iUCP – Estimating Interaction Design Projects with Enhanced Use Case Points

Nuno Jardim Nunes

University of Madeira, Lab:USE, Campus da Penteada,
9000-390 Funchal, Portugal
njn@uma.pt

Abstract. This paper describes an approach to adapt the use-case point estimation method to fit the requirements of agile development of interactive software. Creating product cost estimates early in the development lifecycle is a challenge for the software industry, they require substantial data from past projects and constant feedback and fine-tuning, which are rarely available or consistent through interactive software development. In addition, the profusion of incremental and evolutionary development methods (like Scrum and XP) produced new challenges with estimating frequent releases. Here we propose several changes to the original use-case point estimation method, in particular to take advantage of the enhanced information that can be extracted from usage-centered design (usageCD) that devotes particular attention to critical aspects like weighting actors and uses-cases for complexity. We propose to exploit user-roles, essential use-cases and the usageCD architecture to enhance the weighting heuristics for assigning complexity factors to actors and use-cases required to calculate the unadjusted use-case point reflecting the complexity of the requirements for a given iteration or evolution. We propose to exploit user-roles as the main basis for weighting complex actors, which originally are grouped in the highest weight factor. Conversely we propose to extract the complexity of use-cases from essential use case steps depicted through user intentions and system responsibilities and also the analysis classes extract from those for the usageCD architecture. Detailing this approach the paper presents a contribution, not only to leverage more accurate early lifecycle software estimation, but also to bridge the gap between SE and HCI enabling cross-fertilization between the two disciplines.

Keywords: Software estimation, use-cases, interaction design, integrating SE and HCI.

1 Introduction

For several years the software engineering (SE) and human-computer interaction (HCI) communities tried to bridge methods and techniques that are successful in either software development or interaction design. The cross-fertilization of both disciplines is hard. Methods and techniques are developed independently and are underused mostly because we lack a common understanding between the two communities. Despite that practitioners are increasingly required to work together in multidisciplinary teams, examples of the lack of communication in both disciplines are evident in the major conferences. For instance at CHI'2008 several workshops, panels and special sessions discussed the impact of agile methods in interaction design and in particular how to integrate user-centered design (UCD) techniques in the software development lifecycle. This is more than 10 years after agile development was coined in the SE field and more than 20 years since the early agile methods emerged in the mid 80s (for instance Scrum [1]). On the opposite direction many UCD methods and techniques that are mature and used successfully in the HCI community are unknown and unrecognized by software developers. Although these techniques tackle precisely the major problems of SE (requirements and user-involvement) they are not recognized as suitable and powerful by software practitioners and are still far from large-scale adoption.

In [2] Seffah and Metzer discussed the obstacles and myths of usability and software engineering. The authors argue that we need to educate software and usability engineers to work together, and tackle communication issues that prevent cross-pollinating of disciplines. For instance the fact that usability is a confusing concept and filling the gap between HCI and SE impacts the organization models. These are all outstanding issues that will require a lot of effort and are mostly based on a restrictive vision that decouples the user-interface from the remaining system and builds a barrier between SD and UI specialists. This results in parallel UCD and SE processes that don't communicate and influence each other preventing the whole to become much more than the mere sum of the parts. Here we discuss how one popular and important SE technique for creating product cost estimates early in the development lifecycle through the popular use-case point method. Early estimates are an important challenge for the SD industry and we argue that bringing an HCI insight to this SE technique is not only beneficial for the accuracy of the estimation, but also a basic way to bridge the gap between HCI and SE early in the lifecycle. In this paper we argue that for interactive system development early estimates based on models of requirements can only be accurate if they reflect the HCI concerns related to users and their interaction with the system.

2 Use Case Points (UCP)

Several estimating models have been proposed in the SE field over the years, notably Function Point Analysis (FPA) and Constructive Cost Model [3]. FPA was pioneered by Albrecht in 1977 and assigns a point to each function in an application further adjusting them for the product environmental factors, like complexity of technical issues, developer skills and risk. The Constructive Cost Model (also known as COCOMO) uses statistical returns to calculate project cost and duration within a given probability. Boehm proposed this model in 1981 to provide a tool for predictably estimate product cost and is still evolving today under the sponsorship of the Center for Systems and Software Engineering at USC. The underlying assumption of both FPA and COCOMO is that statistically significant historical data exists to drive the factoring of the models. However companies find very hard to find a consistent definition of functions and environmental factors across multiple projects and development platforms.

With the advent of object-oriented software engineering use-cases emerged as the dominant technique for structuring requirements. This technique pioneered by Jacobson was further integrated in the Unified Modeling Language (UML) and the commercial Rational Unified Process (RUP) thus becoming a de facto standard for requirements modeling in the SE field. Later Karner, also from Rational, created a software estimation technique that assigns points to use-cases in much the same way that FPA assign points to functions [4]. This technique was named Use-Case Points (UCP) and also integrated in the Rational Unified Process receiving tool support from the tools provided by the company at IBM and from other popular UML tool vendors.

The UCP model became popular due to its relative simplicity and rather high level that makes it a good candidate as a method for early estimation of software size and effort. In the following sub-sections we briefly discuss the UCP model based on the original model presented by Schneider and Winters [5]. We refrain from discussing the notion of actor and use-case at this stage and thus adopt the standard UML definitions [6]:

- Use case – A use case is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system.
- Actor – An actor specifies a role played by a user or any other system that interacts with the subject.

These definitions are the starting point of the model since the main activity of UCP is to estimate the complexity of actors and use-cases. The term estimation is not accidental because building a use-case model under given circumstances requires several assumptions that improve over time according to experience and statistically significant data across projects and organizations.

2.1 Estimating Use Case Points

The use-case point method starts by determining the unadjusted actor weight (UAW). For each actor in the use-case model we attribute a weight factor based on the following heuristics:

- Simple actors (factor 1) – system actors that communicate to the system through an API;
- Average actors (factor 2) – system actors that communicate to the system through a protocol or data store;
- Complex actors (factor 3) – human actors interacting normally through a GUI or other human interface;

The total unadjusted actor weight is thus the weighted sum of all the actors in the use-case model. For example if we have a system with 3 simple actors, 2 average actors and 1 complex actor, the total: UAW = 3*1 + 2*2 + 1*3 = 10. Obviously accurate classification of actors needs to be backup up by feedback from historical data from past projects.

After weighting the actors a similar process is applied to use-cases. For each use-case we determine the weight factor of simple, average and complex. Again the method defines a heuristic based on use-case transactions and implementation information:

- Simple use-cases (factor 5) – simple user interface or processing, touches only one database entity, the success scenario involves 3 or fewer transactions and the implementation 5 or less classes;
- Average use-case (factor 10) – moderate user interface or processing, touches 2 or 3 database entities, the success scenario involves 4 to 7 transactions and the implementation 5 to 10 classes;
- Complex use-cases (factor 15) – complex user interface or processing, touches 3 or more database entities, the success scenario involves more than 7 transactions and the implementation more than 10 classes;

The total unadjusted use-case weight (UUCW) is thus the weighted sum of all the use-cases in the use case model. For example if the system involves 3 simple, 2 average and 1 complex use-cases then the total UUCW = 3*5 + 2*10 + 1*15 = 60. Adding the total for actors and use-cases leads to the unadjusted use-case points (UUCP), in the example provide the total UUCP = UAW + UUCW = 10 + 60 = 70.

The UUCP is further modified to reflect the complexity of the project and experience level of the development. This is accomplished through weighting technical and environmental factors, given by the Technical Complexity Factor (TCF) and the Environment Complexity Factor (ECF). For the TCF we assign a perceived complexity value (between 0 and 5) to a series of 13 technical factors. Conversely the ECF is estimated assigning a complexity value (also between 0 and 5) to 13 environmental (or sometimes called experience factors). It is not our goal here to discuss in detail these factors as we are concentrating on weighting actors and use-cases. For a comprehensive discussion of the UCP method please refer to [4, 5].

The following summarizes the calculations required to calculate the final UCP for a given project:

$$UCP = UUCP \times TCF \times ECF \tag{1}$$

$$UUCP = UUCW + UAW \tag{2}$$

$$TCF = C_1 + C_2 \sum_{i=1}^{13} W_i \times F_i, \text{ where } C_1 \text{ and } C_2 \text{ are constants } (C_1=0{,}6 \text{ and } C_2=0{,}01), \text{ W is the weight attributed to the F perceived complexity factor.} \tag{3}$$

$$ECF = C_1 + C_2 \sum_{i=1}^{13} W_i \times F_i, \text{ where } C_1 \text{ and } C_2 \text{ are constants } (C_1=1{,}4 \text{ and } C_2=-0{,}03), \text{ W is the weight attributed to the F perceived complexity factor.} \tag{4}$$

After estimating the total UCP the total estimated number of hours for the project is determined by multiplying the UCP by a productivity factor (PF), which defines the ratio of development man-hours per use case point. PF is based on past project statistics or by establishing a baseline from the UCP of previously completed projects. A value between 15 and 30 is considered typical depending on the team's overall experience. For example taking our example value of UUCP = UAW + UUCW = 10 + 60 = 70, and considering TCF=1.02 and ECF=1.04 the total UCP=70 * 1.02 * 1.04 =

74.256 and applying a PF of 20 yields to a total estimate of 1 485.12 man.hours for the project.

2.2 Related Work

The previous section describes the calculations required to perform an early estimation based on the UCP method. The calculations themselves do not represent any difficulty; the central problem is defining the elements of the model (actors and use cases) and assigning weights to them. In [6] Dieve provides a comprehensive discussion of several issues related to weighting actors and use-cases, in particular the author argues that to obtain reasonably accurate estimates we need to reflect in the use-case model some aspects of the existing applications and project, including some clarifications of the concept or actor and use case across and within projects. In this section we briefly review several of Dieve's issues and then build on them to the domain of interactive software products and interaction design.

- Use case transactions. As we discussed in the previous section the complexity of use-cases is based on the concept of use-case transaction. Dieve formulates this based on the concept of elementary process, meaning that a use-case transaction is the smallest unit of activity that is meaningful from an actor's point of view. A use-case transaction is self-contained and leaves the business of the application sized in a consistent state. A differentiation is required between a use-case transaction and a use-case scenario, the former can contain more that one scenario and vice-versa;
- Scoping. Another important issue is related to the fact that although use-cases are a requirements modeling technique, and thus should not include design concerns, in order to obtain accurate estimates they should reflect some aspects of the conceptual design of the project. Hence the selection of actors may be dictated by the properties of a project, i.e., business concerns and other forms of additional information about actors have an impact in weighting actor complexity, which is not considered in the initial formulation of the UCP method.
- Zero-Weight. An additional important issue is that some actors and use-cases could be excluded from the estimation model. Actors that don't reflect any significant interaction with the system will not impact the project effort. Use-cases can also be zero-weighted because they don't generate an implementation, i.e., some functionality can be provided programmatically and although conceptually they are useful at the requirements level they don't impact the estimation and should be excluded from the calculations;
- Granularity. Since use-cases structure the requirements the issue of granularity becomes very important because for an accurate estimation we have to ensure some uniform sizing. This is particularly relevant when use-cases include much more than the higher complexity limit heuristic of 8 transactions and 10 implementation classes, i.e., not partitioning use-cases that are significantly higher than these limits ignores their real size;

Many other issues could occur when a use case model is used for estimation. The central issue is that use-case estimation relies on the quality of the underlying use-case model, in particular ensuring a consistent application of the heuristics across and within projects. Despite the controversy about estimation methods, in [8] Carroll describes how a large multi-team software engineering organization estimates project cost accurately and early in the software development lifecycle using UCP and the process of evaluating metrics to ensure the accuracy of the model. Although the above discussion applies to general software system development, many issues are clearly related to user-centered design concepts, like models of users (actors) and of interaction (use-cases and transactions). In the remainder of this paper we explore how HCI techniques can inform and guide the estimation heuristics in a way that is not only consistent with interaction design practice, but also highly relevant to sustain several heuristics across projects.

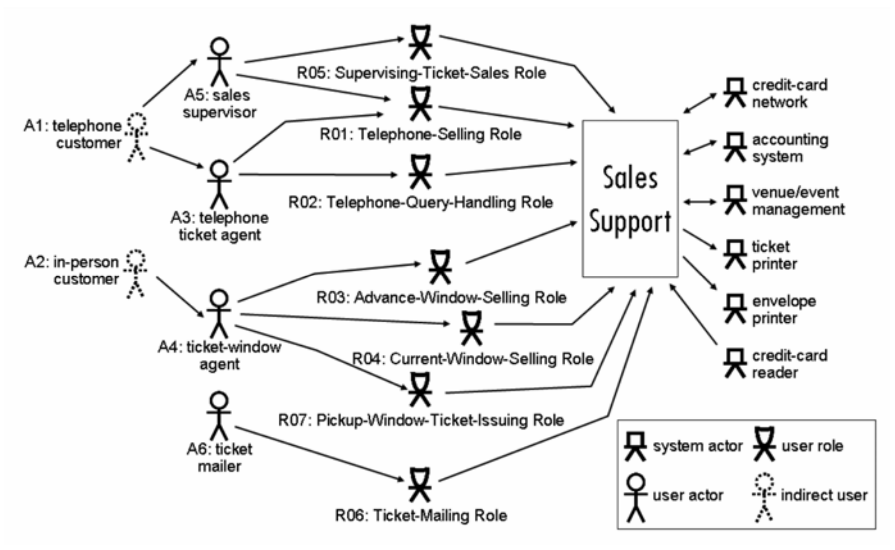
3 iUCP – Estimating Actors and Use-Cases in Interaction Design Projects

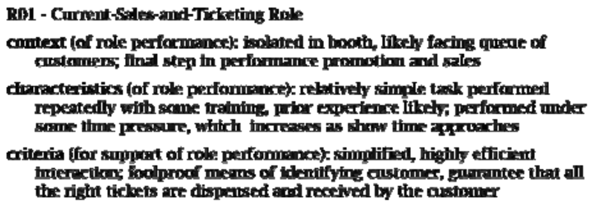
For the purpose of explaining how interaction design can influence estimation of use-case points we consider the model-based techniques pioneered by Larry Constantine [8] and further expanded by colleagues [10, 11] at the Laboratory for Usage-centered Software Engineering (Lab:USE) at the University of Madeira. Although many other techniques are used in design practice (for instance [12]), they usually don't leverage techniques that are popular in the software engineering field, in particular use-cases and user-roles. Constantine's approach is currently named activity-based design in an evolution of the original usage-centered design [9]. The distinction between usage-centered and user-centered design is a matter of emphasis than an absolute difference in perspective. They are both methods that combine field studies, user involvement and modeling. However, in activity or usage-centered design models are in the forefront and drive development that is then evaluated through user studies. On the contrary user-centered design methods rely more on user studies and feedback. It is out of the scope of this paper to discuss the implications of the emphasis in both approaches, we provide examples from activity-centered design because this paper reports our experience applying those methods and producing estimations from a consistent sample of projects. The fact that usage (or activity) based design is more narrowly focused on user performance and on the creation of tools to enhance the efficiency and dependability of user performance is not detriment of the application of the same techniques to other user-centered design methods given that they are used to produce some for of requirements models based on use-cases.

In the following subsections we discuss the implications of these methods in weighting actors and use cases.

3.1 Weighting Actors

Effective interaction design involves understanding users and their needs. Like in the conventional UML and UP tradition, in usage-centered design users who interact with a system are referred to as actors. However, unlike conventional UML the concept of actor is expanded through user roles, an additional abstraction representing a relationship between users and a system. According to Constantine, in its simplest form, a




R01 - Current-Sales-and-Ticketing Role

context (of role performance): isolated in booth, likely facing queue of customers; final step in performance promotion and sales

characteristics (of role performance): relatively simple task performed repeatedly with some training, prior experience likely; performed under some time pressure, which increases as show time approaches

criteria (for support of role performance): simplified, highly efficient interaction; foolproof means of identifying customer, guarantee that all the right tickets are dispensed and received by the customer

Fig. 1. Context map for an example ticketing application (taken from [12]), on the top is the user role map including several actors and their supporting roles and the detail description of the *Current-Sales-and-Ticketing Role*

role can be described by the context in which it is performed, the characteristic manner in which it is performed, and by the evident design criteria for effective support of performance of the role [13]. An example from [13] is shown bellow, both in terms of the underlying context map (an adapted form of a UML use-case diagram) and the detailed user-role description for the *Current-Sales-and-Ticketing Role*.

The difference between a context map produced by the usage-centered design method and a conventional use case model is abyssal in terms of the richness of the information conveyed about the complexity underlying each actor. Considering that a conventional UML model would represent this problem with 12 actors (A1 to A6 and 6 system actors) and applying the original UCP method the estimation would be 4 simple actors (credit card reader, envelope printer, ticket printer, venue/event manager), 2 average actors (credit-card network, accounting system) and 6 complex actors (A1 to A6).

Table 1. Estimation based on the original UCP method for the actors in the ticketing application

Actor type	Description	Qnty	Weight factor	Subtotal
Simple	Defined API	4	1	4
Average	Interactive or protocol driven	2	2	4
Complex	GUI	6	3	18
Total				**26**

Table 1 illustrates the weighting of the 12 use-cases of the ticketing application described by the context map model in Figure 1. Analyzing the model we can easily verify that all the information regarding the user-roles played by each actor are discarded in the complexity weighting of actors. Even if we consider the issues discussed in section 2.2 the only difference would be the zero weighting of the two indirect actors A1 and A2, which means they will be discarded in the estimation reducing the quantity of complex actors to 4 and the total estimation to 20.

Our proposal is to take into consideration the user roles and additional information provided by methods like usage-centered design to inform the early estimation through the weighting of the actors. Clearly the number of roles supported by each actor provides an important way to infer the complexity associated with each actor and consequently the weigh factor that should be applied. Additionally usage-centered design suggest the concept of focal role together with several relationships that make up a model designated user role map (check [13] for a detailed discussion). Focal roles are roles recognized as particularly important for a successful design, they serve a central focus to the rest of the design process, but not to the exclusion of other user roles [13].

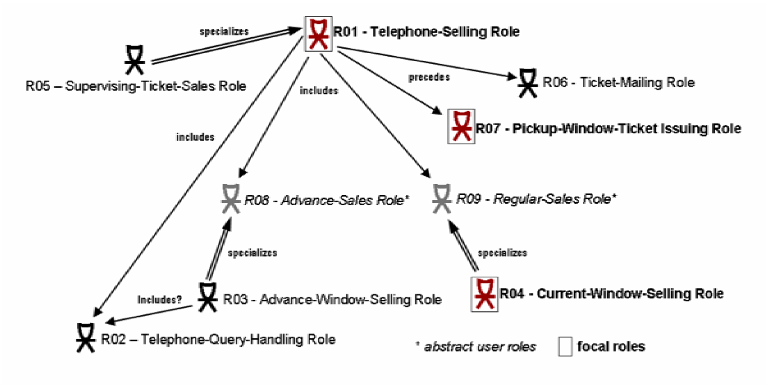

specializes
R01 - Telephone-Selling Role
R05 – Supervising-Ticket-Sales Role
includes
precedes
R06 - Ticket-Mailing Role
R07 - Pickup-Window-Ticket Issuing Role
includes
R08 - Advance-Sales Role*
R09 - Regular-Sales Role*
specializes
specializes
Includes?
R03 - Advance-Window-Selling Role
R04 - Current-Window-Selling Role
R02 – Telephone-Query-Handling Role
* abstract user roles
focal roles

Fig. 2. User role map for an example ticketing application (taken from [13]), focal roles are highlighted

Our experience working and consulting on several projects applying usage-centered design methods suggests the following revised heuristics for actor weighting in interactive software projects:

- Simple system actors (factor 1) – system actors that communicate to the system through an API;
- Average system actors (factor 2) – system actors that communicate to the system through a protocol or data store;
- Simple human actors (factor 3) – human actors interacting with the system supported by one user role;
- Average human actors (factor 4) – human actors interacting with the system supported by 2 or 3 user roles or a single focus role;
- Complex human actors (factor 5) – human actors interacting with the system supported by more than 3 user roles or more than one focus role;

Table 2. Estimation based on the modified iUCP method for the actors in the ticketing application provided in Figures 2 and 3

Actor type	**Description**	**Qnty**	**Weight factor**	**Subtotal**
Simple system	Defined API	4	1	4
Average system	Interactive or protocol driven	2	2	4
Simple human	Support one user role	1	3	3
Average human	Support 2-3 user roles or 1 focal role	2	4	8
Complex human	Support more than 3 user roles or more than 1 focus role	1	5	5
Total				**24**

As we can see from Table 2 the human actors are now divided into simple, human and complex with respectively weight factors of 3, 4 and 5. The analysis of the examples provided in Figure 1 and 2 following the revised heuristics for iUCP suggest the following classification:

- A1 – indirect actor, zero-weighted (0)
- A2 – indirect actor, zero-weighted (0)
- A3 – supports 2 roles (R01 and R02) one focal – average human actor (4)
- A4 – supports 3 roles (R03, R04, R07) two focal – complex human actor (5)
- A5 – supports 2 roles (R01 and R02) one focal – average human actor (4)
- A6 – supports 1 role (R06) not focal- simple human actor (3)
- A7 – A12 – are system actors, 4 simple and 2 average according to the initial weighting which doesn't change on iUCP

From the above classification we can conclude that in a simple example there is a total difference of four use case points (from 20 to 24) based on the revised actor weighting (assuming both approaches zero-weight the indirect actors). Although this difference might look insignificant in a real-world project with three times more actors and roles the impact is far from being neglectful. But our experience shows that more important than the calculation itself; the iUCP revised heuristics provide systematic guidance that prevents many of the problems indentified by Dieve in [8]. The original method simply classifies human-actors with the same weight factor which is arguable a consistent heuristic for interactive applications that usually have one or two system actors and more than one dozen human-actors. By assigning the majority of the actors with the same complexity weight factor the UCP method becomes arguable useful for interactive system development.

3.2 Weighting Use-Cases

Since their introduction by Jacobson in object-oriented software engineering, use cases have enjoyed a seemingly explosive growth to become ubiquitous in both development methods and development practice [14]. Part of their success can be attributed to the simplicity of the concept itself but probably also a consequence of their imprecise definition. In fact entire books and thesis have been devoted to the discussion of what a use-case is and we can find many instantiations of use-case descriptions that vary in scope, detail, focus, format, structure and style. It is not our purpose here to discuss use-cases, a thorough albeit controversial discussion can be found in [14].

It seems obvious that an estimation method relying on weighting use-cases will suffer from the same uncertainty that we can find in the literature about using the concept to structure requirements. However, in usage-centered design they are clearly defined through the pioneering and fundamental concept of essential use-cases [14]. Unlike conventional uses cases, defined in the UML specification (see section 2), essential use cases are define by Constantine [14] as:

> *"a single, discrete, complete, meaningful, and well-defined task of interest to an external user in some specific role or roles in relationship to a system, comprising the user intentions and system responsibilities in the course of accomplishing that task, described in abstract, technology-free, implementation-independent terms using the language of the application domain and of external users in role".*

The difference between the two definitions is not subtle as much as the consequences for designing interactive systems. Not only essential use-cases are more abstract, generalized and technology-free descriptions of the essence of a given problem, but also more importantly they are described in a systematic sequence of steps divided between user intentions and system responsibilities. The essential nature of these steps provides a systematic way of identifying transactions, which are key to classify use-cases in the UCP method. The problem is better explained through a example. Bellow is a popular example provided in textbooks for the UML describing a use-case for withdrawing cash (taken from the EPF wiki at www.eclipse.org):

1. *The use case begins when Bank Customer inserts their Bank Card.*
2. *Use Case: Validate User is performed.*
3. *The ATM displays the different alternatives that are available on this unit. In this case the Bank Customer always selects "Withdraw Cash".*
4. *The ATM prompts for an account. See Supporting Requirement SR-yyy for account types that shall be supported.*
5. *The Bank Customer selects an account.*
6. *The ATM prompts for an amount.*
7. *The Bank Customer enters an amount.*
8. *Card ID, PIN, amount and account is sent to Bank as a transaction. The Bank Consortium replies with a go/no go reply telling if the transaction is ok.*

9. *Then money is dispensed*
10. *The Bank Card is returned.*
11. *The receipt is printed*
12. *The use case ends successfully*

According to the original UCP heuristics this use-case will likely be classified as complex since it involves arguably more than 7 transactions. However when we look at the essential use-case counterpart description, the number of essential steps is much different. Bellow we have 2 "essential transactions" (system responsibilities), which would classify this use case as simple. The discrepancy is vast and questionably the reason underneath the problems applying UCP across companies, teams and even projects. Interestingly enough the heuristics for assigning weight factors to use-cases are highly dependent on assumptions about the user-interface. In section 2.1 a simple use case corresponded to a simple user interface, an average to a moderate user interface and a complex use-case to a complex user interface. However conventional use cases don't reflect the division between user intentions and system responsibilities that conveys the notion of interaction (i.e. a interaction happens when a user specifies an intention to the system).

User intention	***System responsibility***
identify self	
	check identity
specify amount	
	provide cash

Indeed, in usage-centered design an essential use case represents a single, discrete intention carried out by a user in some role. This provides a systematic way of expressing transactions as steps in a dialog in which user intentions and system responsibilities are abstract, simplified, and stripped of all assumptions about technology or implementation. Originally this form of description was intended to get closer to the essence of the task from the perspective of the user in a role, thus avoiding unintended or premature assumptions about the user interface to be designed [14]. When applied to estimating use-cases it becomes an important way to retain scope and prevent the granularity problems described in section 2.2.

However estimating transactions is not the only concern when assigning weight factors to use-cases. The heuristics specifically mention two additional criteria depending on the conceptual architecture: (i) the number of entities manipulated in the context of the use case and, (ii) the number of classes implementing the use case. The relationship between use-cases and implementation classes is accomplished in the UML convention using the entity/control/boundary pattern. However this pattern doesn't reflect the separation of concerns required for interactive system development, boundary classes encapsulate both interface to human actors and system actors and thus there is no clear distinction between human and system interaction. As a consequence the implementation classes extracted from the use-cases will not reflect the complexity of the user-interface, which is key to sustain the assignment of weight factors to use-cases.

In [15] we proposed an extension of this framework to include two important concepts that reflect the user intentions that form the basis of usage-centered design: tasks and interaction spaces. Therefore, the boundary-control-entity pattern is extended with additional task and interaction space class stereotypes:

– <<task>> classes that are used to model the structure of the dialogue between the user and the system in terms of meaningful and complete sets of actions required to achieve a goal; and
– <<interaction space>> classes are used to represent the space within the user interface of a system where the user interacts with all the functions, containers, and information needed for carrying out some particular task or set of interrelated tasks.

In [16] we described how to extract the extended architecture from the essential use-cases. The process is highlighted in Fig. 3, where <<task>> classes originate from user intentions, <<control>> and <<entity>> classes from the system responsibilities and finally <<interaction spaces>> from the crossing of both. This process not only increases the traceability in usage-centered design but becomes central to identify the number of entities and overall classes required to implement a given use-case. In addition, and contrary to the conventional UML approach, the architecture reflects the "complexity" of the structure of use which will eventually originate the user-interface, hence it provides a mechanism to map the complexity of the UI to the use-cases which is not achievable with the original method.

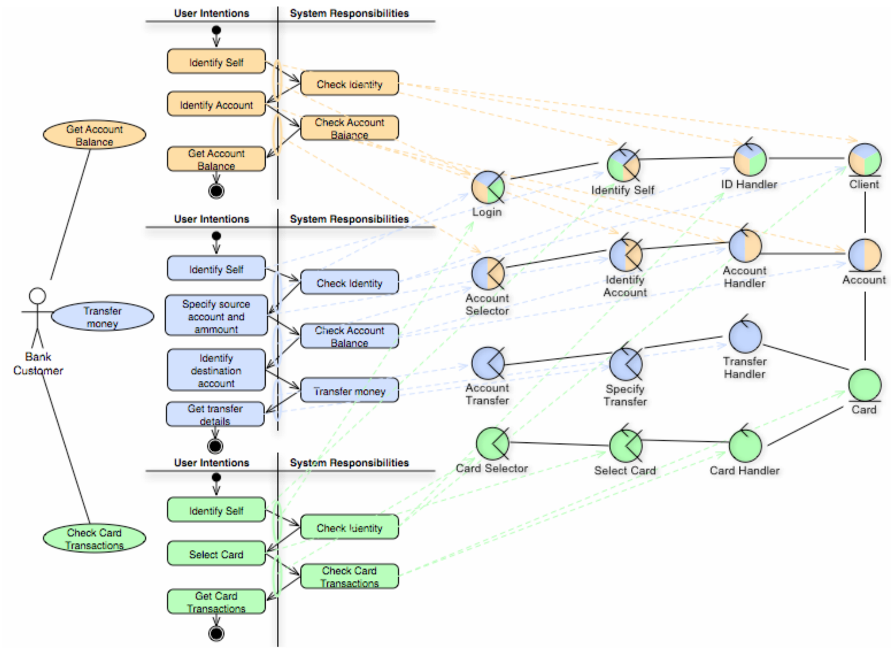


Fig. 3. A conceptual architecture extracted from essential use cases (taken from [16])

The example provided in Fig. 3 for an ATM system illustrates how the usage-centered design architecture can be used to inform the classification of use-cases in the iUCP method. Unlike the heuristics applied to weight actors we don't propose to change the weighting factors for the use-cases, but simply introduce the usage-centered design concepts of essential transaction and implementation class described previously. Therefore transactions are conceived as the total number of system responsibilities identified in a given essential use-case and conversely implementation classes are considered as the total number of classes originating from an essential use case according to the approached described in [16] depicted in Fig. 3 through the dashed lines connecting the use-case descriptions to the conceptual architecture.

Table 3. Estimation based on the modified iUCP method for the use-cases in the ATM application example provide in Fig. 3

Use case type	**Description**	**Qnty**	**Weight factor**	**Subtotal**
Simple	Simple UI, 1 entity, ≤3 transactions	0	5	0
Average	Average UI, 2-3 entities, 4-7 transactions	1	10	10
Complex	Complex UI, >3 entities, >7 transactions	2	15	30
Total				**40**

As we can see from Table 3 for a small example it is clear how to apply the heuristics when considering the small example provided in Fig. 3. We simply count the number of system responsibilities and user intentions per use case and the number of originating implementation classes. This contrasts the uncertainty that we could envision from starting with a conventional use case. Not only it is harder to isolate transactions but there is little guidance regarding the number of implementation classes corresponding to each use case and in particular reflecting the complexity of the UI.

4 Conclusions and Future Work

In this paper we described iUCP, a modified version of the use-case point software estimation method that leverages the techniques from usage-centered design to improve the heuristics traditionally used in Software Engineering to create product cost estimates early in the development lifecycle. Early estimation of software is critical industry challenge and we argue that our approach not only helps bridge the gap between SE and HCI but also accomplished that providing software development with useful systematic guidance towards producing early estimates for software based products. With the profusion of agile incremental and evolutionary approaches used in interactive software development it is increasingly important to find ways to enable both HCI and SE experts to collaborate early in the lifecycle. By proposing to use key usage-centered development techniques - like use roles, essential use-cases and interactive conceptual architectural models - we not only bridge the gap between SE and HCI but more importantly illustrate how HCI techniques can be useful in traditional engineering practice of software development like estimation and models.

The ability to accurately predict the cost of a project early in the lifecyle is a major differentiator for the software industry. The capability of combining SE and HCI, enables cross-fertilization between the two disciplines and encourages new ways of collaboration between interaction designers and software developers. This brings a new perspective to developers because they can foresee the advantage of using HCI techniques early in the lifecycle. Conversely interaction designers can better understand the impact of their models of users and recognize the impact of UI elements at the architecture level, building common ground for other activities like prioritizing development and planning releases.

The iUCP method was developed building on statistical data from usage-centered development projects were the author consulted and worked in the past years. However, a systematic evaluation at the metric level requires extensive data collection and analysis over the course of years. Our remit here is not to evidence the validity of the estimation method, which can be found elsewhere (for instance in [8]). The adaptations of the UCP method are minimal so that we can preserve the integrity of the original model. Our goal with the iUCP adaptation of the method is to help both software developers and interaction designers to apply heuristics that are suitable for interactive applications and work consistently across and within projects. We have taught and applied the iUCP method both in graduated SE and HCI courses with good results in terms of student's capability to accurately estimate cost and effort of interactive applications. We have also backtracked iUCP on several real-world projects against actual data of development effort and cost with good results. We are currently planning to develop an empirical study that could validate the results against standard UCP, and also plan to provide automated tool support for iUCP.

References

1. Takeuchi, H., Nonaka, I.: The New New Product Development Game (PDF). Harvard Business Review (January-February 1986)
2. Seffah, A., Metzker, E.: The Obstacles and Myths of Usability and Software Engineering. Communications of the ACM 47(12), 71–76 (2004)
3. Boehm, B.: Software engineering economics. Prentice-Hall, Englewood Cliffs (1981)
4. Karner, G.: Resource Estimation for Objectory Projects. Objective Systems SFAB (1993)
5. Schneider, G., Winters, J.P.: Applying Use Cases: A Practical Guide. Addison-Wesley, Reading (1998)
6. OMG: The UML superstructure, `http://www.uml.org`
7. Dieve, S.: Use cases modeling and software estimation: Applying Use Case Points. ACM Software Engineering Notes 31(6) (2006)
8. Carrol, E.: Estimating Software Based on Use Case Points (2002)
9. Constantine, L.L., Lockwood, L.A.D.: Software for use: a practical guide to the models and methods of usage-centered design. Addison Wesley, Longman (1999)
10. Nunes, N.J., Cunha, J.F.: Wisdom: A Software Engineering Method for Small Software Development Companies. IEEE Software 17, 113–119 (2000)
11. Nunes, N.J., Cunha, J.F.: Whitewater Interactive System Development with Object Models. In: Harmelen, M. (ed.) OOUID. Addison-Wesley, Reading (2001)
12. Cooper, A.: About Face 3.0: The Essentials of Interaction Design. Wiley, Chichester (2007)
13. Constantine, L.: Users, Roles, and Personas. In: Pruitt, Aldin (eds.) The Persona Lifecycle. Morgan-Kaufmann, San Francisco (2006)

14. Constantine, L., Lockwood, L.: Structure and Style in Use Cases for User Interface Design. In: van Harmelen, M. (ed.) OOUID. Addison-Wesley, Reading (2001)
15. Nunes, N., Cunha, J.F.: Towards a UML profile for interactive systems development: the Wisdom approach. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 101–116. Springer, Heidelberg (2000)
16. Nunes, N.: What Drives Software Development: Bridging the Gap Between Software and Usability Engineering. Human Computer Interaction Series. Springer, Heidelberg (2008)