"To Program is To Model": Software Development is Stepwise Improvement of Models

Bent Bruun Kristensen, Palle Nowack, Michael Caspersen
Center for Computational Thinking
Aarhus University
Denmark
bbk@mmmi.sdu.dk, {nowack, mec}@cs.au.dk

Abstract— **The paper explores the notion of "To Program is To Model" in the realm of an introductory programming course. We present a number of intended learning outcomes and didactical design principles for the course, and we then describe the course content in terms of the system to be developed as well as the project to be undertaken. Based on this, we illustrate the many different ways software development can be understood, as "To Program is To Model". These reflections utilize a conceptual model in terms of domains and models useful when understanding and discussing software development. Finally we present a set of requirements for students to learn programming as modeling.**

Keywords: Teaching introductory programming, to program is to model, software development, stepwise improvement, domains and models in the software development process

I. Introduction

We interpret the quote "at rejse er at leve" ("to travel is to live") by Hans Christian Andersen [1] as the immediate meaning of "to travel" (that is what you do) is by reflection replaced by a more profound understanding namely "to live" (that is what you in fact do). Consequently the meaning of "to program is to model" is that when you program some system you in fact model the system [2]. And when you through stepwise improvement construct and evaluate your software you in fact understand the meaning of "to program is to model".

The purpose of this paper is to explain the intention and contents of an introductory programming course. The course includes presentations of selected subjects and a project. All though it is a programming course the aspects modeling and design are also included. The intention is that the participants modify and extend parts of an existing software system. The actual system is a specific example but the course outline is general. This teaching approach is motivated and evaluated.

The paper is structured as follows: In Section 2 we outline the intended learning outcomes and corresponding knowledge areas of the course; Section 3 briefly summarizes the didactical design principles applied in the design of the course in order to support the intended learning outcomes ; Sections 3 and 4 describe the system used in the course and the specific project-based approach taken, respectively; and finally Section 6 evaluate and reflect on the overall course, and Section 7 summarize our findings.

II. Intended Learning Outcomes & Knowledge Areas

The intended learning outcome of the course covers the following knowledge areas (ranging from concrete and explicit craftsmanship to abstract thinking and understanding):

A. Programming

The student must be able to conduct an object-oriented software programming process:

- Read and understand programs.
- Change programs.
- Develop new elements of programs.

B. Modeling

The student must be able to explain, develop, and evaluate Object-Oriented software system models (e.g. UML models: use-case model, conceptual model, class diagram, sequence diagrams, etc.) as described in [17, 18].

C. Design

The student must be able to explain and evaluate:

- Object-Oriented software designs including abstract classes, selected and provided applications frameworks and design patterns.
- Functional and non-functional software system qualities.

D. Development Process

The student must be able to explain and use:

- An iterative and incremental software development process with particular focus on stepwise improvement.
- Software system requirements (e.g. using use-cases).

E. To program is to model

The student must be able to explain:

- To program is to model and to model is to understand.
- To stepwise improve software and/or models is to change understanding of a domain and/or a system.

These knowledge areas are not described further in this paper.

III. Didactical Design Principles

A number of didactical design principles have been applied as guidelines when designing the course:

- A learning activity is not (necessarily) the same as a knowledge area.
- Learning activities should:
  - Be application-oriented.
  - Facilitate and guide a consume-before-produce progression through the materials.
  - Include several substantial worked examples.
  - Illustrate stepwise improvement as a general approach to incremental development of artefacts.
- Realism Dilemma: Professionally relevant knowledge areas versus teachable learning activities

A. Knowledge Areas vs. Learning Activities

The learning activities form the toolbox, from which the teacher select, combine, design, and implement his/her particular version of the subject which should be adapted and adjusted to the relevant context (education, level, and individual students). A learning activity may include subject matters from one, multiple, or all of the knowledge areas as illustrated in Fig. 1. A learning activity is comprised by a description for students and teachers, materials and resources, and a process (cookbook) for using the materials in the learning activity.

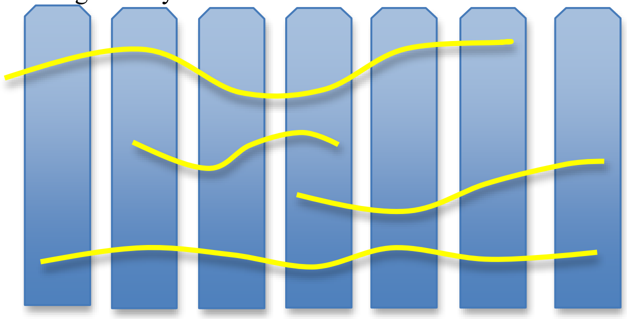

Figure 1. Content Structure Framework: Knowledge Areas (blue columns) versus Learning Activities (yellow lines). From [3].

Example: In Section 5 we present two groups of project activities: "Design and implementation" and "Experiments and evaluation". Each activity within these groups constitutes a learning activity involving multiple knowledge areas.

B. Application-oriented (outside-in)

This means, that we start the various learning activities by introducing well-known or familiar applications, which we then split apart for conceptual and/or technical examination, evaluation, and modification. For motivational reasons, we choose applications based on the criteria, that they must by themselves be naturally appealing to students in our age range. Applications, which they find interesting to use and hopefully to examine and improve.

Example: The eShop application is provided to the students at the start of the course. eShop is presented in Section 4. Students are expected to be familiar with this type of application.

C. From Consumer to Producer

When designing learning activities, we aim at organising the material in such a way that the students experience a *consume-before-produce* progression through the material [4]. Initially, the students act as consumers of an artefact by using and studying it; then, they go on to make first simple and then gradually more complex modifications to the artefact. Eventually, the students may be requested to build similar artefacts from scratch.

The *consume-before-produce* principle —sometimes alternatively characterised as a *use-modify-create* progression— can be applied in many areas. In programming, students can use programs or program modules before they start making modifications and eventually create modules or complete programs on their own. The approach applies equally well to other areas, e.g. modelling and interaction design.

Example: As mentioned, the eShop application described in Section 4 is provided to the students initially in the course. They start out by using the application, then modifying it, and then adding new elements to it. Simultaneously, the students are provided with various UML models, which they use. modify, and create new elements/variants of.

D. Worked Examples

A Worked Example (WE), consisting of a problem statement and a procedure for solving the problem, is an instructional device that provides a problem solution for a learner to study. WEs are meant to illustrate how similar problems might be solved, and WEs are effective instructional tools in many programs, including computing [5]

Example: In the course, we provide the students with multiple worked examples, e.g. the Boundary-Control-Entity design pattern, which is also provided as a framework.

E. Stepwise Improvement

Stepwise improvement is a conceptual framework for incremental development of an artefact [6]. According to stepwise improvement, development takes place in three dimensions: from abstract to concrete, from partial to complete, and from unstructured to structured. Thus, development of an artefact can be characterised as a mixed sequence of refinements, extensions, and restructurings of the artefact.

Example: During the learning activities "Design and implementation", students are asked to extend the system by adding concepts (e.g. Book and Wine) and functionality (e.g. View Basket and Checkout use cases). During "Experiments and extensions", the students are expected to think about (but *not design or implement) necessary and potential refinements and restructurings.*

F. The Realism Dilemma: Professionally relevant knowledge areas versus teachable learning activities

When teaching one must always address a number of dilemmas. One of the most important when teaching programming and software development, is the dilemma between, on one hand, involving realistic, professionally relevant knowledge areas, and on the other hand, to devise teachable learning activities suitable for the level of competence for the involved students.

In this course, the dilemma is that the product and process presented on the one hand must have professional qualities, and on the other hand be illustrative at the given educational level and with the chosen (necessarily limited) educational focus. This requires respect for professional standards, and explicitness about how the dilemma is tackled. For object-oriented example programs, this tension is explored in [19, 20].

Example: Refactoring as an example of the dilemma: The software system, eShop, presented to the student has been iteratively refactored because the software system must be professional — and without refactoring it would be unrealistic. However by refactoring the naiveness and immediate understandability of the software system disappears. Still naiveness and easy understandability may be expected as an essential issue in an introductory programming and modeling experience.

IV. SYSTEM: REQUIREMENTS, DESIGN AND PROGRAM

The software system is an internet shop, eShop: The eShop offers various products including T-shirts. A customer visits the eShop, shops for various products, adds selected products to a basket, and eventually checks out.

A use case diagram and a conceptual model describe the requirements to the eShop. A class diagram and design sequence diagrams support the corresponding design. The program follows the design thoroughly and is illustrated by program extracts.

A. Requirements

The eShop is described in more detail by a use case diagram [7] in Fig. 2. The actors *Customer* and *Administrator* are not described further. For *Update Products* and *Shop Products* the *brief* use case descriptions are:

Update Products: The administrator updates products available at the eShop.

Shop Products: A customer browses products available at the eShop.

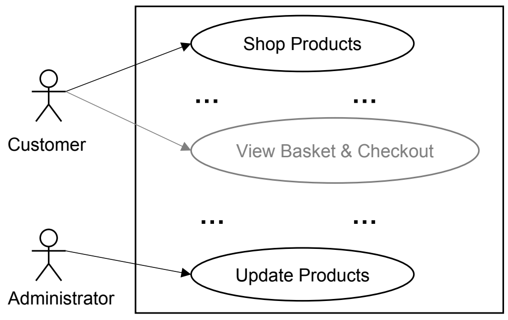

Figure 2. Use Case Diagram

The eShop is described in more detail by an additional conceptual model [8] (domain model in [7]) in Fig. 3. Only the concepts *Customer*, *Product* and *T-shirt* are related in the diagram: *Customer* has a simple (many to many) relation to *Product* whereas *T-shirt* is a specialization of *Product*. Additional potential concepts are also included.

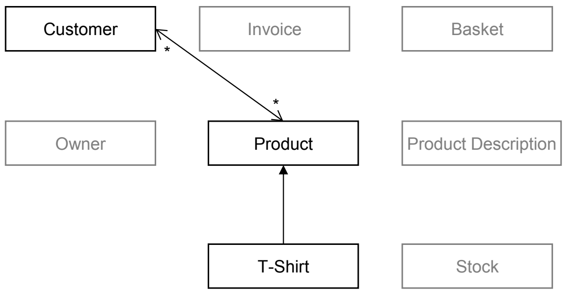

Figure 3. Conceptual Model

B. Design

The *Boundary-Control-Entity* principle (B-C-E) is used to support the software organization and includes concepts/classes in the form of the stereotypes boundary, control and entity. Control classes have access to boundary and entity classes, whereas boundary classes may have access to entity classes:

- *Boundary*: Boundary classes handle the interaction between actors and control classes. Each actor–use case pair, identifies these user interface classes with exactly one boundary class for each pair.
- *Control*: Control classes handle the flow of control for a use-case and are seen as coordinating representation classes. They co-ordinate with entity classes that do the work for them.
- *Entity*: Entity classes model the information handled by the system, and the functionality associated with the information.

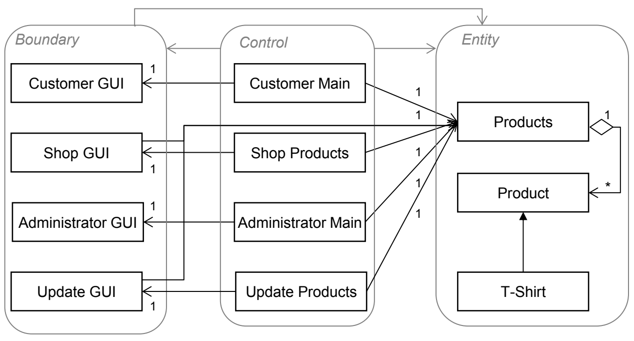

Figure 4. Class Diagram

An application framework [9] for the B-C-E principle with classes `Boundary`, `Control` and `Entity` classes supports the eShop system. The class diagram [7] in Fig. 4 includes specializations (extensions) of boundary, control and entity classes as illustrated. Neither methods nor attributes are included in the diagram due to reasons of simplicity. The classes handle the use cases from Figure 2 as well as main use cases (for selecting between specific use cases) for customer and administrator. Class `Products` aggregates `Product` where `T-Shirt` is a specialization of `Product`. Corresponding specific relations between the classes supply the overall relations between boundary, control and entity.

Fig. 5 is a sequence diagram [7] of the B-C-E principle applied to eShop: The interaction between `Boundary`, `Control` and `Entity` objects is handled at the abstract class level. Fig. 5 illustrates the combinations of these classes with the specialized classes `Shop GUI`, `Shop Products` and `Products`. The framework requires the method invocations shown in **bold** in Fig. 5 to be supplied in the `Shop GUI` and `Shop Products` classes. The `go()` method of the `Shop Products` object is invoked. The object invokes its `setupBoundary()` method from which a `Shop GUI` object is instantiated. The constructor of this object invokes its `relateEntity(Control c)`. Next the object invokes its `setupGUI()` and a reference to this `Shop GUI` object is returned to the `Shop Products` object. The `Shop Products` object then iteratively executes an interaction with the `Shop GUI` object until the lifecycle of the `Shop Products` object ends. The `Shop Products` object invokes `waitFor()` on the `Shop GUI` object and waits. Upon use of the GUI the method `actionPerformed()` is applied to the `Shop GUI` object that invokes its `onActionEvent()` and `proceed()` methods. This makes the `Shop Products` object continue and invoke its `handleAction()` method. If the action performed is `show` the `Product` is retrieved from `Products` and presented and the iteration continues. If the action performed is `cancel` the iteration ends.

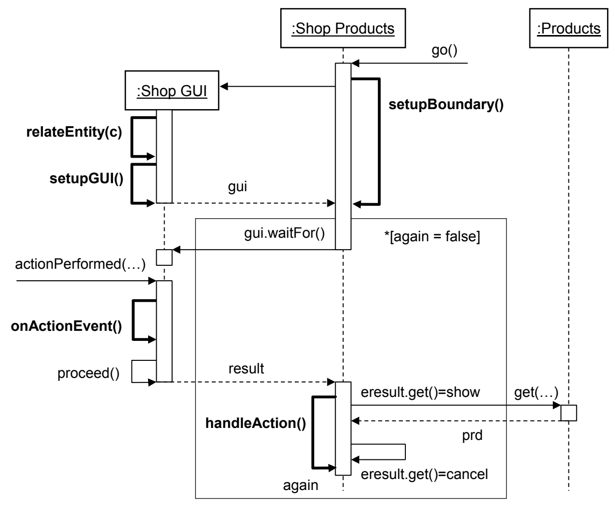

Figure 5. Sequence Diagram: Boundary-Control-Entity

C. *Program*

Class diagram, sequence diagrams and the program in java [10] describe identical artifacts. However the program provides additional details: The abstract classes `Boundary`, `Control` and `Entity` are extended with abstract methods as follows:

Class `Boundary` includes

- `void relateEntity(Control c)` (not an abstract method to be overwritten only if necessary): References to `Entity` objects that must be available in the constructor for the `Boundary` class are achieved from the `Control` object `c`.
- `void setupGUI()`: The form and contents of the GUI for this `Boundary` class is constructed.
- `void onActionEvent()`: The action performed and accompanying text is saved to be available for the `Control` object.

Class `Control` includes

- `setupBoundary()`: The `Boundary` object is created and a reference to the object is returned.
- `boolean handleAction()`: The reaction to user actions through the `Boundary` object is handled.

Class `Entity` has no contents. No database or similar is included and instead various `Entity` objects (i.e. various products) are initialized when the program starts.

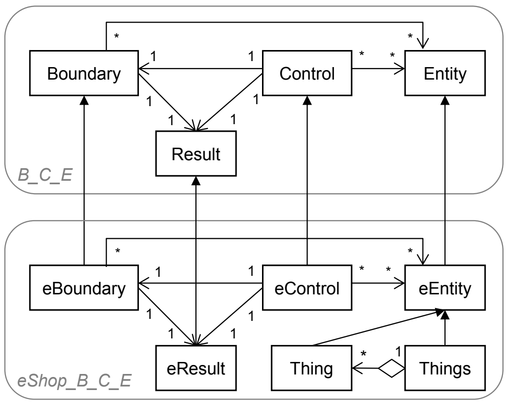

Figure 6. Frameworks B_C_E and eShop_B_C_E

Abstract classes `Control` and `Boundary` communicate by means of the `Result` class. Fig. 6 shows how the `Boundary`, `Control`, `Entity` and `Result` classes are slightly specialized to classes `eBoundary`, `eControl`, `eEntity` and `eResult` especially to support the eShop system, e.g. `eResult` adds the specific actions to be used in eShop. Class `eEntity` is specialized to `Things` and `Thing` where `Things` is an aggregation of `Thing`.

```
public class Products extends Things {
  …
}
abstract public class Product extends Thing {
  public Product (int i) {…}
   public String infoToString() {
    return (id+" "+getClass().getSimpleName());
  }
}
public class T_shirt extends Product {
  public T_shirt(int i, String s, String c) {…}
  public String infoToString() {
    String ss = super.infoToString();
    return(ss+" "+size+" "+color);
  }
  …
  public SIZES size;
  public COLORS color;
}
```

Figure 7: Entity Classes: `Products`, `Product` and `T_shirt`

Fig. 7 shows the classes `Products` and `Product` as specializations of `Things` and `Thing`, respectively. Class `T_shirt` extends class `Product` where `T_shirt` extends method `infoToString()` and supports size and color. Only a very simple graphical user interface is included.

V. COURSE CONTENTS AND EXPECTATIONS

A. Course

The course includes presentations of a number of subjects to support the intended learning outcome and knowledge areas of Section 2. The presentations of these subjects are introductory, i.e. mainly the basic characteristics of the concepts, techniques, descriptions, languages etc. are included.

B. Project

The project exposes the didactical design principles of Section 3 and includes two phases:

- Design and implementation
- Experiments and evaluation

The deliverables from the project are described below (no credit is given neither for addition of database functionality nor for improvements to the graphical user interface).

Design and implementation (addition including revision of functionality of the existing system) include (no order is required for the conduction and description):

- Include additional products *book* (with author and ISBN) and *wine* (with name and vintage).
- Modify the conceptual model to include relevant concepts in relation to these additions.
- Include additional functionality, i.e. *View Basket & Checkout*, in order for the customer to use a shopping basket and to eventually order the products in the basket.
- Make a (fully dressed) use case description for *View Basket & Checkout*.
- Make the additional class diagram in relation to *View Basket & Checkout*.
- Make the sequence diagram for *View Basket & Checkout*.
- Complete the program to support *View Basket & Checkout* as well as the products *book* and *wine*. The program must translate and execute appropriately.

Experiments and evaluation (additional revisions to and discussions of the revised system) include:

- Describe two snapshots (including alternatives, evaluation of these and selected solution) for each of the above parts of *View Basket & Checkout* in order to illustrate the stepwise improvement in the process.
- Describe, compare and evaluate to which extent the addition of *View Basket & Checkout* comply with the notion of design patterns in general and the existing B-C-E application framework.
- Discuss how the Creator pattern [8] is used to support `Entity` or subclasses of `Entity`.
- Discuss, but neither design nor implement, the implications the additional non-functional requirement to avoid a scenario where customers simultaneously add products to their baskets may receive an “out of stock” message when checking out.
- Discuss, but neither design nor implement, potential consequences of and problems with an additional requirement related to *View Basket & Checkout* to avoid a single customer to add too many items in the basket for too long time.

C. Project Expectations

There is no single solution to the tasks of the project. Consequently the following observations and comments regard problems and aspects of various solutions especially from a modeling point of view:

- Use case *View Basket & Checkout* consists of two parts namely an inspection of (and probably adjustments to) the contents of the basket possibly followed by a conclusion of the shopping in order to buy the items in the basket. Alternatively after inspection of the basket the customer may return to the shopping, i.e. browsing the items of eShop. *View Basket & Checkout* is a single use case or may be split it into two use cases *View Basket* and *Checkout* depending on modeling, complexity and the scheduling between the use cases.
- A `Product` object represents an actual item in the eShop, i.e. *n* T-shirts are represented by *n* objects. The product identification `pid` identifies the actual item not the product type. Because the description of the product type is identical for identical item each `Product` could refer to the same *product type* object. Consequently the collection of all items currently on *stock* is a collection of references to these `Product` objects. And similarly *basket* could be the collection of items currently in `Basket`, i.e. a collection of references to these objects. When an item is included in *basket* it is removed from *stock*. Alternatively `Product` actually represents the product type and `pid` the identification for this type. `Product` may then also contain the price and the number of items of this type on *stock* and in *basket*.
- Stock is a `Products` object. Basket could be a `Basket` aggregated by a `Products` object, as well as the total price of the `Products` in `Products`. When *n* items of a `Product` are put into the `Basket` the number of items of this `Product` object is reduced by *n* and a new `Product` object is created accordingly where the number of items is *n*. It is relevant to consider what happens to the actual contents of the basket in the case where *n* items of a product is added to the basket and then later *m* items of the same product is added.

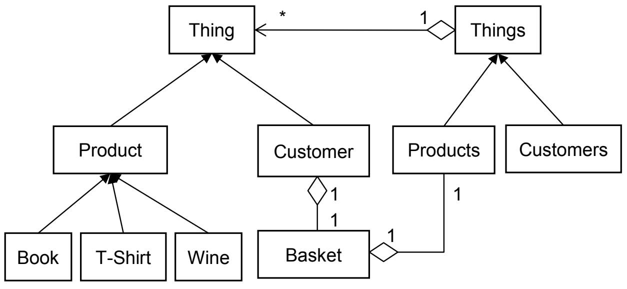

Figure 8. Conceptual model/Class diagram: Subclasses of eEntity

Fig. 8 summarizes a design of subclasses of `eEntity`. `Book` and `Wine` are modeled as specializations of `Product` similar to `T_Shirt`. `Customers` and `Products` are specializations of `Things` and `Customer` and `Product` are specializations of `Thing`. `Customer` is aggregated by `Basket` that is aggregated by `Products`.

VI. Evaluation & Reflection

A. Evaluations

The learning outcomes and knowledge areas are necessary restrictions to ensure the right—although limited—focus. For example no testing—and as mentioned no GUI—are included in the course. By intention the introduction to the eShop system follows the steps 1) requirements, 2) design and 3) program in order to structure in the presentation. Still the software development process as such is not included in the course.

The didactic design principles form a conscious choice: The principles intend to control and support the student to explore and experience the right problems and solutions in the right way by using appropriate effort and time. Additional realism dilemmas—professionally relevant knowledge areas versus teachable learning activities—also include graphical user interface [11], database system [12], testing [13] and software development methodology [14]. In the eShop case the user interface design is real dilemma: Graphical user interface design and implementation as a topic is left out and only premature GUIs are available with awkward windows, buttons etc. The consequence is that the software system appears as nonrealistic and primitive. Alternatively the course could contain a limited but realistic introduction to GUI's for example concentrated around on the 8 golden rules of interface design [11] and the project could include the design of a few GUI's in recognition of the high expectations of today to user interfaces to almost any software system. The typical effect of this alternative choice is that the GUI aspect would more or less take possession of the project and remove the focus from the modeling aspect. In contrast the topics database system, testing and software development methodology are potential, but not real dilemmas: The lack of these aspects is tolerable because the course still can be conducted without disturbance, dissatisfaction or lack of realism.

The system appears to be very simple when presented although the implementation is rather complex and profound. Still the system is very illustrative and supportive for the modeling experience. The solution illustrated is only one solution, maybe not the right solution and certainly not the only solution. The purpose of the solution included is merely to illustrate the kind of experience that is expected for the student.

B. Reflections

In Fig. 9 we illustrate the domains and models from an abstract generalized model of the software development process (adapted from [15], [16]):

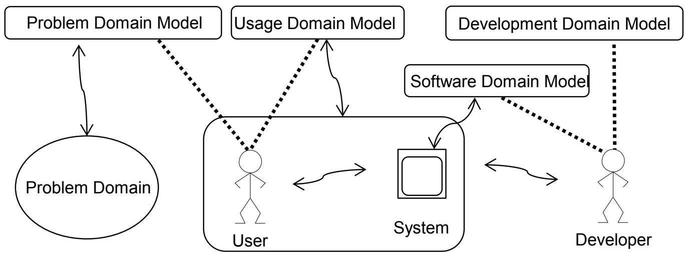

Figure 9. Domains and Models

The domains are all perspectives on real or mental phenomena, i.e. we select, envision, and/or identify certain phenomena and classify them as belonging to one or more domains:

- The *Problem Domain* contains the phenomena that the user wants to use the system to administrate, control, monitor or manipulate. It must reflect the understanding of the future user of the system.
- The *Usage Domain* contains the phenomena that are in involved in the user's interaction with the system: work processes, different ways of interacting, user interfaces, etc. This must also reflect the understanding of the future user of the system.
- The *Software Domain* contains the phenomena that constitute the system, from a software developer's point-of-view. These are typically reflected in the tools and methods applied by the developer.
- The *Development Domain* contains the phenomena that constitute the working environment of the software developer. In this sense the domain contains the three other domains as true subsets.

When developing software, we use, modify and create models of phenomena from the various domains in the process. Some models are only implicit and mental, but many models are also made explicit and manifest.

In the course the explicit models are exemplified with:

- *Problem Domain Models:* e.g. the conceptual model in Fig. 3 and Fig. 8.
- *Usage Domain Models:* e.g. the use-case diagram in Fig. 2. Other models include interaction design models and user-interface mock-ups.
- *Software Domain Models:* a large range of design models falls within this category, e.g. the concept/class diagrams in Figures 4, 6, 7 and 8, and the sequence diagram in Fig. 5.
- *Development Domain Models:* these models are typically not graphically depicted, but examples include process models, such as stepwise improvement.

The domains and models illustrate the vision during the system development process — as such they reflect the understanding according to the current iteration. The focus point in the illustration is the software to be developed.

The development domain is special in the sense that it contains the three remaining domains. Hence when developing, we apply stepwise improvement to all of the corresponding models: we evolve them from abstract to concrete, from partial to complete, and from unstructured to structured. This reflects our improved understanding of the related domains, and (as a special case for the software domain) the increased degree of completeness of the desired system (in the sense: working and tested software). Thus, development of a model can be characterised as a mixed sequence of refinements, extensions, and restructurings of the model. A program is simply a special case of model, and as such programming is a special type of modelling.

In essence, Fig. 9 thus captures a snapshot of a software development process based on stepwise improvement of the various models.

VII. Summary & Conclusion

In Section 1 we introduced the overall learning goal of our proposed course: understanding that "*to program is to model*". This was then fleshed out into more specific learning goals related to programming, modeling, design and the software development process described in Section 2, as well as the didactical principles described in Section 3 related to knowledge areas, learning activities and professional dilemmas. Section 4 described the system, which the course is based on, and Section 5 described the actual (project-based) course contents. In Section 6 we reflected on the relationship between the various elements of the course, and we reflected on the nature of the relationships between models, domains and software development.

In summary, we conclude that when training to become a software professional, we find it important, that students are able to:

- Apply models and programs.
- Change models and programs, e.g.
  - Make a model more concrete or more abstract.
  - Extend a model or remove elements from a model.
  - Restructure/refactor a model in order to change its non-functional qualities.
- Create new models and programs.

All of these skills are applied in the development domain, and they all deal with the problem domain, the usage domain and/or the software domain.

The relevant and necessary intended learning outcomes of section 2 are exposed and worshipped by the project. The didactical principles of section 3 are supportive and necessary in order to be able to teach the course efficiently and productively. The didactical design principles also express a general approach to teaching. The actual software system fits with the intended learning outcomes and is designed to comply with and support the didactical design principles. Finally the actual course contents and especially the project organization and tasks fit to and underline the didactical design principles. Together intended outcomes, didactical principles, partial software system and project forms a coherent, useful and simple basis for communicate the conviction that programming actually is modeling.

References

[1] H. C. Andersen. Mit Livs Eventyr. 1855.

[2] K. Nygaard. Private communication, 1990.

[3] M. E. Caspersen, P. Nowack (2013): Computational Thinking and Practice — A Generic Approach to Computing in Danish High Schools. *Proc. of the 15th Australasian Computing Education Conference*, Adelaide, Australia, 15:137-143.

[4] H. B. Christensen, M. E. Caspersen (2002): Frameworks in CS1: a Different Way of Introducing Event-Driven Programming. *Proc. of the Conference on Innovation and Technology in Computer Science Education*. Aarhus, Denmark, **7**:75-79.

[5] M. E. Caspersen (2007): *Educating Novices in the Skills of Programming*, DAIMI PhD Dissertation PD-07-04, ISSN 1602-0448 (paper), 1602-0456 (online).

[6] M. E. Caspersen, M, Kölling (2009): STREAM: A First Programming Process, *ACM Transactions on Computing Education*, **9**(1):4.1-4.29.

[7] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 2005.

[8] C. Larman. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, (3rd Edition), Prentice Hall, 2004.

[9] M. E. Fayad, R. E. Johnson, D. C. Schmidt. Building Application Frameworks: Object-Oriented Foundations of Framework Design. Wiley, 1990.

[10] K. Arnold, J. Gosling. The JAVA Programming Language. Addison Wesley, 1999.

[11] Shneiderman, B.: Designing the User Interface: Strategies for Effective Human-Computer Interaction. Addison-Wesley, 1992.

[12] Ullman, J.: *First Course in Database Systems*. Prentice-Hall, 1997.

[13] Meyer, B.: *Seven Principles of Software Testing*. Computer, vol. 41, no. 8, pp. 99–101, 2008.

[14] Kroll, P., Kruchten, P.: Rational Unified Process Made Easy - A Practitioner Guide. Addison-Wesley, 2003.

[15] E. E. Jacobsen, B. B. Kristensen, P. Nowack. Architecture = Abstractions over Software. *Proceedings of International Conference on Technology of Object-Oriented Languages and Systems (TOOLS PACIFIC 99)*, Melbourne, Australia, 1999.

[16] E. E. Jacobsen, B. B. Kristensen, P. Nowack. Models, Domains and Abstraction in Software Development. *Proceedings of International Conference on Technology of Object-Oriented Languages and Systems (TOOLS ASIA 98)*, Beijing, China, 1998.

[17] J. Bennedsen, M.E. Caspersen. Teaching Object-Oriented Programming — Towards Teaching a Systematic Programming Process, *Proceedings of the Eighth Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts*, 18th European Conference on Object-Oriented Programming (ECOOP 2004), Oslo, Norway, 2004.

[18] J.B. Bennedsen, M.E. Caspersen. Model-Driven Programming, *Reflections on the Teaching of Programming*, LNCS 4821, Springer-Verlag, 2008, pp. 116-129.

[19] J. Börstler, M.E. Caspersen, M. Nordström. *Beauty and the Beast — Toward a Measurement Framework for Example Program Quality*, Technical Report, Department of Computing Science, Umeå University, 2007. ISSN 0348-0542.

[20] J. Börstler, H.B. Christensen, J. Bennedsen, M. Nordström, L.K. Westin, J.E. Moström, M.E. Caspersen. Evaluating OO Example Programs for CS1, *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE 2008, Madrid, Spain, 2008, pp. 47-52.