USER-SYSTEM INTERACTION DESIGN FOR REQUIREMENTS MODELLING

HALIL I. ERHAN, ULRICH FLEMMING
(1)College of Information Technology, UAE University Al-Ain UAE.
(2)School of Architecture, Carnegie Mellon University, Pittsburgh, PA.
hierhan@uaeu.ac.ae and ujf@cmu.edu

Abstract. RaBBiT (Requirements Building for Building Types) provides computational support for architectural programming or requirements modelling in building design. A highly interactive graphical user interface (GUI) allows users to adapt RaBBiT to various programming styles and terminologies. Since users are not expected to have any prior *computer* programming experience, the design of RaBBiT's GUI posed particular challenges, which we attempted to meet through a direct-manipulation interface based on the model-world metaphor.

1. Introduction

RaBBiT (Requirements Building for Building Types) is a computer-based tool aiming to assist architects and other stakeholders in building design during the programming[1] phase of that process (Erhan, 2003; Erhan and Flemming, 2004). The motivation for developing a system like RaBBiT is discussed in Erhan (2003). In brief, we believe that computer-based tools for modeling design requirements have the potential of overcoming some of the bottlenecks that plague traditional programming methods in building design (limited change propagation within a program, programming knowledge upgrade, maintenance and transfer within a firm, etc.). But the computational tools currently used in architectural programming are limited to simple, highly specialized database or spread-sheet applications that are unable to address the observed bottlenecks in a general way. Academic research also has paid little attention to computer-assisted architectural programming. Notable exceptions are the SEED-Pro (Akin et al., 1995) and SEED-Pro II (Donia, 1998) systems developed at Carnegie Mellon University as part of the SEED project (Flemming and Woodbury, 1995). RaBBiT continues this line of investigations.

Any attempt to develop a general tool to support architectural programming is immediately confronted with a fundamental problem: There exists no generally

1. "Programming" refers to the phase in architectural design where design requirements are compiled. We use the term interchangeably with "requirements modelling" as it is known from disciplines like software engineering.

accepted programming method; even the terminology used differs significantly across firms and the programming literature. However, we discovered that underneath the methodological and terminological variations, there lies a shared information-processing logic, and we were able to design RaBBiT around a *knowledge-modelling component* that allows users to define interactively the terms, concepts and methods a firm habitually uses during the programming phase (specialized, if needed, for various recurring building types). The resulting knowledge models can be used by RaBBiT's *program generation component*, again in interaction with the user, to produce a specific program for a specific project.

We have described the approach underlying RaBBiT elsewhere (Erhan 2003; Erhan and Flemming, 2004). In the present paper, we focus on the design of its graphical user interface (GUI) because the combination of generality, (computational) programmability and interactivity pose particular usability challenges, given that the system is meant to be used by domain experts with little or no experience in *computer* programming.

Despite our present emphasis on RaBBiT's GUI, we have to introduce the underlying conceptual framework because the user interaction with RaBBiT has to be understood in the context of this framework. We do this in Section 2. Subsequent sections introduce the approach we used to design RaBBiT's GUI, from general conceptual decisions to selected design and implementation details.

2. Conceptual Framework

An extensive literature survey and three detailed case studies convinced us that architectural programming is essentially *an information refinement process that gradually transforms higher-level (non-spatial) requirements into measurable and operational (spatial) requirements at lower-levels* (see Erhan (2003) for a detailed discussion of our sources). Furthermore, this process resembles Means-Ends Analysis (MEA), a problem-solving method first introduced in the General Problem Solver (GPS) (Newell & Simon, 1972; Simon, 1989), which solves problems by successively reducing the differences between an initial and a desired state through a process of step-wise refinement in which the means derived at one level become the ends to be satisfied at the next lower level (Sternberg, 1996).

In classical MEA, the transition from higher to lower levels is strictly hierarchical, and the refinement process has a tree-like structure. In architectural programming, however, the refinement process can be multi-directional and may contain cycles. In order to capture these cyclic relationships, MEA needs to be extended so that one means can satisfy different ends, possibly at different levels higher up. We call the resulting process Generalized MEA (GMEA). ýFigure 1 illustrates this in terms of the concepts and methods proposed by a particular author (Duerk, 1993).

GMEA is attractive in the context of RaBBiT because it is (a) general enough to capture any one of the programming methods we have encountered, and (b) operational enough so that it can be implemented as a computer process.

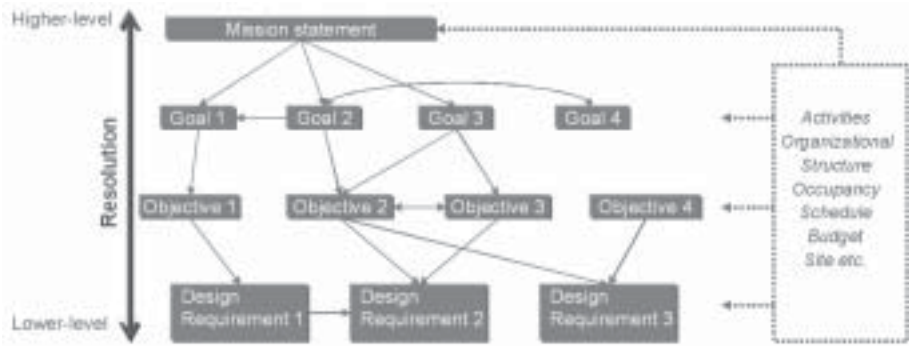


Figure 1. Sample transition from high-level to low-level requirements.

Knowledge modelling and program generation in RaBBiT are based on a GMEA framework that relies internally on five basic information types:

2.1. COMPONENTS representing salient programming concepts, or means and ends in the GMEA sense (e.g. a mission statement). In a concrete knowledge model, each component may belong to a particular category with a user-specific name. Components are the generic entities RaBBiT uses to represent any one of these concepts. Readers familiar with requirements modelling may think of components as requirements.

2.2 CONSTRUCTS representing attributes or parameters associated with components that have values, which can be directly assigned by users or derived from the values of other parameters through formulas entered by the user.

2.3. GLOBAL CONSTRUCTS representing programming parameters critical for a building type and not associated with a particular component (e.g. the number of students to be served by a school).

2.4. ASSOCIATIONS between components and constructs can be used to represent (a) means–end relationships between components at the same or from higher to lower levels (called *dependencies* below); (b) other component associations (like desired proximities between two design objects); (c) associations between constructs, for example, to compute the value of a parameter from the values of other parameters as happens in a spreadsheet. Users may also attach conditions to an association, for example, to indicate that a design object is needed as part of a more complex design object if (and only if) certain other functional requirements exist.

2.5. INFORMATION CATEGORY LEVELS allow the user to classify and group components according to a particular programming approach. For example,

a user following the approach outlined in ýFigure 1 may define the category levels "mission statement," "goal," "objective," and "requirement". Category levels are the main mechanism allowing users to express their own notion of programming and to "wrap" a familiar terminology around RaBBiT's generic components. However, the use of this feature remains optional because defining category levels and assigning components to them poses an additional burden on users and may not be needed for less complex building types.

3. Users and Functions

The primary users of RaBBiT are practitioners with a particular interest in architectural programming for specific building types.[2] Given the generality of GMEA, there may even be requirements modellers in other domains. Specifically, a primary user may play one of two roles (Figure 2).

3.1. THE ARCHITECTURAL PROGRAMMING KNOWLEDGE MODELLER (APM) models interactively with RaBBiT the process to be used when generating a program for a recurring building type independently of project specifics. The APM is able to edit interactively the resulting *programming knowledge model* (PKM) and to save it persistently. When generating a PKM, an APM can use any preferred terminology and model any programming process consistent with GMEA.

3.2. THE ARCHITECTURAL PROGRAM COMPOSER (APC) is able to retrieve a PKM to compose—with RaBBiT's assistance—a program for a particular project calling for the respective building type. After a program has been generated, the APC can modify the generated program, save it, and share it with other people or applications in various formats.

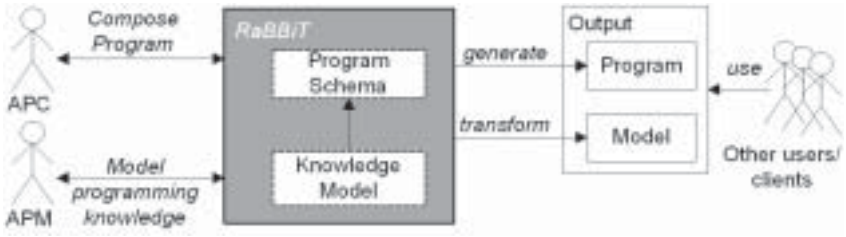


Figure 2. Overall system structure: users and functions.

2. Secondary users may be other computer-aided design and reasoning systems or clients who want to utilize a PKM or a generated program in their own way—such as layout generators or budget planners.

4. User Interface Design Paradigms

RaBBiT's GUI adapts the *direct manipulation* paradigm as introduced by Shneiderman and Plaisant (1982, 2005). This paradigm is commonly used in current computer applications relying heavily on visual or graphical information that can be shown on a computer screen as identifiable objects and interactively modified (shape, location, etc.). The paradigms do away with written command languages and let users manipulate directly (domain) objects visible on the screen. At the same time, users are able to observe the consequences of an action (immediate feedback). This interaction style appears appropriate for RaBBiT not only because of its inherent advantages: Designers, in particular, are trained to create and organize visual information—such as flowcharts, schemas, or spatial layouts—and may therefore take naturally to the direct manipulation of graphical objects.

RaBBiT's GUI provides a graphical *modelling area* showing objects representing components, constructs and associations interactively created and modified by the user through direct manipulation. By creating and modifying these objects, users create and modify indirectly the PKM internal to RaBBiT.

Hutchins and his colleagues provide a more cognitive justification for direct manipulation based on a *model-world metaphor* (Hutchins et al., 1986). In this metaphor, the user interface is not only a medium between user and system, but represents the system itself from the user's perspective. It establishes "a world where the user can act, and which changes state in response to user actions". Ideally, users would have the sense that they are acting not upon graphical entities, but upon the objects of the task domain itself and "engage" with these objects.

In designing RaBBiT's GUI, the model-world metaphor guided us specifically in our handling of the possibly complex associations among components and constructs in a PKM, which we display as a *graph*. Its nodes represent components and global constructs; dependency and relational associations are shown as lines connecting nodes. The lines belong, in fact, to two overlapping sub-graphs. Dependencies form an acyclic directional sub-graph, while relational associations form a sub-graph that is bidirectional and cyclic.[3] RaBBiT's GUI conveys this distinction by varying the display of lines in the respective sub-graphs. Users are able to visualize and manipulate a PKM through the nodes and lines of the graph.

Parametric associations, on the other hand, are too complex to be shown in the same graph. They would clutter the view and prevent users from assessing visually the integrity of the graph—and of the model, for that matter. We therefore handle parametric associations through separate tables that can be modified in a spreadsheet-

3. We may note in passing that RaBBiT composes a program by traversing the dependency graph, starting with the global constructs and evaluating the conditions and expressions it encounters to compute the values of constructs.

like fashion. The spreadsheets can be viewed as a sub-metaphor with which many users should be familiar.

Figure 3 shows the display of a PKM in RaBBiT. Component nodes appear as small, floating tables with a yellow (light) header and rows showing the associated constructs in the form of parameters or attributes. Global constructs are also represented as floating tables, but with a blue (dark) header and exactly one parameter. A dependency is shown as an arrow and, if needed, an attached box containing a dependency condition. A red arrow indicates that the dependency condition is currently true; the colour turns blue when the condition becomes false. Other associations are shown with a light gray line connecting the two components involved. An attached label indicates the type of the association.

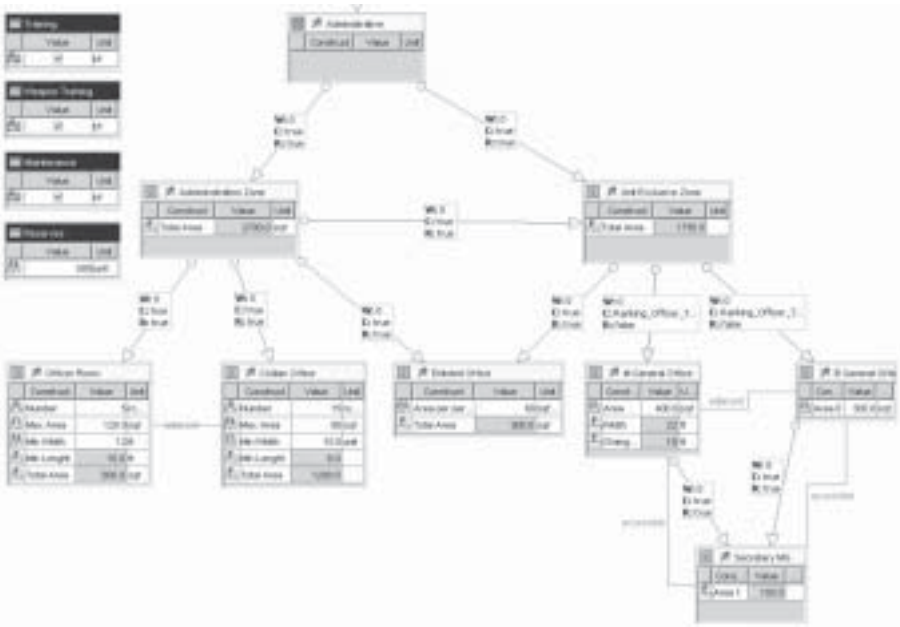


Figure 3. Example of graph-based modeling in RaBBiT.

Since RaBBiT lets the user view and manipulate a PKM as a graph, its central work window provides a view of that graph, which is split into two areas (ýFigure 4). The *tree-view* area displays all components with respect to the category level under which they fall. The *graph-view* is the main interaction area where a knowledge model is composed. The status bar at the bottom of the window prompts users and informs them about the state of the system or the consequences of their actions.

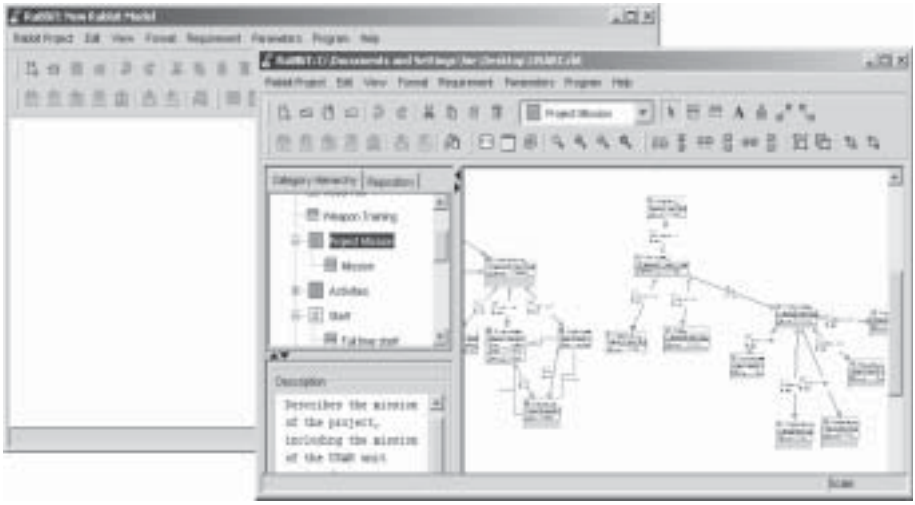


Figure 4. RaBBiT's graph modeling area.

5. Use Cases

The APM and APC accomplish their respective goals through a series of actions or operations performed in interaction with RaBBiT that involve a host of interface objects or widgets, from commands accessible through menus or tool boxes to the various dialog boxes and system messages needed to accomplish individual tasks. The respective sequences of actions and GUI components supporting them should be designed not in an ad-hoc fashion, but in a systematic way to assure that the system provides its functionality to users as intended, that is, completely, effectively and in a way that users can understand and remember.

We have found the *use case* approach particularly effective for designing a GUI that conforms to these requirements (Flemming et al., 2003). But note that this approach is capable of guiding not only the physical GUI design of an application, but also of structuring and controlling the entire development process through all of its phases (Jacobson et al., 1999), that is, use cases can—and should—be used to handle more than physical GUI design, an aspect we cannot pursue further in the present paper.

A use case describes a "sequence of actions an actor (user) performs using a system to achieve a particular goal" (Rosenberg, 1999) or "a sequence of actions a system performs that yields an observable result of value to a particular actor" (Booch et al., 1999). Specifying a set of use cases covering all actors (users) is the all-important first step in use case-driven software development.

TABLE 1. Main use cases of RaBBiT

Session Control	Knowledge Modelling	Program Generation	View Manipulation
Start new session Open an existing PKM/program Save a PKM/ program Close a PKM/ program Exit session	Define/modify requirement category levels Create (insert) a component Insert a construct into a component Create a global construct Insert an association between two components Modify a component/ construct/ association Remove a component/ construct/ association	Input project-specific information Modify global parameters Generate a program Display sample program	Relocate component Reroute component associations Zoom (in and out) Pan Layout graph Pan from overall view Align components

The APM and APC are the primary actors interacting with RaBBiT, and use cases have to describe, from their perspective, the individual tasks they can perform in interaction with the system to accomplish their overall objective (model building, program generation).

We may group the use cases defining the functionality of RaBBiT under four categories: (a) session control, (b) knowledge modelling, (c) program generation, and (d) graph view manipulation. Table 1 lists the most important use cases in each category that have been implemented in the current version of RaBBiT.

The following example illustrates the level of detail or "granularity" at which we like to specify use cases. It also demonstrates how use case development and GUI design go hand-in-hand.

Use Case: Create a component

Primary actor: The APM (programming knowledge modeller)

Description: The APM creates a component as part of the current PKM. The component represents a programming concept in the APM's terms.

Preconditions: A PKM was created and opened in RaBBiT for editing. The APM selected a category level, if desired.

Flow of Events:

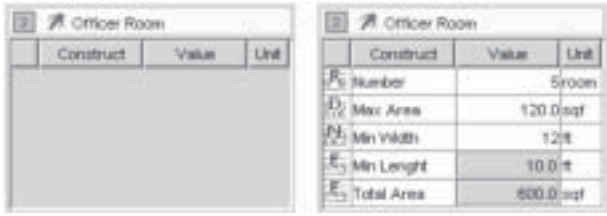
1. The APM selects the "*insert a component*" command from the component menu or toolbox.
2. RaBBiT prompts the APM in the status bar to draw the component (as a rectangle).
3. The APM draws a rectangle in the editing window by defining two opposite corners to indicate where the component should appear.
4. RaBBiT creates a view of the component as a floating table and makes it available for editing (Figure 5).

5. The APM enters a name for the component in the name field. He/she may continue with the various use cases to insert constructs.

Alternative Events:

3a. The APM can abort the use case by clicking on the modelling area, selecting another command, or pressing the *escape* key on the keyboard.
5a. If the entered name has been used before, RaBBiT opens an alert box asking the APM to enter another name or abort the use case.

Post-conditions: A component object representing the programming concept is created internally and added in the knowledge model. This object is accessible through its GUI representation.



(a) (b)

Figure 5. A component view: (a) without and (b) with parameters.

We cannot describe the other use cases at this level of detail and have to restrict ourselves to highlighting specific portions of a few other use cases that illustrate how we dealt with “trickier” aspects of RaBBiT’s GUI.

Figure 6 shows how an association between two components can be established by selecting the components with a pointing device and how RaBBiT gives feedback by displaying the association graphically.

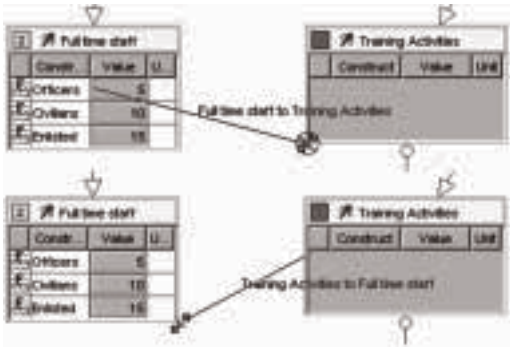


Figure 6. Inserting associations between components.

Figure 7 shows two dialog boxes used when the value of a construct is being set or edited. In the Figure (a) shows the box used when the value is set by reference

to the value of another construct, while (b) shows the dialog box for entering an expression to compute the value, possibly by referring to the values of selected other constructs in different components; the expression may also contain conditionals referring to values of other constructs. We cannot present the details here—suffice it to say that this operation resembles writing an expression in a spreadsheet.

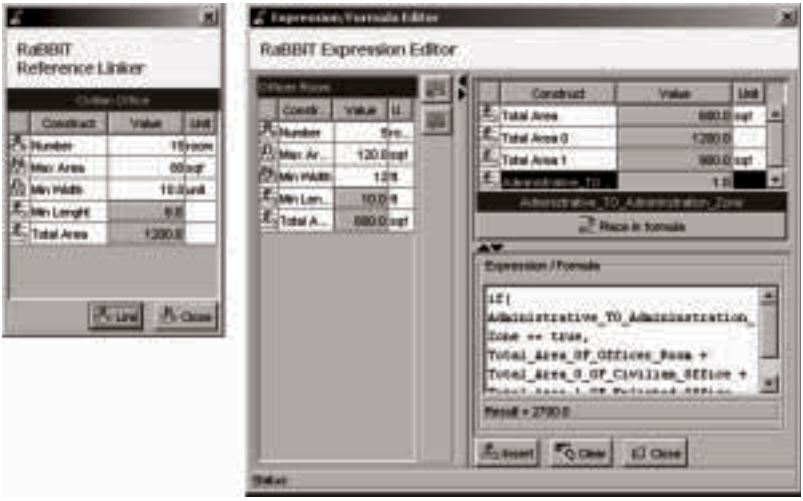


Figure 7. Dialog boxes for inserting a value by (a) reference and by (b) an expression.

6. Conclusion and Future Work

Our work on RaBBiT demonstrates that despite the diverging terminologies and methods used in programming practice, there exists a shared underlying logic that can be captured through Generalized Means–Ends Analysis (GMEA). Furthermore, programming processes following GMEA can be made operational in the form of a programming knowledge model (PKM) consisting of components and constructs. These generic entities can be customized using category levels and associations. The resulting PKM can be complex. But it is possible to design a direct-manipulation graphical user interface (GUI) based on the model-world metaphor that allows users without experience in computer programming to interactively construct a PKM and to compose architectural programs automatically from it for specific building projects.

In implementing RaBBiT's GUI, we followed well-established usability principles as introduced in (Nielsen 1995). However, resource limitations have

4. RaBBiT has been implemented in Java using additional open-source third-party Java libraries such as JGraph, JGraphPad, and JEP.

prevented us from testing the first RaBBiT prototype[4] with intended end-users. This should be done in the context of real-world building projects and with firms having established programming expertise.

We also observe that it is possible to use RaBBiT in other domains than architectural design. GMEA and the concepts implementing it in the context of RaBBiT are generic enough to model product requirements for other systems composed of multiple parts that must satisfy various requirements and may vary depending on variable overall specifications. For example, the requirements for a custom-built computer or a manufacturing assembly tools can be modeled in this way. We would like to test with experts from such domains to what extent our framework (and RaBBiT) can be used in their domains.

References

Akin, Ö., Sen, R., Donia, M. & Zhang, Y. 1995. SEED-Pro: Computer assisted architectural programming in SEED. *Journal of Architectural Engineering*, vol. 1, no. 4, pp. 153–161.

Booch, G., J. Rumbaugh & I. Jacobson 1999. *The Unified Modeling Language User Guide*. New York: Addison-Wesley, p. 19.

Donia, M. 1998. Computational Modeling of Design Requirements for Buildings. Doctoral Thesis. School of Architecture, Carnegie Mellon University. Pittsburgh, PA.

Duerk, D.P. 1993. *Architectural Programming: Information Management for Design*. New York: Wiley, p. 20, 36.

Erhan, H.I. 2003. Interactive support for modeling and generating building design requirements. Doctoral Thesis. School of Architecture, Carnegie Mellon University. Pittsburgh, PA.

Erhan, H. I. & U. Flemming 2004. Interactive Support for Modeling and Generating Building Design Requirements. *Proc. Generative-CAD 2004 Symposium*. Carnegie Mellon University, Pittsburgh, PA.

Flemming, U. & R. Woodbury 1995. Software Environment to Support Early Phases in Building Design (SEED): Overview. *Journal of Architectural Engineering*, vol. 1, no. 4, pp. 147–152.

Flemming, U., H.I. Erhan, & I. Ozkaya. 2003. Object-Oriented Application Development in CAD. *Automation in Construction*, vol. 13, no. 2, pp. 147–158.

Hutchins, E.L., J.D. Hollan, & D.A. Norman 1986. Direct manipulation interfaces, in Norman, D.A. and Draper, S.W. (Eds.) *User Centered System Design: New Perspectives on Human-Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum, pp. 87–124.

Jacobson, I., Booch, G., Rumbaugh J. 1999. *The Unified Software Development Process*. Reading, MA: Addison-Wesley.

Newell, A. & H. A. Simon. 1972. *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall.

Nielsen, J. 1995. *Usability Engineering*. Boston, MA : Academic Press.

Rosenberg, D. 1999. *Use Case Driven Object Modeling with UML*. A Practical Approach. Reading, MA: Addison-Wesley, p. 38.

Simon, H.A. 1989. *Models of Thought*. Volume II. New Haven, CT. Yale University Press, pp. 36–37.

Sternberg, R.J. 1996. *Cognitive Psychology*. Philadelphia, PA: Harcourt Brace, p. 483.

Shneiderman, Ben and Plaisant, C. 2005, *Designing the User Interface, Strategies for Effective Human-Computer Interaction*. Reading, MA : Addison-Wesley.