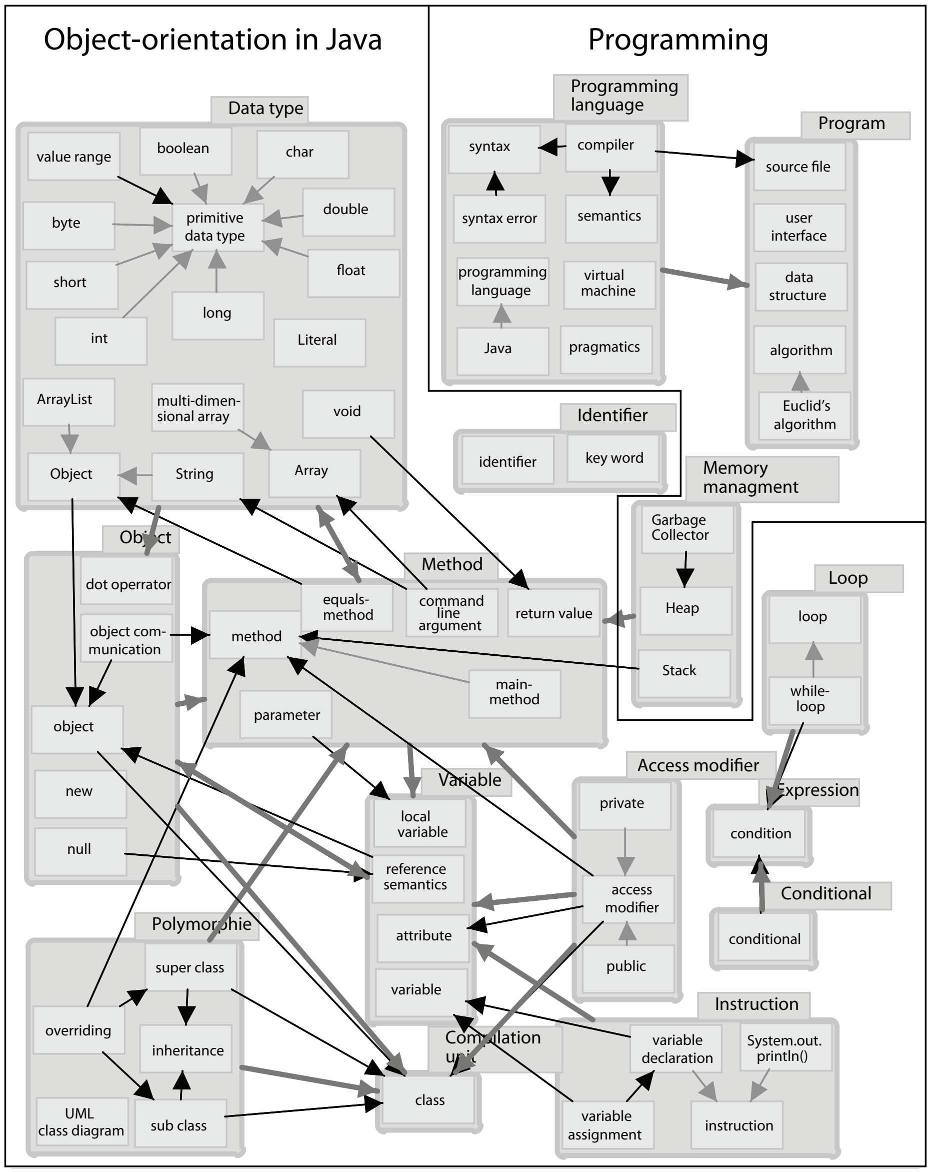


Fig. 3: truc-notion graph of Introduction to Programming at RWTH Aachen (done by D. Herding, German terms translated into English)