Ferreira, W., Baldassarre, M.T., Soares, S. Codex: A metamodel ontology to guide the execution of coding experiments
(2018) Computer Standards and Interfaces, 59, pp. 35-44.

DOI: 10.1016/j.csi.2018.02.003

Computer Standards & Interfaces

Codex: A metamodel ontology to guide the execution of coding experiments

Waldemar Ferreira[a,*], Maria Teresa Baldassarre[b], Sergio Soares[a,c]

[a] *Federal University of Pernambuco – CIn, Av. Prof. Moraes Rego, Recife 1235, Brazil*
[b] *University of Bari – DIB, Piazza Umberto I, 1, Bari, Italy*
[c] *SENAI Innovation Institute for ICT, Av. Norte Miguel Arraes, 539, Recife, Brazil*

A R T I C L E I N F O

Keywords:
Metamodel
Experiment
Software engineering
Coding experiment
Ontology

A B S T R A C T

Background: Experiments have been conducted in many domains of software engineering (SE). *Objective:* This paper presents a metamodel, the Codex metamodel, containing standard concepts found in any coding experiment. We classify coding experiments as any SE experiment where participants have to carry out coding activities (construct, test, debug, and forth). *Method:* The paper presents results of an exploratory study that proposes a metamodel for the domain of coding experiments. Besides, we present how our metamodel specifies a real coding experiment. *Results:* Our metamodel for coding experiments was modeled with few elements, and it can precisely describe all coding activities. *Conclusions:* Our metamodel facilitates the development of a tool to support executing a coding experiment.

1. Introduction

In software engineering (SE), a key issue is how to develop software and deliver it faster, better, and cheaper. Many solutions have been discussed in SE contexts for decades, such as spiral development [1], agile development [2], and DevOps [3]. These solutions have brought many benefits to SE. However, few empirical studies support them [4].

Since the 80's, the SE community has been discussing how to provide better support to empirical research, in particular to experiments [5]. Many researchers have proposed several approaches to aid experiments in SE [6–8]. In this work, we focus only on approaches able to support experiment specification [9–11].

Despite all the benefits provided by current approaches to support experiment specification, they fall into same limitations. Aiming at providing support for any SE experiment, they lack features to precisely specify specificities in a specific SE area. To illustrate, we can take as examples two SE experiments, one on HCI (Human–computer Interface) [12] and evaluating two coding techniques [13]. The former focuses on user interactions, and the relevant variables are mouse clicks and screen records. In the following experiment, the relevant variables are changed lines of code and time spent to develop them. In both experiments, a precise specification of such information is pertinent to plan, execute, or even replicate them. With the current approaches to specifying experiments, all the variables mentioned above are listed in the same format (usually in natural language). It can lead to an inevitable semantic loss [14].

An alternative to precisely specifying experiments involving software development are languages tailored only to describe software development processes, such as BPMN [15], BPEL [16], and SPEM [17]. In fact, these languages enable researchers to precisely specify all experiment tasks involving coding activities (code, artifacts, tests, documents, and so forth.). However, they fail in specifying relevant information concerning the experiment domain such as variables and measurements. In other words, they lack what approaches that support experiment specifications have.

Despite the importance of each solution for SE experiments or software developments, there is no standardized modeling language available for describing experiments in software development. In this paper, we address the research question of which concepts are in common with these two areas (SE experiment or software development) and how they are related to each other. To this end, we investigate languages to specify SE experiments and popular languages to specify the software development process. Our contribution is a metamodel specifying only essential data to execute or replicate coding experiments. Therefore, our solution allows (partial or full) automation of some experiment procedures (such as environment configuration and data collection).

In a previous work, we proposed a meta-ontology to specify domain-specific ontologies for SE experiments [18]. As a proof of concept, we proposed an ontology that was specific for coding experiments. This ontology merged all common concepts in approaches to specifying experiments and software development. However, we noticed that each specific ontology became easily verbose and redundant. To mitigate this limitation, we simplified the ontology. We call this new approach as

* Corresponding author.
E-mail address: wpfn@cin.ufpe.br (W. Ferreira).

Table 1
Models for SE experiment scoping and planning phases.

	Reference	Year	Paradigm
ESEML	Cartaxo et al. [11]	2012	DSL
ExpDSL	Freire et al. [21]	2013	DSL
Exper Ontology	Garcia et al. [9]	2008	Ontology
eSEE	Travassos et al. [10]	2004	Ontology

Coding Experiment metamodel (Codex metamodel). We decided to propose a metamodel instead of ontology since our approach focuses on a specific solution domain. Therefore, the models in compliance with our metamodel can be used by Integrated Development Environments (IDEs) to configure and monitor coding activities [19] easier, lead to less intrusive coding experiments, and more integrated experiments in day to day tasks, as recommended by Wohlin [20].

The rest of the paper is composed as follows. Section 2 presents the related work. Section 3 presents the research design. Sections 4–6 presents our metamodel proposition. Section 7 discusses our evaluation results. Section 9 summarizes the paper and offers suggestions for further research.

2. Related work

We classified the related work into two categories: (i) models for SE experiments and (ii) models for software development process. The following sections present relevant literature falling into each category.

2.1. Experiment models

In this section, we only summarize our finding focusing on characteristics of interesting in our discussion. Table 1 presents the identified MDE approaches.

ESEML (Empirical Software Engineering Modeling Language) is a DSL and tool to specify experiment plans in SE. The authors describe the DSL following a model also called ESEML. According to the authors, the language enables a researcher to represent all relevant information, while the tool allows an automated generation of experimental plans in PDF [11].

ExpDSL (Experiment Domain-Specific Language) also comprises a DLS and tool. However, besides supporting experiment procedure specification, the tool also monitors the experiment execution. The model to specify experiments in ExpDSL has four views. The process view is responsible for defining the activities, artifacts, and roles. The metric view describes the metrics collected during the experiment execution. The experimental plan view describes information such as the factors or the statistical design. Finally, the questionnaire view represents all surveys to collect quantitative and qualitative data from participants.

ExperOntology (Experiment Ontology) is an ontology whose concepts were created to accommodate SE experiment representation. It aims to facilitate the reviewing and understanding of experimental lab packages. The ontology comprises two levels of details. The first refers to the most general controlled experiment concepts (similar to ExpDSL's experimental plan view). The second level focuses only in laboratory package. Similar to ESEML and ExpDSL, there is a tool based on this ontology to support SE experiment execution [22]. eSEE (Experimental Software Engineering Environment) is an infrastructure to manage knowledge about SE experiment definition, planning, execution, and packaging. There are two critical components in this infrastructure: the glossary and ontologies. The first aims at establishing a standard terminology in experimental SE area. Ontologies represent the formalization of the knowledge expressed in the glossary's list of terms. Moreover, this work has tool support [10].

All approaches presented in this section focus on describing general experiment characteristics, such as variables, hypothesis, goals, etc. They do not have any mechanism to specify domain-specific characteristics of coding experiments in detail, such as artifact dependency, code, and tests. A precise specification of such characteristics is fundamental to provide automatic support for coding experiments. For instance, Müller and Höfer [23] carried out an experiment comparing experts and novices when using test-driven development. The previously presented approaches only are able to specify part of observed variables (time spent in each task and changed files). However, some essential observed variables are the number of executed test methods, and how many of them passed or failed during the experiment execution. Such variables are specified by the general-purpose approaches as plain text. It is difficult to provide an automatic support to this experiment. On other hand, this variable can be precisely specified by our solution as we present in Section 6.2.

2.2. Software process modeling languages

García-Borgoñon et al. [24] performed a systematic mapping study to identify Software Process Modeling Languages (SPML). The authors identified more than 40 languages. We highlight only languages electable to specify coding experiment characteristics:

- Business Process Modeling Notation (BPMN) is a standardized notation for creating visual models of business or organizational processes [15].
- Software Process Engineering Metamodel (SPEM) is defined as a profile (UML) by the Object Management Group [17].
- Business Process Execution Language (BPEL) is an OASIS standard [16]. This language is an executable language for specifying actions within business processes with web services.

All these languages were adopted by various enterprises to specify their development process. However, regarding the experiment process (as described by Wohlin et al. [6]), these languages may be considered as a nemesis to the approaches presented in Section 2.1. The SPMLs are powerful enough to specify precisely all domain-specific characteristics in coding experiments. However, they lack means to specify information related to some experiment concepts, such as observed variables and treatments.

3. Research design

Considering the previous section, we conclude that there is a lack of standard specification for SE experiments and software development processes. Aiming at fulfilling this gap, we analyzed coding experiments in literature and models for SE experiments and development processes Borges et al. [25]. Following up from this result, (1) we elicited meaning and developed knowledge into concepts used within both areas and (2) we identified patterns combining the identified concepts.

3.1. Methods

To address the first point, we gathered coding experiments and we analyzed each of them, together with the solutions presented in Section 2. Our goal in this phase consisted in looking for how such solutions are being used. As a result, we generated a set of clusters which encapsulated all identified general concepts and their relationships. To address the second point, we adopted an approach based on Muehlen and Recker [26]. In this work, the authors obtained a significant set of models to analysis. Their first step as to create an Excel spread sheet counting the type of constructs in use per model. Each occurrence of a construct was marked as 1, otherwise 0. This coding allowed the authors to treat the individual models as binary strings for further analysis. In coding effort, the authors kept track of the data sources for each model. The resulting tables provided the basis for the application of statistical techniques such as cluster analysis, frequency analysis, covariance analysis and distribution analysis.

Our approach was similar to Zur et al., firstly we created a spread sheet, and we recorded each information relevant to provide automatic support to the experiment. We also recorded the relationships among concepts. We encoded their usage with 1 or 0 (as Zur et al.). After, we applied a hierarchical clustering on the produced data to identify concepts that frequently or rarely occur together. Based on these results, we propose the Codex metamodel.

In data mining and statistics, hierarchical clustering is a method of cluster analysis which seeks to build a hierarchy of clusters. Strategies for hierarchical clustering generally fall into two types: Agglomerative and Divise. Since we are starting from coding experiment's raw data, we adopted Agglometative clustering. This technique follows a "bottom up" approach: each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy. For instance, Wang and Arisholm [27] conducted an experiment where subjects had to modify legacy code. In the paper the authors wrote:

"... For each task in the experiment, the following steps were carried out: Download and unpack a compressed directory containing the Java code to be modified. This step was performed prior to the first maintainability task for the coffee-machine design change tasks (c1c4), because these change tasks were related... ". This whole sentence was recorded and a cluster was created for it.

In another paper, Santos et al. [28] investigated the use of God Class in Java Projects. In this paper the authors wrote: *"... Six small programs were used in the experiment. All of them implement familiar applications or games in Java... ״*. This sentence was also recorded along with its cluster. After creating a cluster to all relevant information, we agglomerated them according to their similarities. In this case, we agglomerated them in a unique cluster named *artifacts*.

Depending on the research perspective, research results may have different interpretations. Our work targets the perspective of planning and carrying out a simple coding experiment. However, we believe that our results are relevant to other research perspectives, such as replications, family of experiments, and meta-studies as well.

4. Modeling approach

After analyzing coding experiments in literature together with models for SE experiments and development processes, we identified eight concept clusters:

1. *standards in empirical studies*: concepts shared by any empirical study [29];
2. *goal*: any information describing results or possible outcomes;
3. *variables*: all variables involved (controlled or observed) in the experiment process [6];
4. *subject description*: experiment sample concepts;
5. *design of experiment* (DoE): concepts regarding the chosen DoE;
6. *tasks and activities*: any information describing experiment tasks;
7. *artifacts and instruments*: any objects used to carry out an experiment or any measurement instruments;
8. *validity evaluation*: any information about threats to experimental validity. More information about our clustering process can be found in Ferreira et al. [30].

All previously cited concepts must be specified in any experiment [31]. However, despite their importance in SE experiments, only some of them provide useful information for automation in coding experiment procedures. After analyzing each cluster content, we identified three clusters describing useful information for coding experiment procedure automation: *variables, artifacts and instruments*, and *tasks and activities*.

Initially, our metamodel incorporated all concepts in each previously mentioned cluster. They were organized in the same hierarchy level. The spreadsheet with concepts relationships pointed out few relationships between *artifacts and instruments* and *tasks and activities* clusters. On the other hand, these clusters are largely related with the *variables* cluster.

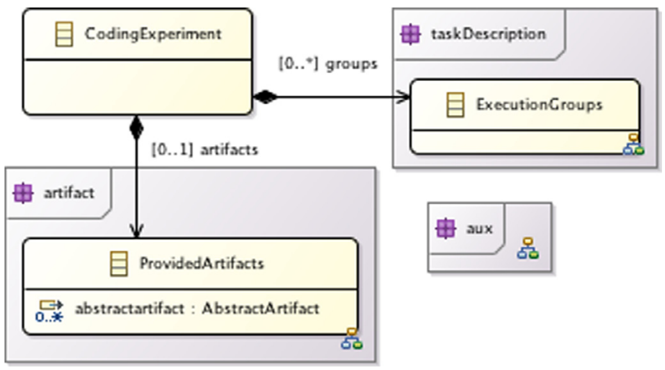


Fig. 1. Overview of Codex metamodel.

This information was crucial when proposing each sub-package in our metamodel.

The following sections present the Codex metamodel concepts and architecture. Besides, we motivate each metamodeling decision using an illustrative running example. Finally, we take the opportunity to point out a mutual synergy between our solution and other solutions (Section 2).

4.1. Codex metamodel overview

Fig. 1 gives an overview of Codex metamodel. The most basic entity in Codex metamodel is *Coding Experiment*. It compiles all main concepts identified by our analysis. As said before, we observed that all relationships between *artifacts and instruments* and *tasks and activities* clusters are via the *variables* cluster. Therefore, we created a package comprising *Artifacts and Instruments* concepts, the *artifact* package (Section 5), and another package specifically for *tasks and activities* concepts, *task description* package (Section 6). All concepts in *variables* clusters were defined inside the two aforementioned packages, and they build a bridge between these packages. In Fig. 1, the *aux* package does not define any particular experiment concept. It supports model concepts in other packages. This package contains the abstract class *Nameable*, which has only one attribute *name*. Therefore, any nameable concept in our metamodel does not have to define an attribute called *name*. It only has to extend *Nameable* class. Another abstract class is *Identifiable* (for classes with unique identification) and *Describable* (for classes with a description). Other metamodels adopt this metamodeling pattern, such as SPEM [17] and BPEL [16].

5. Artifact package

When carrying out a coding experiment, a fundamental information is the artifact definition (such as source codes and tests). In a coding experiment, artifacts represent any article required to carry out a coding experiment activity. Fig. 2 presents our artifact concept representation in Codex metamodel. As shown in Fig. 2, our metamodel classifies artifacts as *Artifact Container, Simple Artifact*, and *Abstract Questionnaire*. Next, we present the first two artifact types (*Artifact Container* and *Simple Artifact*). Due to its complexity to describe all elements in a questionnaire, we have separated this concept in a specific package, *questionnaire* package. Section 5.1 details it.

The abstract class *Artifact Container* represents a grouping of other artifacts (child artifacts). There are two means to assemble artifacts together, as a *Simple Container* or a *Project*. The *Simple Container* assembles child artifacts without any meta-information. For instance, in Stotts et al. [32], participants had access to a zip file containing all archives required to carry out each coding task. A *Simple Container* is enough to model such cases. Another artifact arrangement is the abstract class *Project*. This class assembles child artifacts according to their coding

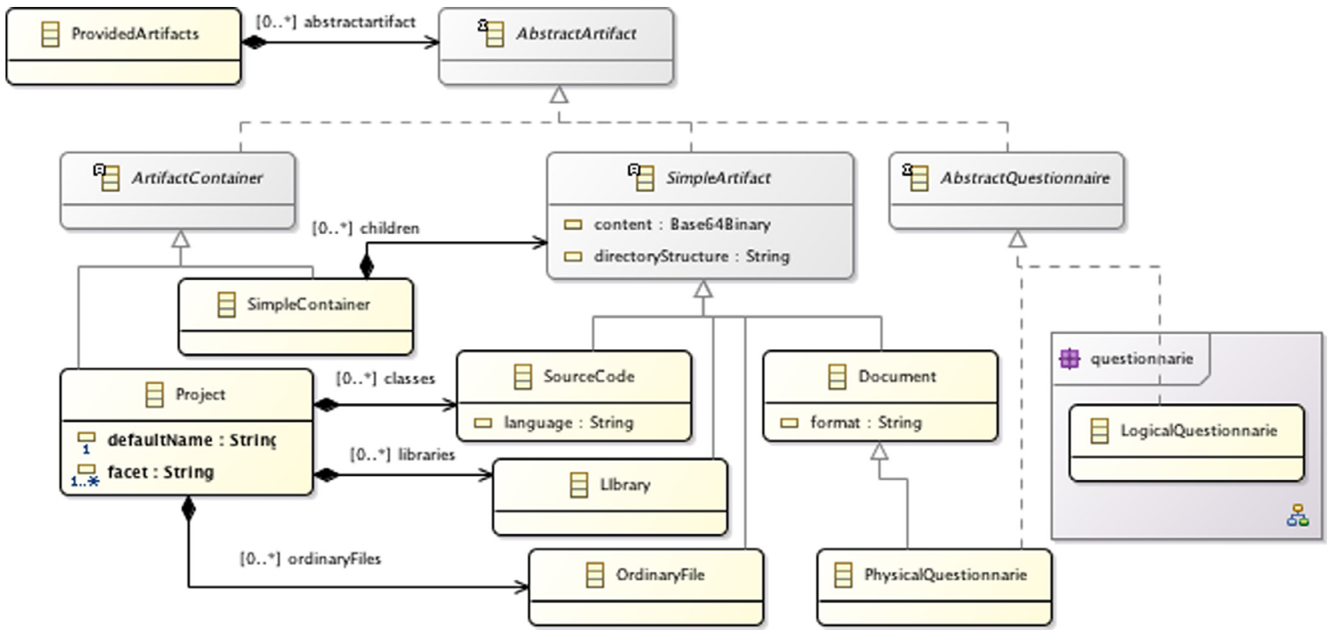


Fig. 2. Overview of artifact metamodel.

project characteristics in an IDE. This class has two attributes: *default-Name*, it suggests an experiment project name; *facet*, it identifies each facet required to configure the subject IDE environment. Some *facet* examples are languages (Java, C, Python, and others), frameworks (Junit, Rails, Cucumber, and others), and infrastructures (Web development, mobile development, and others).

The abstract class *Simple Artifact* represents any atomic artifact. This class has two attributes: *content* which serializes the artifact content following base64 scheme [33] and *directory structure* which describes any required structure (folder hierarchy) for an artifact. In Fig. 2, we see our artifact classification. This classification depicts every possible artifact role in an experiment. Our artifact classification is outlined according to Cattaneo et al. [34] and Silva and Oliveira [35]:

- *Source Code* comprises a collection of computer instructions, written using a human-readable programming language, usually as ordinary text. The attribute *language* identifies the source code's programming language (including tests).
- *Library* represents any artifact required to compile *Source Codes*. Usually, such artifacts are a set of low-level routines used by a compiler to invoke some run-time environment behaviors.
- *Document* is an artifact to inform any crucial arrangement about the experiment. For instance, Santos et al. [28] made a task description available for each participant.
- *Other File* is any other artifact needed in any experiment task. For instance, in Stotts et al. [32], the researchers provided a configuration file to each participant to start the experiment server.

Each source code class can seem like a compilation unit. In principle, a compilation unit is a section of text that can be submitted to the compiler, to create one or more modules of a program Cormen et al. [36]. We prefer to model this concept as a source code since there are many programming paradigms where this definition is not valid (for instance, logical paradigm). Moreover, we did not describe this concept in a fine-grained description, since this description changes based on different programming paradigms. For instance, while OO programs have classes, methods, and attributes, logical programs only have clauses and facts.

5.1. *Questionnaire package*

As anticipated in the previous section, a questionnaire is a particular artifact type. Questionnaires are one of the most common data collection means [6]. Questionnaires can both be provided in paper or electronic formatb(such as e-mail or on-line). We proposed three types of questionnaires:

- Physical Questionnaire. Questionnaires that must be printed and manually filled out by participants.
- Virtual Questionnaire. Questionnaires that are completed in web pages (such as, Google Forms and Survey Monkeys).
- Logical Questionnaire. In this type of questionnaire, each element of the questionnaire must be described in detail. Fig. 3 summarizes our metamodeling approach for basic questionnaire elements. Our metamodel follows survey guidelines in SE [37,38].

Punter et al. [37] define a questionnaire as a set of questions organized in a systematic way for the purpose of eliciting information from respondents. Moreover, questionnaires are classified as structured, semi-structured, or checklist [37]. To model this variability, we proposed the abstract class *Abstract Component*. Any *Abstract Component* subclass shares some characteristic, such as a label. Unlike other solutions (such as ExpDSL [21] and eSEE [10], a questionnaire component can have more than one label. Allowing many labels for a same component allows researchers to specify their questionnaire in different languages [39]. Next, we explain each subclass in *Abstract Component*.

The first *Abstract Component* type is *Block*. It comprises a set of other *Abstract Components*. Block organizes and modulates questionnaires. For instance, Santos et al. [28] used questionnaire blocks to group questions about demography and user experience.

The most fundamental *Abstract Component* type is *Question*. According to Azanza et al. [40], there are more than ten question types. However, in our analysis, we observed two question types represent all questions in coding experiment questionnaires. *Question* types are:

- *Text Answer*. Such questions are designed to encourage a full, meaningful answer using subject's knowledge and feelings. The attribute *isTextBox* specifies if a response should be in few words (a text field) or a full paragraph (text box).

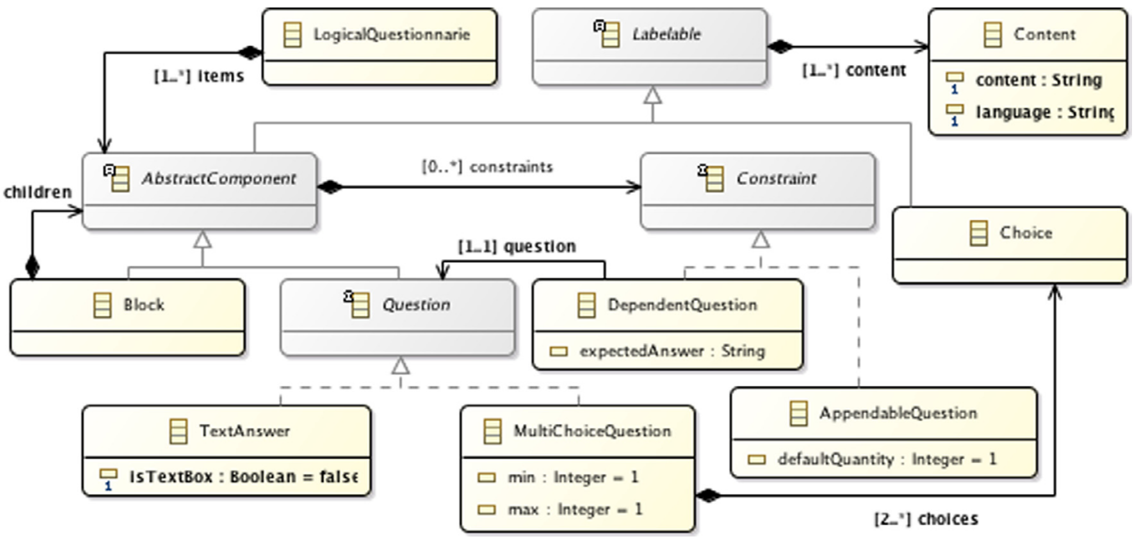


Fig. 3. Questionnaire metamodel overview.

- *Multiple Choice Question*. In such question, participants have to choose their answer within a list of possible answers. The abstract class *Choice* specifies every possible answer.

Our metamodel also allows some question *Constraints*. The first question constraint is *Dependent Question*. This constraint in a question means that the question requires specific answers from other questions. For instance, we can cite the Crime and Safety Survey Questionnaire [41]. If the respondent is a female, she is given a Sexual Victimization Block apart from the Base Questionnaire. The second question constraint is *Appendable Questions*, which represents questions that can be answered many times. For instance, in Santos et al. [28], the participants had to identify classes with too much responsibilities, and for each class the participant had to answer a set of questions.

5.2. Synergy with other solutions

Some authors proposed metamodels only to specify Artifacts. Cattaneo et al. [34] cover Web page artifacts, and Silva and Oliveira [35] proposed an artifact metamodel to represent artifact parts and their relationships. These works are a convenient way to specify coding experiment artifacts. Such approaches inspired our concept of *Simple Artifact* (Section 5). Some SPMLs also propose an artifact specification. BMPN and SPEM classify artifacts according to their roles (input and output artifacts). The SPEM is the most accurate artifact representation with *FragmentDefinition* and *WorkProductDefinition* [17]. These definitions supported our concept of Artifact Content and Simple Artifact.

Concerning models that support SE experiments (Section 2.1), they usually specify artifacts as parameters or controlled variables. However, the ExpDSL and eSEE have entities to specify coding experiment artifacts. In ExpDSL, there is an entity called Artifact, which comprises a name, a description, and a type (input, output or input/output). This representation is similar to artifact specification in BMPN and SPEM. On the other hand, the eSEE proposed a complete sub-ontology to specify software artifacts [10]. It details a software artifact in a fine-grained description, differentiating interfaces, functions, and attributes. We have chosen not to follow this coding artifact description since this description is too bound with OO and imperative software paradigm.

To the best of our knowledge, there is no specific metamodel for questionnaires. However, some models to support SE experiments have some packages to specify questionnaires (Section 2.1). In ExperOntology, two classes (Questionnaire and Form) specify questionnaires. Questionnaire specifies demography data collection instruments and Form represents other questionnaires for nondemography data collection [42]. eSEE designs questionnaires in a sub-ontology for qualitative methods. Finally, ExpDSL proposes a view only for questionnaire specification. We tailored our metamodel to minimize variations among these models. Moreover, Codex metamodel is compliant with all these models. A simple M2M transformation can map our entities into other model entities.

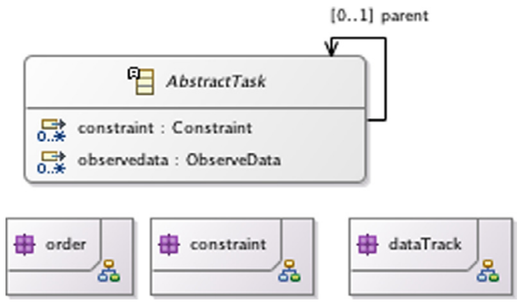


Fig. 4. Coding task overview.

6. Coding task package

According to the Software Engineering Body of Knowledge (SWEBOK) [43], a coding or programming activity may vary from designing, writing, testing, debugging, and maintaining a system. Besides, activity duration also may differ in coding experiments. For instance, a coding inspection may be carried out on one occasion. On the other hand, writing an entire system can be executed during a much longer time span. In such cases, researchers cannot participate in all experiment and data collection activities.

Any coding task specification has to accommodate all possible contexts presented before. Aiming at such goal, we propose *Task* package into Codex metamodel (Fig. 4). We defined one abstract class representing coding tasks and three sub-packages to describe them:

- *Task Order Package* defines how coding tasks are ordered and organized in coding experiment.
- *Task Constraint Package* specifies constraints on coding tasks, for instance, maximum amount of time spent performing a coding task.
- *Task Data Tracking Package* blueprints what data has been collected in each task.

The next sections detail each previously presented sub-package.

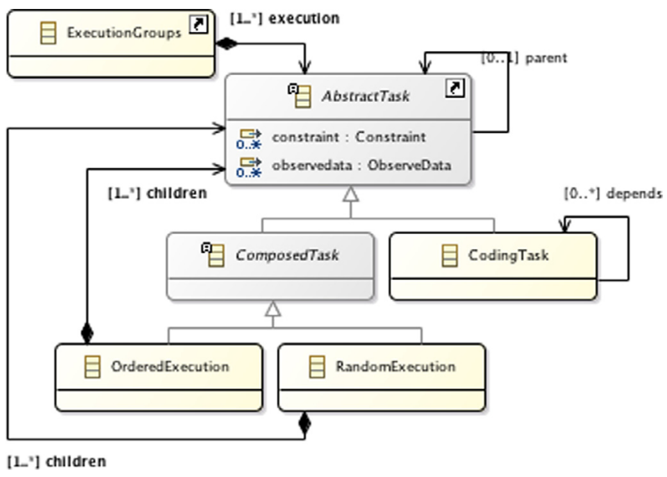


Fig. 5. Task order package overview.

6.1. *Task order package*

This sub-package specifies task description, order, and organization. Fig. 5 presents a metamodel overview. The first abstract class is *ExecutionGroup*. This class identifies which tasks have to be executed in each experiment trial. For instance, an experimental design AB has two *ExecutionGroup*, one indicating tasks *treatment A* and another indicating tasks with *treatment B*.

In our metamodel, task order specification is a BPEL and BPMN metamodel simplification [15,16]. Therefore, there are two action types: *Coding Task* and *Composed Task*. *Coding Task* specifies atomic tasks in coding experiments. This class has an association called *depends*. This association identifies if a *Coding Task* requires artifacts or results from other tasks.

When participants have to perform more than one task, experimenters have to specify a *Composed Task*. This abstract class, in fact, is only an abstraction. We proposed two concrete classes for this abstraction:

- *Ordered Execution*: In this composition, tasks follow a predetermined execution sequence. This task composition is the most common composition type;
- *Random Execution*: This class represents executions without any predetermined order. *Random Task* executions are desirable to avoid some experiment threats [6].

6.2. *Task data tracking package*

In any experiment, data tracking is fundamental to control independent variables, as well as, to observe results in dependent variables [6]. Our metamodel allows tracking specification in most common coding experiment data sources. Fig. 6 presents an overview of *Task Data Tracking* package.

Our metamodel specifies two data tracking means:

- *DataFromAction*: When data sources are participant actions. Usually, in coding experiments, two data types are tracked from participant actions: time stamp or involved artifacts. In our metamodel, *Track Enum* enumeration represents such data tracking types: *Time* and *Artifact. Tracking Actions* have another enumeration, *Moment Enum*. This enumeration specifies which action has to be tracked:
  - *Editing File*: Editing action in any (or a set of) Artifacts.
  - *Executing File*: When participants execute any (or a set of) Artifacts.
  - *Executing Test*: Similar to Executing File, when the participant executes test (or set of tests).
  - *IDE Activity*: Tracking any activity at coding development environment.
  - *Completed Task*: Data is tracked only at the end of a task.
- *DataFromQuestionnaire*: When participants provide responses to questionnaires. The attribute allocation specifies when a questionnaire has to be administered. Moreover, each *DataFromQuestionnaire* is associated to at least one *Artifact Questionnaire*. We detail *Artifact Questionnaire* in coding experiments at Section 5.1.

6.3. *Task constraints*

There are some examples in literature, where some independent variables in coding experiments may be specified as task constraints [13,28,44]. In our metamodel, the abstract class Task Constraints specifies such constraints (Fig. 7). In our analysis, we identified three common constraints in coding experiment tasks:

- *Time Constraint*. Time window constraint applied in any task in a coding experiment. It also specifies task deadlines.
- *IDE Constraint*. Constraint employed within the coding experiment environment. In the current version, our metamodel only specifies required or forbidden Eclipse plug-ins. However, some metamodel extensions can include more constraints.
- *Test Constraint*. Defining success on a task is not easy. In some examples in literature, researchers define a set of tests to ensure task success. Our metamodel implemented such scenarios with *Test Constraint*. This constraint is associated with a *Source Coding* (or a set of) identifying tests. Then, a participant only finishes a task when all tests pass.

6.4. *Synergy with other solutions*

As stated previously, BPMN and BPEL are our task definition and composition pillars. In principle, *Abstract Task* is equivalent to *Activity* in BPMN. And, BPEL does not have any task representation. However, there is a standard extension, BPEL4People [45] where Abstract Task is equivalent *WS-HumanTask*. Regarding the models to specify experiments, only ExpDSL and eSEE allow task definition and order. Moreover, similar to SPMLs, they are equivalent to *Abstract Task*. However, a surprising finding is a lack of random task order specification in both SPML and models to specify experiments.

Regarding data tracking, measurement is a central role in many standards and models such as ISO 15504 and ISO 12207 [46–48], including SPML and models to specify experiments. From the methodological perspective, software measurement is backed by a broad range of proposals, like the Goal Question Metric (GQM) method [49], the Practical Software & Systems Measurement (PSM) methodology [50,60,61], and the ISO 15539 [17] and IEEE 1061-1998 [51] standards. However, these is no clear link between these standards and experiment concepts. Models to specify experiments (ESEML, ExpDSL, and ExpOntology) do not make a clear distinction between the observed variables and their measurements. Only eSEE specifies such information in Study Structure sub-ontology [10]. Furthermore, our measurement definition is a simplification of eSEE definition.

7. Metamodel assessment

The main goal of this section is to assess our metamodel with respect to the modeling of coding experiments from the perspective of experimenters. To demonstrate the expressiveness of our metamodel, we have developed a tool to model coding experiments according to our metamodel, *Codex Modelling Tool*. As stated in the introduction, this tool is part of a platform that supports execution of coding experiments. Moreover, the Codex metamodel implements the concepts of coding experiments at core.

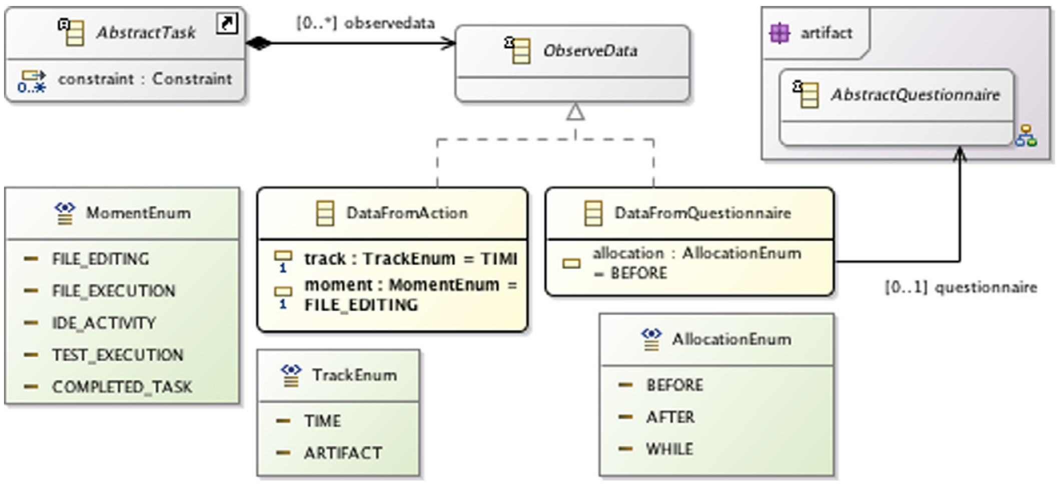


Fig. 6. Overview of task data tracking package.

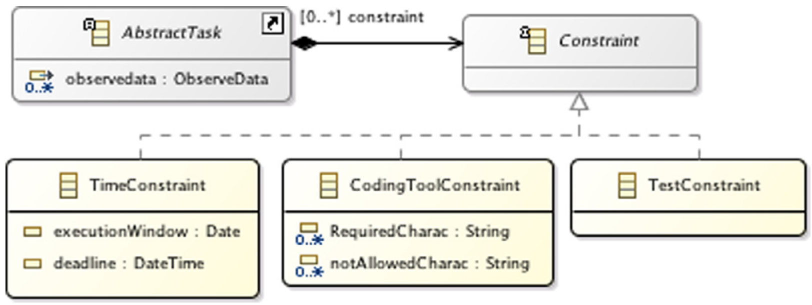


Fig. 7. Task constraints overview.

7.1. Assessment method

We selected three experiments as examples of coding experiments in literature.

- Santos et al. [28]. In this experiment, researchers designed a controlled experiment for investigating the concept of God Class, which is a class or methods with high complexity. According to Fowler and Beck [52], who introduce the concept, a God Class is a class that tries to do a lot (i.e., it has many responsibilities and instance variables). Moreover, Ratiu et al. [53] suggests a God Class detection technique considering the classes that use various data from the nearby classes having a high complexity or low cohesion between methods. Based on this concept, the objectives of Santos et al. [28], this study is to find empirical support to evaluate the impact of a tool in supporting god class detection;
- Accioly et al. [54]. This experiment was conducted with students simulating a test execution environment. While executing test suites, they collected time and reported change requests. This data was collected by an Eclipse plugin developed by the authors.
- Vokáč et al. [13]. In this experiment, the researchers replicated the experiment performed by Prechelt et al. [55], which investigated the question whether it is useful (with respect to maintenance) to design programs using design patterns, even if the actual design problem is simpler than the one solved by the pattern. The replication sought to increase experimental realism by using a real programming environment instead of pen and paper, and by using paid professionals from multiple consultancy companies as subjects.

It is important to mention that our selection also considered the availability of documentation about experiment planning and conduction. The specifications of these experiments following our metamodel are available on-line.[1]

We adopted one criterion for evaluating our metamodel, completeness. This criterion analyzes whether all concepts of coding experiments can be expressed in the Codex metamodel. For the completeness analysis, we investigated whether different aspects of the selected experiments can be properly specified using our metamodel. The following aspects from coding experiments were assessed during the specification process: *Tasks, Artifacts*, and *Measurements* (the clusters presented in Section 4).

8. Results

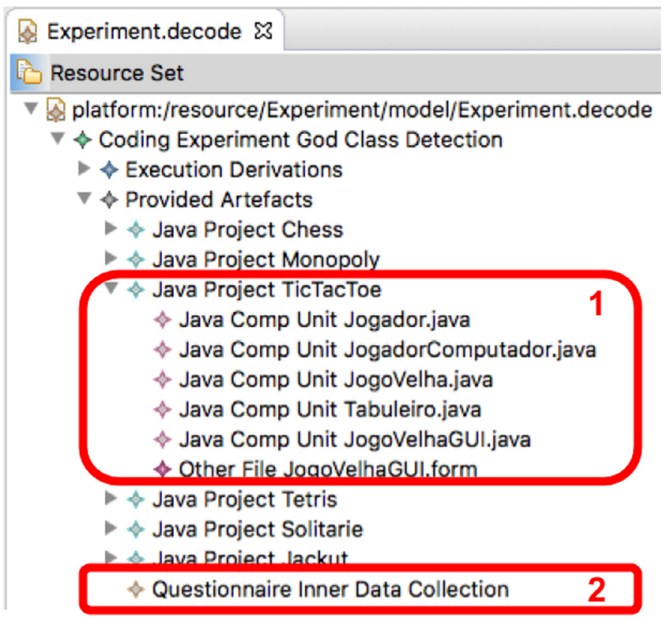
In this section, we present and discuss the results of our study. Section 8.1 examines our findings regarding each cluster. Finally, Section 8.2 presents a discussion about our study and how it can be used by other researchers.

8.1. Analysis of each cluster

The modeling of controlled experiments in our study revealed that the investigated metamodel satisfactorily addressed most of the evaluation criteria. On the other hand, it also exposed improvement opportunities for specific elements and aspects of the Codex metamodel.

This section presents results about the completeness criteria. We considered different experimental aspects that were modeled using Codex metamodel. We show and discuss how to model experiments in our metamodel. Also, we describe how the achieved results can be used to propose improvements for our metamodel.

[1] https://github.com/netuh/DecodePlatformPlugin.



```
Experiment.decode
Resource Set
▼ platform:/resource/Experiment/model/Experiment.decode
  ▼ Coding Experiment God Class Detection
    ▶ Execution Derivations
    ▼ Provided Artefacts
      ▶ Java Project Chess
      ▶ Java Project Monopoly
      ▼ Java Project TicTacToe                              1
          Java Comp Unit Jogador.java
          Java Comp Unit JogadorComputador.java
          Java Comp Unit JogoVelha.java
          Java Comp Unit Tabuleiro.java
          Java Comp Unit JogoVelhaGUI.java
          Other File JogoVelhaGUI.form
      ▶ Java Project Tetris
      ▶ Java Project Solitarie
      ▶ Java Project Jackut
        Questionnaire Inner Data Collection                 2
```

Fig. 8. Example of artifact specification.

8.1.1. *Artifacts*

Our metamodel allowed to define all the artifacts required to perform coding activities in each selected experiment. Fig. 8 presents an overview of the model for artifacts in Santos et al. [28]. Two types of artifacts were involved in this experiment, developing projects and questionnaires. Fig. 8.1 presents a *Project* (Section 5). In this case, a Java *Project* which comprises each required source code (each java file). Originally, the researchers stored on-line all experiment artifacts. However, with a Codex model, it was no longer necessary as the Codex model stores this information together with all information required to carry out an experiment. The same approach was adopted by Vokáč et al. [13] to specify their artifacts.

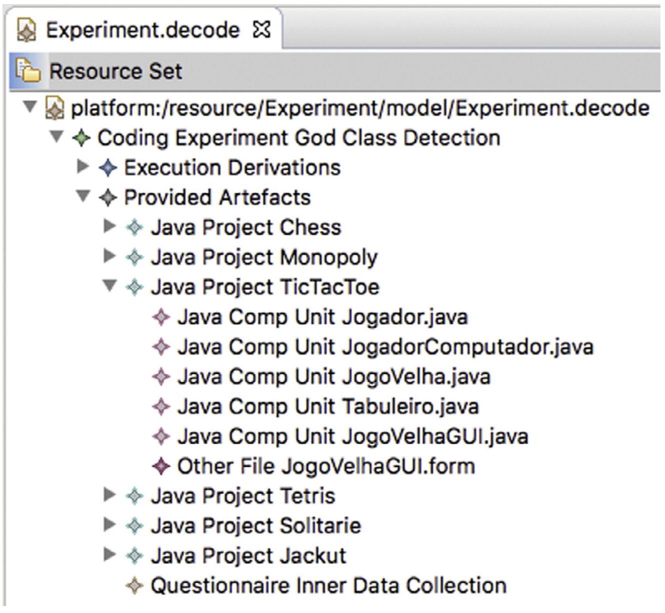
Another artifact present in all selected experiments was questionnaires. Similar to previous mentioned Java artifacts, questionnaires are also included in Codex. Moreover, this model contains data about who and when to apply each questionnaire. Fig. 8.2 presents how the questionnaires in Santos et al. [28] may be specified in our metamodel. In this case, a *Document* represents each questionnaire. However, researchers can specify each questionnaire element (according to Section 5.1).

8.1.2. *Tasks*

As we presented for artifacts, many relevant aspects of coding tasks were also defined in our metamodel. Fig. 9 presents how the tasks in Santos et al. [28] are modeled in our metamodel. We decided to present the tasks in Santos et al. [28], since, in this experiment, part of the participants had to inspect three projects without any tool support and other three with a tool. These two set of tasks were executed in sequence. However, the three inspections with or without the tool were randomly executed. In Vokáč et al. [13] and Accioly et al. [54], all tasks were executed in sequence.

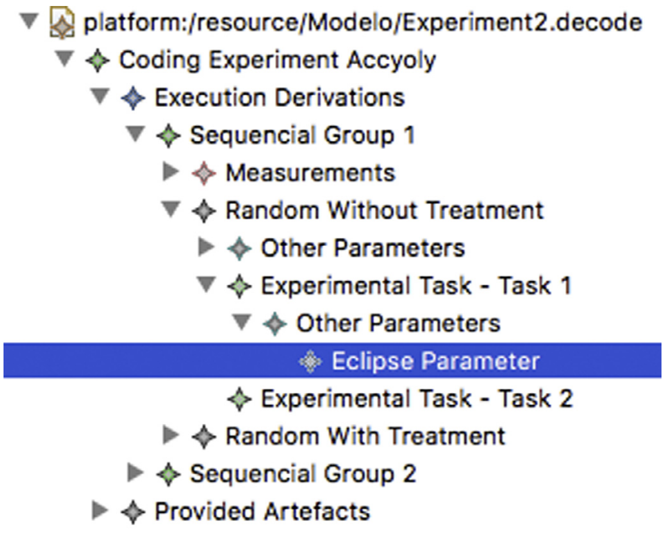
As can be seen in Fig. 9, the derivation execution has two sequential tasks representing each trial in experiment design (*Sequential Group 1* and *Sequential Group 2*). For the sake of simplicity, Fig. 9 only details the *Sequential Group 1*. However, all comments about *Sequential Group 1* are valid for *Sequential Group 2*. Moreover, each group was specified as an *OrderedExecution*; each *OrderedComposition* comprised two *RandomExecution*. Finally, each *RandomExecution* included a set of *Coding Tasks*, connected with corresponding *Project* artifacts (Section 5).

Only Accioly et al. [54] had constraints to task execution. Fig. 10 presents how such constraints are modeled considering our metamodel.



```
Experiment.decode
Resource Set
▼ platform:/resource/Experiment/model/Experiment.decode
  ▼ Coding Experiment God Class Detection
    ▶ Execution Derivations
    ▼ Provided Artefacts
      ▶ Java Project Chess
      ▶ Java Project Monopoly
      ▼ Java Project TicTacToe
          Java Comp Unit Jogador.java
          Java Comp Unit JogadorComputador.java
          Java Comp Unit JogoVelha.java
          Java Comp Unit Tabuleiro.java
          Java Comp Unit JogoVelhaGUI.java
          Other File JogoVelhaGUI.form
      ▶ Java Project Tetris
      ▶ Java Project Solitarie
      ▶ Java Project Jackut
        Questionnaire Inner Data Collection
```

Fig. 9. Task specification in case study.



```
▼ platform:/resource/Modelo/Experiment2.decode
  ▼ Coding Experiment Accyoly
    ▼ Execution Derivations
      ▼ Sequencial Group 1
        ▶ Measurements
        ▼ Random Without Treatment
          ▶ Other Parameters
          ▼ Experimental Task - Task 1
            ▼ Other Parameters
                Eclipse Parameter
            Experimental Task - Task 2
        ▶ Random With Treatment
      ▶ Sequencial Group 2
    ▶ Provided Artefacts
```

Fig. 10. Task specification with test constraints.

As we can see, researchers can attach a test suite to a task, *Test Constraint*. It means that the task may be considered as finished only when such tests pass.

8.1.3. *Measurements*

In Codex metamodel, we can define measurements for each task. During the data collection procedure definition (process element), the researcher can choose to collect data related to dependent variables using our metamodel in different ways, such as: (i) collecting artifacts created or changed during the experiment that should be analyzed by the researcher; (ii) requiring participants to inform the specific data (e.g. number of defects found) during their tasks execution in the experiment; or (iii) including a field in a questionnaire (through cross-reference) to be answered by the participants during the execution of the experiment.

Considering our example Santos et al. [28], Fig. 9 presents how a researcher can specify measurements in tasks. In this experiment, two kinds of data have to be collected, (i) time spent to finish each task and (ii) quantity of actions executed by each participant. The first measurement was specified by a *DataFromAction*, with *Time* at track field and with *IDE_Activity* at moment field (details in Section 6.2). Similarly, for the second measurement, the corresponding *DataFromAction* has *Time* at track field, and *Final Task* at moment field.

The other two experiments (Accioly et al. [54] and Vokáč et al. [13]) collect almost the same kind of variables. However, they also collect the artifacts produced during the experiment execution. Therefore, their models have an extra element: *DataFromAction* which is attached to *Artifacts* and *Completed Task*. As so, once the tasks are completed, all artifacts (produced and changed) during the task execution are part of the results.

8.2. Discussions and lessons learned

In this section, we present and discuss lessons learned related to the results of our study.

Appropriateness Evaluation. In our study, we have tried to address such criterion by specifying three different experiments, among which we modeled one replication Baldassarre et al. [56]. Given this variability of experiments, we have modeled experiments: with diverse experimental designs and executed by different research groups. This variety of the chosen experiments supports us to conclude that the Codex metamodel provided an appropriated characterization for the set of modeled experiments. However, we recognize that it is crucial to model an extensive amount of additional experiments to enlarge the variability of experiments that can be expressed by our metamodel.

Experiment Replication. In spite there are various guidelines for reporting experiments, the fact that they have not been formalized according them limits their replicability Baldassarre et al. [56], Juristo and Vegas [57]. In collaborative experimental research, results from previous studies are needed to transfer knowledge between the involved researchers. In this context, our metamodel provides a mean to formalize the laboratory package for coding experiments and their correspondent replicas. It is one of the tangential contributions of our approach. We believe that it could facilitate information communication and exchange among experimenters, contributing to fill the gap related to providing a complete experiment definition. We modeled one experiment replication, and we observed that when modeling a replica, a researcher can reuse almost all original experiment specification. The changes are localized and related to the following actions: to add or change artifacts, tasks, and/or measurements. We intend to systematically evaluate and explore the Codex metamodel benefits regarding replications in our future work.

Execution Environment. The experiment knowledge formalized with our metamodel is used in our approach Ferreira [58] as input to an experimentation support environment. The environment allows executing an experiment as a workflow that guides the tasks of each participant. The environment supports some experiment functionalities:

- Experiment documentation – As said before, other approaches are most appropriated to document an experiment in SE [30]. However, our metamodel allows a precise description of many relevant characteristics in an experiment plan. This description should provide a formal way to store the plan of several kinds of experiments in software engineering to increase the documentation precision. For our execution environment, the experiment definition in Codex metamodel itself represents this functionality.
- Participant guiding – We developed a tool to aid participants in their experiment execution. The environment guides participants through each task that have to be performed by them. The participants have to finish one task after another (sequentially). The experiment's Codex model specifies the task order.
- Data collection – Each participant produces data that are relevant for the experiment. This data is automatically collected by the environment, such as changed files, or project execution, test passes and failed, and so on. Besides, the environment may provide a means for participants to answer questionnaires (according to their specification). Moreover, the environment is responsible for automatically collecting the spent time for executing each experimental task.
- Gathering of feedback – It is fundamental to gather feedback from participants during experiment execution. A questionnaire is a known instrument to collect feedback. The participants are asked to answer online feedback questionnaires according to the Codex questionnaire definition.

Limitations. Regarding the models to specify extensive experiments in SE. Our metamodel is able to specify all characteristics identified in cluster. However, we should conduct interviews with researchers from other companies and universities to try to understand their views about these models, as well as in other fields, and carry out experiments in practice. For instance, an information relevant to be specified is that only code without errors or faults cannot be accepted when finishing a task.

9. Conclusion

Our study has initially identified the need for a unified metamodel for coding experiments since other alternatives do not have enough expressiveness. After a detailed analysis of these alternatives together with reference to real coding experiments that have been conducted and reported in literature, we have proposed a specific metamodel option: Codex metamodel. This metamodel provides a precise panorama focusing on coding experiment execution. Moreover, this metamodel identifies core elements that enable an automatic support to coding experiment execution. A customized model following our metamodel emphasizes variables, tasks, and artifacts involved when carrying out a coding experiment. Furthermore, it may foster coding experiments since experimentation is a cost-intensive empirical method. Therefore, by facilitating the experiment execution, researchers can focus more efforts in improving the samples, allowing more reliable conclusions [59]. As of now we have specifically applied the model to coding experiments with positive results. We do not exclude that in the future the model will be expanded to other domains as well.

As future work, we intend to develop a tool based on our metamodel. This tool has to configure the subject environment for task execution, as well as, collect data according to experiment specification. Finally, we are going to replicate our case study within this tool to evaluate its relevance in supporting coding experiments.

Acknowledgments

This work is the result of a bilateral agreement between Cin/UFPE and UNIBA. This work was supported by CNPQ, grants 141066/2013-0 and 205613/2014-4. Moreover, it is partially supported by INES, grants CNPq/465614/2014-0, FACEPE/APQ/0388-1.03/14. Sergio Soares is partially supported by CNPq, grant 304499/2016-1.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.csi.2018.02.003.

References

[1] B. Boehm, W.J. Hansen, Spiral Development: Experience, Principles, and Refinements, Technical Report, DTIC Document, 2000.
[2] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, D. Thomas, Manifesto for Agile Software Development, 2001.
[3] M. Httermann, Devops for Developers, first ed., Apress, Berkely, CA, USA, 2012.
[4] T. Dybå, T. Dingsøyr, Empirical studies of agile software development: a systematic review, Inf. Softw. Technol. 50 (9–10) (2008) 833–859, doi:10.1016/j.infsof.2008.01.006.
[5] V. Basili, R. Selby, D. Hutchens, Experimentation in software engineering, IEEE Trans. Softw. Eng. SE-12 (7) (1986) 733–743, doi:10.1109/TSE.1986.6312975.
[6] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer-Verlag, Berlin, Heidelberg, 2012, doi:10.1007/978-3-642-29044-2.

[7] N. Juristo, A.M. Moreno, Basics of Software Engineering Experimentation, first ed., Springer Publishing Company, Incorporated, 2010.
[8] V. Basili, The Experimental Paradigm in Software Engineering, in: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 706 LNCS, Springer, 2015, pp. 3–12.
[9] R. Garcia, H. Erika Nina, E. Barbosa, J. Maldonado, An ontology for controlled experiments on software engineering, in: Proceedings of 20th International Conference on Software Engineering and Knowledge Engineering, SEKE, 2008, pp. 685–690.
[10] G. Travassos, P. Dos Santos, P. Mian, A. Dias Neto, J. Biolchini, An environment to support large scale experimentation in software engineering, in: Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, ICECCS, 2008, pp. 193–202, doi:10.1109/ICECCS.2008.30.
[11] B. Cartaxo, I. Costa, D. Abrantes, A. Santos, S. Soares, V. Garcia, Eseml – empirical software engineering modeling language, in: Proceedings of ACM Workshop on Domain-Specific Modeling, DSM, 2012, pp. 55–60, doi:10.1145/2420918.2420933.
[12] I. Benbasat, A. Dexter, P. Masulis, An experimental study of the human/computer interface, Commun. ACM 24 (11) (1981) 752–762, doi:10.1145/358790.358795.
[13] M. Vokáč, W. Tichy, D.I. Sjøberg, E. Arisholm, M. Aldrin, A controlled experiment comparing the maintainability of programs designed with and without design patternsa replication in a real programming environment, Empir. Softw. Eng. 9 (3) (2004) 149–195, doi:10.1023/B:EMSE.0000027778.69251.1f.
[14] D. Davidson, G. Harman, Semantics of Natural Language, 40, Springer Science & Business Media, 2012.
[15] M. Chinosi, A. Trombetta, Bpmn: an introduction to the standard, Comput. Stand. Interfaces 34 (1) (2012) 124–134, doi:10.1016/j.csi.2011.06.002.
[16] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D.F. Ferguson, Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.
[17] S. OMG, O. Notation, Software & Systems Process Engineering Meta-model Specification, OMG Std., Rev. 2(2008).
[18] W. Ferreira, M.T. Baldassarre, S. Soares, G. Visaggio, Toward a meta-ontology for accurate ontologies to specify domain specific experiments in software engineering, in: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 9459, 2015, pp. 455–470, doi:10.1007/978-3-319-26844.
[19] C. Calero, F. Ruiz, M. Piattini, Ontologies for Software Engineering and Software Technology, Springer, Berlin, Heidelberg, 2006, doi:10.1007/3-540-34518-3.
[20] C. Wohlin, Empirical software engineering research with industry: top 10 challenges, in: Proceedings of the 1st International Workshop on Conducting Empirical Studies in Industry, CESI '14, IEEE Press, Piscataway, NJ, USA, 2013, pp. 43–46.
[21] M. Freire, P. Accioly, G. Sizílio, E. Campos Neto, U. Kulesza, E. Aranha, P. Borba, A model-driven approach to specifying and monitoring controlled experiments in software engineering, in: Proceedings of the Product-Focused Software Process Improvement, PROFES, Springer, Berlin, Heidelberg, 2013, pp. 65–79.
[22] L. Scatalon, R. Garcia, R. Correia, Packaging controlled experiments using an evolutionary approach based on ontology, in: Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering, SEKE, 2011, pp. 408–413.
[23] M.M. Müller, A. Höfer, The effect of experience on the test-driven development process, Empir. Softw. Eng. 12 (6) (2007) 593–615, doi:10.1007/s10664-007-9048-2.
[24] L. García-Borgoñon, M. Barcelona, J. García-García, M. Alba, M.J. Escalona, Software process modeling languages: a systematic literature review, Inf. Softw. Technol. 56 (2) (2014) 103–116, doi:10.1016/j.infsof.2013.10.001.
[25] A. Borges, W. Ferreira, E. Barreiros, A. Almeida, L. Fonseca, E. Teixeira, D. Silva, A. Alencar, S. Soares, Support mechanisms to conduct empirical studies in software engineering: a systematic mapping study, in: Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering, EASE, ACM, 2015, pp. 22:1–22:14, doi:10.1145/2745802.2745823.
[26] M.Z. Muehlen, J. Recker, How much language is enough? Theoretical and practical use of the business process modeling notation, in: Proceedings of the 20th International Conference on Advanced Information Systems Engineering, CAiSE '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 465–479, doi:10.1007/978-3-540-69534-9_35.
[27] A. Wang, E. Arisholm, The effect of task order on the maintainability of object-oriented software, Inf. Softw. Technol. 51 (2) (2009) 293–305, doi:10.1016/j.infsof.2008.03.005.
[28] J. Santos, M. De Mendonúa, C. Silva, An exploratory study to investigate the impact of conceptualization in god class detection, in: Proceedings of ACM International Conference Proceeding Series, 2013, pp. 48–59, doi:10.1145/2460999.2461007.
[29] A. Donovan, R. Laudan, Scrutinizing Science: Empirical Studies of Scientific Change, 193, Springer Science & Business Media, 2012.
[30] W. Ferreira, M.T. Baldassarre, S. Soares, B. Cartaxo, G. Visaggio, A comparative study of model-driven approaches for scoping and planning experiments, in: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, EASE'17, ACM, New York, NY, USA, 2017, pp. 78–87, doi:10.1145/3084226.3084258.
[31] A. Jedlitschka, M. Ciolkowski, D. Pfahl, Reporting Experiments in Software Engineering, Springer, London, 2008, doi:10.1007/978-1-84800-044-5_8.
[32] D. Stotts, L. Williams, N. Nagappan, P. Baheti, D. Jen, A. Jackson, Virtual Teaming: Experiments and Experiences with Distributed Pair Programming, Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2753, Springer, 2003, pp. 129–141.
[33] N. Borenstein, Mime: a portable and robust multimedia format for internet mail, Multimed. Syst. 1 (1) (1993) 29–36.
[34] F. Cattaneo, E. Di Nitto, A. Fuggetta, L. Lavazza, G. Valetto, Managing software artifacts on the web with labyrinth, in: Proceedings of International Conference on Software Engineering, 2000, pp. 746–749, doi:10.1109/ICSE.2000.870486.
[35] M. Silva, T. Oliveira, Towards detailed software artifact specification with spemarti, in: Proceedings of International Conference on Software Engineering, 2011, pp. 213–217, doi:10.1145/1987875.1987912.
[36] T.H. Cormen, C. Stein, R.L. Rivest, C.E. Leiserson, Introduction to Algorithms, second ed., McGraw-Hill Higher Education, 2001.
[37] T. Punter, M. Ciolkowski, B. Freimut, I. John, Conducting on-line surveys in software engineering, in: Proceedings of the International Symposium on Empirical Software Engineering, ISESE, Institute of Electrical and Electronics Engineers Inc., 2003, pp. 80–88, doi:10.1109/ISESE.2003.1237967.
[38] S.L. Pfleeger, B.A. Kitchenham, Principles of survey research: part 1: turning lemons into lemonade, SIGSOFT Softw. Eng. Notes 26 (6) (2001) 16–18, doi:10.1145/505532.505535.
[39] G. Kersten, M. Kersten, W. Rakowski, Software and culture: beyond the internationalization of the interface, J. Global Inf. Manag. 10 (4) (2002) 86–101, doi:10.4018/jgim.2002100105.
[40] M. Azanza, D. Batory, O. Díaz, S. Trujillo, Domain-specific composition of model deltas, in: Proceedings of the Third International Conference on Theory and Practice of Model Transformations, ICMT, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 16–30.
[41] J.F. Devoe, K. Peter, P. Kaufman, A. Miller, M. Noonan, T.D. Snyder, K. Baum, Indicators of School Crime and Safety, National Center for Education Statistics, 2004.
[42] R.W. Root, S. Draper, Questionnaires as a software evaluation tool, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI, ACM, 1983, pp. 83–87, doi:10.1145/800045.801586.
[43] I.C. Society, P. Bourque, R.E. Fairley, Guide to the Software Engineering body of Knowledge (SWEBOK(r)): Version 3.0, third ed., IEEE Computer Society Press, Los Alamitos, CA, USA, 2014.
[44] A. Ko, T. LaToza, M. Burnett, A practical guide to controlled experiments of software engineering tools with human participants, Empir. Softw. Eng. 20 (1) (2013) 110–141, doi:10.1007/s10664-013-9279-3.
[45] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. König, F. Leymann, R. Müller, G. Pfau, et al., Ws-bpel Extension for People (bpel4people), Version 1.0, Web address: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/wsbpel4people/BPEL4People_v1.pdf. (2007).
[46] I. ISO, IEC 15504-2: Information Technology-process Assessment. Part 2: Performing an Assessment, 2004.
[47] C. Pardo, F. Pino, F. García, F.R. Romero, M. Piattini, M.T. Baldassarre, Hprocesstool: a support tool in the harmonization of multiple reference models, in: Proceedings of the 2011 International Conference on Computational Science and its Applications, ICCSA'11, Volume Part V, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 370–382.
[48] F. Pino, M. Baldassarre, M. Piattini, G. Visaggio, D. Caivano, Mapping software acquisition practices from ISO 12207 and CMMI, in: Proceedings of Communications in Computer and Information Science, 69, 2010, pp. 234–247, doi:10.1007/978-3-642-14819-4_17.
[49] V.R. Basili, H.D. Rombach, The tame project: towards improvement-oriented software environments, IEEE Trans. Softw. Eng. 14 (6) (1988) 758–773, doi:10.1109/32.6156.
[50] J. McGarry, Practical Software Measurement: Objective Information for Decision Makers, Addison-Wesley Professional, 2002.
[51] S. S. E. S. Committee, et al., IEEE std 1061-1998 IEEE Standard for a Software Quality Metrics Methodology, 1998.
[52] M. Fowler, K. Beck, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 1999.
[53] D. Ratiu, R. Marinescu, S. Ducasse, T. Gırba, Evolution-enriched detection of god classes, in: Proceedings of the 2nd CAVIS, 2004, pp. 3–7.
[54] P. Accioly, P. Borba, R. Bonifacio, Comparing two black-box testing strategies for software product lines, in: Proceedings of the 2012 Sixth Brazilian Symposium on Software Components, Architectures and Reuse, SBCARS '12, IEEE Computer Society, 2012, pp. 1–10, doi:10.1109/SBCARS.2012.17.
[55] L. Prechelt, B. Unger, W. Tichy, P. Brssler, L. Votta, A controlled experiment in maintenance comparing design patterns to simpler solutions, IEEE Trans. Softw. Eng. 27 (12) (2001) 1134–1144, doi:10.1109/32.988711.
[56] M.T. Baldassarre, J. Carver, O. Dieste, N. Juristo, Replication types: towards a shared taxonomy, in: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, EASE, ACM, 2014, pp. 18:1–18:4, doi:10.1145/2601248.2601299.
[57] N. Juristo, S. Vegas, Using differences among replications of software engineering experiments to gain knowledge, in: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM, IEEE Computer Society, 2009, pp. 356–366, doi:10.1109/ESEM.2009.5314236.
[58] W. Ferreira, Together we are stronger: facilitating the conduction of distributed human-oriented experiments, in: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, ACM, 2014, pp. 56:1–56:4, doi:10.1145/2601248.2613083.
[59] M. Jørgensen, T. Dybå, K. Liestøl, D.I. Sjøberg, Incorrect results in software engineering experiments: how to improve research practices, J. Syst. Softw. 116 (2016) 133–145, doi:10.1016/j.jss.2015.03.065.
[60] D. Caivano, Continuous software process improvement through statistical process control, in: Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, 2005, pp. 288–293.
[61] M.T. Baldassarre, D. Caivano, G. Visaggio, Software renewal projects estimation using dynamic calibration, IEEE International Conference on Software Maintenance, ICSM (2003) 105–115, doi:10.1109/ICSM.2003.1235411.