

Separation in Theory – Coordination in Practice

Torkil Clemmensen^a, Jacob Nørbjerg^b

*Department of Informatics, Copenhagen Business School, Howitzvej 60, DK-2000, Frederiksberg,
Denmark*

tc.inf@cbs.dk^a, jan.inf@cbs.dk^b

Abstract

The lack of a common language and mutual understanding between the disciplines of systems development/software engineering and HCI does create challenges for both teaching and practice in systems and software development. In this paper we propose, however, not to attempt to ‘bridge the gap’ between the disciplines when teaching information systems development. Using our own teaching as an example, we argue that it would require a considerable effort to reconcile the differences in terminology and approaches to systems development between the two disciplines and that the reasonable way forward is to teach the students how to coordinate in practice, while keeping the theories separate. Coordination of HCI and SE curricula can take place by using a vehicle of tacit knowledge, the software prototype, which enables students to learn through practical experience the richness and usefulness of both approaches. The implications of this position are discussed.

1. Introduction

The authors of this paper teach systems development methods and HCI at a university program in Informatics. With a background in psychology and human-computer interaction, respectively computer science and software engineering, we’ve come to see HCI and software engineering as two widely separate ‘bodies of knowledge’, both occupied with the design and implementation of useful information systems, but with very separate world views, research traditions and theoretical foundations. Building shared concepts and languages, and integrating the two fields in a joint curriculum, is therefore a considerable undertaking, which may furthermore reduce the students’ appreciation of the richness and diversity of each topic. Our position is, therefore, that as far as teaching is concerned, the two disciplines should be coordinated rather than integrated by being presented and discussed separately but

concurrently, with constant consideration of and attention to their practical interactions and dependencies, but as disciplines in their own right, each with their own vocabulary, theoretical foundations, and research traditions. The dependencies and interactions between the two fields can be demonstrated and appreciated through practical assignments where the students apply methods and techniques from HCI and software engineering in small projects building prototypes of a system. In this way, they demonstrate that they know how to apply a systems development methodology as well as analyze users’ interactions with a system.

Our position builds on our experience from jointly planning and teaching a course in information systems development where we try to apply the above principles. The course is part of the 2nd year of a bachelor program in business administration and computer science which – on a larger scale – applies similar ideas to the whole program.

The significance of our contribution is that we state a general principle for coordinating HCI and SE curricula: coordination can take place by using a vehicle of tacit knowledge, the software prototype. Furthermore, choosing the prototype as a point of coordination – instead of trying to integrate everything in e.g. a new shared language or common method – allows students to learn an important medium for practitioners: prototype design, which computer scientists, usability specialists and buyers and users all can relate to.

The rest of the paper is organized as follows: section 2 outlines the bachelor program in business administration and computer science. Section 3 describes the HCI and systems development curricula of the 2nd year course in informatics and argues why we want to teach these as different subjects. In section 4 we present and discuss how the two fields can be systematically combined in practical assignments. Section 5 concludes the paper.

2. The bachelor program in business administration and computer science

The bachelor program in Business administration and computer science at the Copenhagen Business School

(CBS) is a three year, higher education study program aiming to provide students with the necessary qualifications for participation in functions concerning formulation of requirements, development, implementation, operation and sale of IT-based systems on the basis of knowledge of relevant business and societal conditions. Each year 200 students enroll in the program. After the three years, the bachelors will typically be employed as “bridge builders” between users of IT systems within companies and those persons who deliver IT systems (hardware and software).

The program has a fixed structure and all courses are mandatory, though there is no compulsory attendance. The teaching consists of lectures based on textbooks and exercises supervised by student instructors. The exams combine comprehensive written assignments (projects) and oral exams based on the curriculum. Thus, the study program provides a basic theoretical knowledge within the disciplines taught, but is also very problem-oriented and give priority to the application of methods and theories in cases and practical project work, including learning how to manage IT projects.

The structure of the 3-year study program builds on three parallel lines of courses, one for each of the disciplines: 1) Organization: management and cooperation, conflict resolution, etc. 2) Economy: national and business economy, cost estimation related to information systems development, etc. and 3) Informatics: programming and database design, operating systems and networks, and information systems development. The study program ends with a 6 month group project, in which the students integrate the three disciplines taught in the program and applies what has been learned on a real problem in cooperation with an external organization.

This paper focuses on the informatics part of the program.

3. The Informatics courses

The first year course in informatics focuses on databases and introduction to programming. Thereafter, the second year introduces complex IT systems through four course modules: operating systems, object oriented system development, man-machine interaction, and distributed systems. During the third year, students take two courses related to large scale systems development: definition of information systems and computer supported systems development.

The authors of this paper are responsible for the systems development module and the interaction design module in the 2nd year course. These modules together take 44 lectures and a similar number of exercise sessions and are followed by a two week assignment where the students apply the methods and techniques from the course to a practical problem. The curriculum focuses on methods and techniques but emphasizes underlying

theories and concepts too. When we in this paper discuss HCI and SE curricula, it is therefore these kinds of curricula that we talk about. Below we briefly describe the two course modules followed by a discussion of the differences in terminology usage and perspectives between systems development and interaction design that we’ve experienced, and how these affect the way we plan and conduct the course.

3.1 Object Oriented systems development module

The object oriented systems development module takes up 32 lectures and a similar number of exercises. The module’s core curriculum consists of a text-book in an object oriented analysis and design (OOA/D) methodology [9]. The methodology uses the UML notation and applies many of the same ideas as the Unified Process (see e.g. [6]) but with higher emphasis on the business modeling step. In addition to the methodology, the module introduces students to other SE topics such as process modeling, quality assurance and configuration management.

The OOA/D methodology divides analysis and design into five main steps: definition of system scope and purpose, development of a model of the system’s problem domain, analysis and definition of functionality and interface requirements, architectural design, and object design. The methodology emphasizes a layered architecture and the use of analysis and design patterns.

The textbook focuses on clear and concise descriptions (diagrams, tables, structured descriptions) to document work and support subsequent work; i.e. programming and maintenance. It explicitly recognizes that some of the activities in the methodology – particularly definition of system scope and purpose, and determination of interaction requirements and functionality – depend on skills and techniques not covered by the methodology; e.g. organizational analysis, work process analysis, and interaction design. These skills and techniques are briefly mentioned and some short examples of their use are given but in general they are treated as belonging to (important) activities ‘outside’ the methodology, which can be used to produce input to the analysis and design process; e.g. in the form of use cases and interface specifications.

The teaching and the evaluation of the students’ work emphasize consistent application of the method’s principles and development of analysis and design models of high technical quality.

3.2 Interaction design module

The students follow the Interaction Design module in parallel with the systems development module,. The Interaction Design module spans 12 lectures and a similar number of exercise sessions. The curriculum consists

mainly of a textbook in Interaction Design [11]. The rationale for using this book is that our everyday interaction with systems and products require more than a flawless technical design. It is also necessary to design products that are easy, effective and fun to use. Hence the purpose of the module is to allow the students to familiarize themselves with theories and methods for user-oriented analysis, design and evaluation of IT interfaces.

The module combines lectures that introduce concepts, models and principles of interaction design with exercises where the students apply these principles. For example, a lecture may introduce the students to the principle that beneath every physical design lies a conceptual model and in the exercises the students will study calendar systems and their underlying conceptual models. Thus, the module is not a course in the construction of graphical interfaces, but an introduction to user-oriented design. The aim is that the students:

- acquire a terminology to reason about a product's user-friendliness,
- understand how conceptual models influence the design and use of products,
- can apply different techniques to identify users' needs and describe their requirements,
- can demonstrate hands-on experience with the use of prototypes to involve users in the design process,
- have insight in strengths and weaknesses in usability test methods.

The module is the students' first encounter with HCI and it is therefore of utmost importance that they become able to see the usefulness and richness of this approach. They can later (at the graduate level) engage in more deep level studies within the discipline.

3.3 Differences in concepts and approaches

During our discussions on how to organize and run the second year informatics course we have come to realize how difficult it will be to integrate the disciplines in a coherent manner which at the same time respects both areas' research traditions, theoretical underpinnings, and perspectives on systems development and humans. We illustrate this point with examples of the issues and problems we've encountered followed by a brief reflection on the two fields' position in the larger landscape of IS and SE research and practice.

Our first example illustrates a terminological and conceptual confusion caused by the different meaning and use of the concepts *problem domain* and *problem space*. Within systems development 'problem' or 'business' domain – in some instances also denoted application domain – is a common term to denote the area of concern of an information system; e.g. accounting, information exchange, or process control. A central activity in systems

development methods is the construction of a suitable model of the problem domain to be included in the final computerized system. Differences between methodologies concern the nature of the model and the modeling steps as well as the final representation and use of the model in the computerized system; e.g. as data flow diagrams and data models in classical structured analysis, and use cases and object models in the Unified Process.

The OOA/D textbook used in the course defines problem domain as 'that part of a [system's] context that is administered, monitored or controlled by a system' [9, p. 6]; i.e. accounts, transactions, invoices etc. in an accounting system, or planes, radars, flight paths etc. in a flight control system. The method stipulates that the information system shall model the state and behavior of the problem domain. Thus, a pivotal activity in the methodology concerns modeling the problem domain as objects, classes and structures and identification of the (real world) events and event sequences that influence the state of individual objects.

The Interaction Design textbook uses a similar concept – *problem space* – in a very different manner. "Problem space" is the analyst's "space", i.e. a subjective space, and its elaboration requires the analyst to reflect upon his or her perceptions of the world, and the idiosyncrasies, and social relations of the human (future) user of an information system. In order to avoid beginning design at the physical level, i.e. beginning with decisions on whether to use menus, commands, icons, gestures etc., the interaction designer should think through how his or her design will support people in their everyday work activities [11]. This means defining the usability and the user experience goals by thinking through design ideas and explicating assumptions and claims. These may relate to design ideas that need to be better formulated or user needs that should be determined by identification of problematic human activities or potential useful applications of new technology. In this explication process, frameworks for looking at the system from the users' point of view may be useful. A central output from the process of defining the problem space is conceptual models, i.e. models that describe "the proposed system in terms of a set of integrated ideas and concepts about what it should do, behave and look like, that will be understandable by the users in the manner intended" [11, p. 40]. Thus, working out the *problem space* in Interaction Design is a different process from that of modeling the *problem domain* in OOA/D.

Our second example illustrates the danger inherent in attempting to share a new language between SE and HCI. In 1992, [7] introduced *use cases* into the object oriented community and since then this approach has spread to the HCI community. Today, a use case in HCI is a recognized form of task description, which focuses on the user-system interactions [11]. The interactions of interest are those necessary to achieve the goal most users want in the

situation. Thus, the main use case describes what is called the ‘normal course’ (alternative courses are also considered) through a use case, i.e. the set of actions that the analyst believe to be most commonly performed. For example, if data gathering has led the analyst to believe that most users of a library go to the catalog to check the location of a book before going to the shelves, then the ‘normal course’ for the use case would include this sequence of events [11, p. 227]. In this way, use cases in HCI is useful in the conceptual design stage, and less so during requirements or data gathering [11, p. 228].

Use cases in HCI are thus a possible way to express insights gained from in-depth enquiry into a use situation whereas OOA/D apply them as means to *determine* the application domain [9, p. 120]. This is done - not by deep analysis of work tasks in the current physical system - but by following a ‘blitzing’ strategy of making a brief survey and then jump directly to a specification of functional requirements. Thus, the first step in defining the target systems usage is to identify actors by determining their different roles, the second step is to describe the actors by stating their goals and characteristics. The final step is examination of the application-domain tasks in sufficient detail to distinguish between different use cases that describe the interactions between the target system and the actors. Thus, OOA/D models the application domain by structuring actors and use cases. And though the wisdom in following a ‘blitzing’ strategy vs. doing a deep work analysis may be hard to judge, the point is that HCI and SE enter the matter of use case descriptions from opposite directions and therefore have opposite views and backgrounds. Use cases seems to function as figure-ground pictures; what is the ground for HCI in a use case, is the figure for SE, and inversely.

These, and other examples of terminological differences and inherently different perspectives on shared concepts, indicate that it will require a considerable effort to develop shared curricula and teaching approaches for the two disciplines. However, the example also points to deeper differences in the basic assumptions and underlying world views which are even more difficult to overcome than differences in terminology.

The problem domain analysis step in OOA/D requires the analyst to abstract out relevant (with respect to the system’s purpose) features of the ‘real world’ in order to construct a model which – after some further transformations – will be included in a computerized system. Objects representing human beings; e.g. customers, pilots, or operators may be included in the model, but they are seen as no different from objects that represent machines, tools or abstract entities. Aspects of the real world not suitable for modeling purposes, such as emotions, conflicts, or social relations are (deliberately) ignored as are artifacts or concepts that the analyst deems outside the scope of the system under consideration.

On the other hand, the main idea in the concept of “problem space”, within interaction design is that the analyst works out and explicates his or her *implicit assumptions about the use of the design*, preferably by using an explicit theoretical framework and associated techniques. Plurality of perspectives and representation are encouraged when working out a problem space and the purpose of the resulting descriptions is not to build an unambiguous representation of a part of the world, but to inspire reflection and generate ideas about the system’s purpose, scope and relation to human work.

The differences outlined above relate to the two disciplines’ foundation in widely different research traditions. OOA/D is rooted in software engineering and hence in a tradition that focuses on the *construction* of computer systems. Focus is therefore on the production of unambiguous abstract descriptions of the world that are suitable for representation in a computer program or that can be used to define the interaction between the system and its surroundings. Human beings are relevant insofar that we need to produce interfaces that allow for effective and efficient interaction with them. In this respect OOA/D shares the perspective and approach of the majority of systems development methods, and like these belong in the functionalist systems development paradigm as discussed by [5] (see also [1-3] for an elaboration and discussion of this point.)

Interaction design is, on the other hand, rooted in a different research tradition that emphasizes *understanding* of the social world rather than the construction of technical systems. The purpose of the study of humans and their (possible) interactions with a technical system is therefore to understand human work and – eventually – to improve it with or without a computer system. Plurality of perspectives and inclusion of a wide range of aspects of human and organizational behavior and interaction; e.g. emotions, social relations, (political) conflicts, are accepted and encouraged. HCI – as taught by us – is therefore rooted in the social-relativist paradigm as discussed in [5].

3.3 Integration or coordination?

These examples indicate that attempts to integrate the two disciplines in one course curriculum face several difficulties and challenges. Not only will it require a considerable effort to reconcile the differences in terminology and approaches to systems development, but there is also a risk that a joint or shared curriculum becomes disengaged from the underlying theories and perspectives of one or both of the disciplines. One could – as an example – resolve the ambiguity of problem domain/problem space by inventing new concepts or terminology. This would enable us to discuss “problem space” and techniques for analyzing and describing it within a coherent curriculum, but it would cut the students

off from HCI research and theories. We would further risk reducing the teaching of Interaction Design to a set of ready-to-use techniques with no appreciation of the paradigmatic differences between software engineering and interaction design and of the implications of these differences. A similar point has been raised by [10].

Practical software development on the other hand also requires a careful balancing of the two areas to avoid the danger of one approach becoming dominant. Dittrich and Lindeberg has observed how software developers engaged in the study of a user organization risk ‘going native’. Like ethnographers risk being immersed in the culture of the natives, software developers can become too immersed in the culture of the users if they adopt a use oriented approach without at some point stepping back and reflecting upon their assumptions [4]. Thus ‘working out the problem space’ is not an innocent phrase in the Interaction Design vocabulary; it requires considerable insight into and experience with applying frameworks for understanding users.

Instead we propose to teach the two disciplines as different but equally important subjects. This will allow teachers to present and discuss concepts, ideas and techniques within the framework of the research tradition that produced them. Such an approach furthermore opens up for critical reflection and discussion of the differences and similarities between the fields; e.g. the reasons for working with different – equally valid and useful – ambiguous descriptions of “problem spaces”, vs. the production of an abstract and precise “problem domain model”.

The challenge facing the teacher (researcher, practitioner) is of course how to combine the two disciplines in practical systems development. We describe how we deal with this problem below.

4. The assignments – using the prototype as a means of coordination

We propose to overcome the above mentioned challenge in two ways: First, by constantly emphasizing the relations between the two areas when teaching. When teaching OOA/D, for example, we emphasize that the use cases produced during interaction design and function specification are the visible end-result of a complex analysis of human users’ work processes and interaction requirements. Conversely, when teaching the interaction design module we emphasize how some of the artifacts produced when studying users and experimenting with various interface designs relate to the more ‘formal’ modeling activities in OOA/D.

Second, we use the final two-week project assignment as a means to demonstrate how to combine interaction design and OOA/D in practical project. To do so, we ask

the students to develop two different prototypes of the same software system:

A *horizontal* prototype that demonstrates the user interface of a complete system. The prototype should be the end-result of systematically applying theories, tools and techniques taught in the interaction design module and it should conform to accepted user interface design principles. This prototype and the associated documentation documents the students’ ability to use the material taught in the interaction design module and thus proves that they can do the analysis necessary to get a firm grasp of the interaction between the user and the design.

A *vertical* (running) prototype that implements a small subset of the (intended) final system’s functionality. The prototype – and its associated analysis and design documentation – should be developed according to the methodology and conform to general software engineering analysis, design and programming principles. Thus, the vertical prototype demonstrates the students’ ability to systematically apply the OOA/D methodology and produce a running program that is consistent with their analysis and design.

As indicated, the above prototype(s) allow the students to apply the theories and techniques from each of the modules taught within the framework of an assignment of a reasonable size. By deliberately letting the students work in a ‘grey zone’ between the two clearly different approaches, we enable them– through their own practical experience - to realize how the fields of interaction design and software engineering together contribute to the construction of a system. Thus, they can either choose to make two independent prototypes and describe in words how they are related, or they can choose to develop one prototype that combines an implementation of a subset of the system’s functionality with ‘dummy’ design of the user’s interaction with a full system. Either way, they need to reflect on how to coordinate the two approaches.

For example, deliberations over the ‘problem space’ and experiments with different user interfaces inform the scoping of the system and definition of functional requirements. It can also help the students become aware of different aspects of the ‘problem domain’ – thus serving to inspire and validate the ‘problem domain’ analysis and subsequent design activities. On the other hand, the problem domain analysis and design activities inspire and lead to the design of new user interface prototypes, which can help surface implicit assumptions about the use of the system, and thus expand the students’ understanding of the ‘problem space’ of the design. Ideally, such iterations should provide the students with deeper insights into the purpose of the design of a system.

From a project management point of view, this approach allows the students to learn how to coordinate interaction design activities with other analysis and design activities.

5. Discussion and conclusion

The students' iteration between human-computer interaction and software engineering methods cannot be taken for granted; obviously the design problem can be solved by choosing some linear approach. Students may choose to do all the OOA/D first, then the Interaction Design part. In such cases, reflections on how to coordinate the two approaches will be retrospective and unnecessary. However, learning such a linear approach to design is not the goal with the course; therefore we will advise the students to use the two approaches in parallel.

To instruct the students in how to coordinate the use of interaction design and OOA/D methods requires that we can point to obvious points of coordination. To identify these points is an issue for further discussions, but we have a few ideas for our course: First we may ask the students to assign, among themselves and within their project group, advocates for 'problem space' and 'problem domain', and thus have a continuous discussion of these two perspectives of the analysis and design. Second, we may explicitly instruct the students to reflect upon 'use cases' as a concrete meeting point between the disciplines but also as having slightly different connotations and purposes. This approach resembles that of [8] in that we recommend building different kinds of knowledge and insights during a development project by applying different perspectives and sets of tools.

We have argued that coordination between SE and HCI can take place by using a vehicle of tacit knowledge, the software prototype, which also provides students with the opportunity to pay equal attention to interaction design and systems analysis and design in their work. Furthermore using the prototype approach to learn to coordinate will help make implicit assumptions behind the HCI and SE methods more explicit, e.g. assumptions of the power of modeling tools or the precision of usability design principles.

Finally, we have argued that choosing the prototype as a point of coordination – instead of trying to integrate everything in e.g. a new shared language or common method – allows students to learn an important medium for practitioners: prototype design, which computer scientists, usability specialists and buyers and users all can relate to. In this way, the students already during their basic education learn (on a small scale) to coordinate different approaches to design. Hence they are not left to acquire this competence on their own, when they, after becoming bachelors, stand face to face with the 'reality shock' in their first years as practicing designers.

6. References

- [1] Bansler, J., *Systems Development in Scandinavia: three theoretical schools*, *Office Technology and People*, 1989, 4, pp. 117-133.
- [2] Bansler, J.P. and K. Bødker, *A Reappraisal of Structured Analysis: Design in an Organizational Context*, *ACM Transactions on Information Systems*, 1993, 11 (2), pp. 165-193.
- [3] Dahlbom, B. and L. Matthiassen, *Computers in Context. The Philosophy and Practice of Systems Design*, 1993, Malden, MA, Blackwell.
- [4] Dittrich, Y. and O. Lindeberg, *Can Software Development be too Use Oriented? Going Native as an issue in Participatory Design*, in *IRIS*, 2001, Bergen, Norway.
- [5] Hirschheim, R. and H. Klein, *Four Paradigms of Information Systems Development*, *CACM*, 1989, 32 (10), pp. 1199-1216.
- [6] Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Object Technology Series, ed. I. Jacobson, G. Booch, and J. Rumbaugh, 1999, Boston, Addison-Wesley.
- [7] Jacobson, I., et al., *Object Oriented Software Engineering: A Use-Case-Driven Approach*, 1992, Reading, MA, Addison-Wesley.
- [8] Kensing, F. and A. Munk-Madsen, *Structure in the Toolbox*, *Communications of the ACM*, 1993, 36(4), pp. 78-83.
- [9] Matthiassen, L., et al., *Object Oriented Analysis and Design*, 2000, Aalborg, Marko.
- [10] Nyce, J.M. and G. Bader, *On Foundational Categories in Software Development*, in *Social Thinking - Software Practice*, R. Klischewski, C. Floyd, and Y. Dittrich, Editors, 2002, The MIT Press, Cambridge, MA, pp. 29-44.
- [11] Preece, J., Y. Rogers, and H. Sharp, *Interaction Design: Beyond Human-Computer Interaction*, 2002, John Wiley & Sons.