

Deriving Object-Oriented Design from Functional Requirements

Thomas Harron
Department of Computer Science
University of Wisconsin – La Crosse
La Crosse, WI 54601
harron.thom@students.uwlax.edu

Abstract

The process of developing an object-oriented software design document from a functional requirements specification is a fundamental issue in software engineering. Important process issues such as discovering data elements present in the requirements, deriving design objects, specifying object relationships, and others can be obscured as software engineers exert significant low-level clerical effort to produce consistent, traceable, standards-compliant requirements and design documentation. This paper describes a software tool that assists in the generation of functional requirements and guides the user through a well defined process to derive an object oriented design. The tool explicitly exposes users to the central processes governing a formal software design methodology, alleviates much of the associated low level effort and ensures consistency between the requirements and the corresponding design. In addition to the design, the tool generates a detailed report on the derivation process and source code stubs for each class.

1. Background and Motivation

The Requirement Compiler (RC) tool is based on a rigorous methodology published in [4]. The methodology prescribes a sequence of actions to translate a set of functional requirements into an object-oriented (OO) design. Section 2 in this paper describes the methodology thoroughly. Conceptually, this translation can be compared to the function of a programming language compiler. That is, a compiler takes a program as input (i.e. source code) and translates the statements into machine code instructions. The Requirement Compiler similarly translates its input (functional requirements contained in a requirements document) into an OO design document. Figure 1 illustrates this concept.

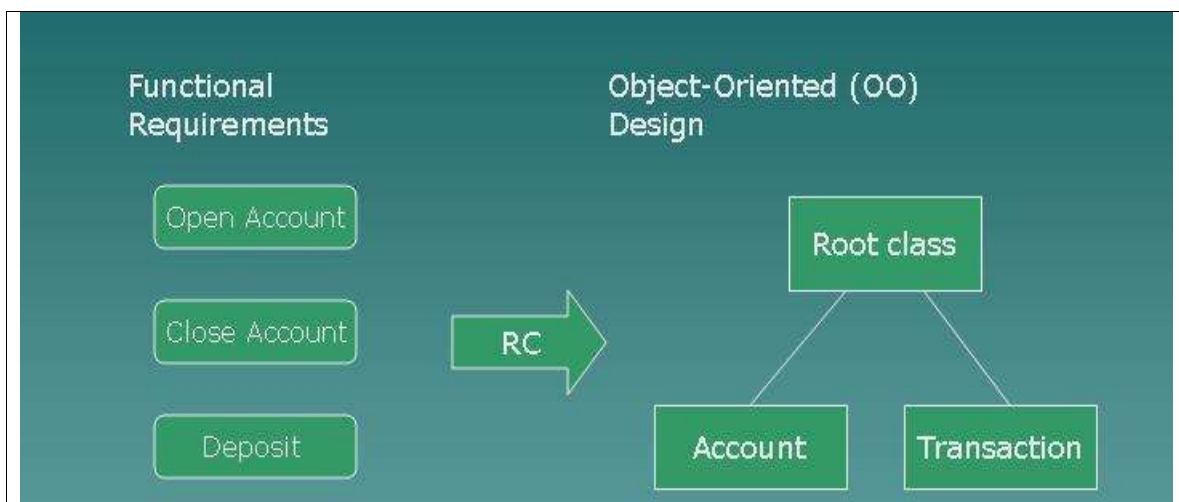


Figure 1: Conceptual model of Requirements Compiler

RC requires a specific syntax when parsing the requirements document, similar to a programming language compiler, and also generates error messages and a diagnostic report providing insight into the translation process.

The Requirements Compiler was developed to address difficulties in teaching core software engineering topics. The SE2004 [1] document was developed as a joint ACM/IEEE effort and serves as a guideline for curricular development in undergraduate software engineering. The topics addressed by this project are listed within two of the 10 knowledge areas cataloged by SE2004: software modeling and analysis (MAA) and software design (DES). RC is a software tool that assists in teaching core concepts in MAA and DES. RC provides support for authoring standards compliant requirement specifications and for transforming the elicited requirements into a coherent design. The methodology employed by RC intentionally exposes software designers to essential principles in software development while significantly reducing much of the low-level detailed

considerations necessary in the construction of quality requirement and design documents.

When learning fundamentals of requirements gathering and analysis, it is apparent that students often have difficulty in producing documentation that is consistent and standards compliant, even when a detailed template is provided. Students often become so enmeshed in low-level details such as correctly numbering sub-sections and choosing font styles that substantive issues are obscured. In addition, students are often resistant to following rigid documentation requirements such as those commonly practiced in industry. These problems are exacerbated when students are asked to produce design documents that are traceable to the requirements specification and which can be shown to consistently and coherently address all specified functionalities. RC was developed as a way of hiding the low-level clerical details of the software design process while emphasizing core principles.

RC will also have benefits to software engineering industry. It enables the enforcement of standard engineering processes (requirements writing, OO design generation, etc.) and relieving software engineers from the previously described low-level clerical details of constructing requirement and OO design documents. Also, standardization of requirement engineering (i.e. one tool for creating and editing requirements) will enable software developers to move between projects without learning new tools and techniques.

2. Methodology

Alagar and Periyasamy [4] developed a methodology to derive an object oriented (OO) design from a functional specification described using VDM-SL notation. Their methodology describes a formal process for deriving an OO design that is consistent with the functional requirements expressed in the specification. This method has limited pedagogical use since the functional requirements are expressed in a formal notation. The Requirements Compiler application has taken the methodology proposed by Alagar and Periyasamy and has extended the concepts to the analysis of informally documented functional requirements conforming to the IEEE 830-1998 standard [3]. The RC application assists in the creation of requirements specifications compliant with IEEE 830-1998 and interactively guides the subsequent derivation of an OO design.

2.1. IEEE 830-1998 Standard

The IEEE 830-1998 standard defines an organizational structure for a conformant requirements document, referred to as a Software Requirements Specification

(SRS). An IEEE 830-1998 conformant SRS consists of the sections listed in Figure 2. The introduction section provides an overview of the entire requirements document listing the name and purpose of the software to be produced, intended audience for the SRS and benefits/advantage of the software. The overview section provides more high level details about the software to be built. The product perspective describes how the software relates to other software systems (e.g. standalone or a component of larger system, etc.). The overview section also lists major product functions and a description of the characteristics of the intended users of the software. A list of assumptions and dependencies is also included, which is a list of factors that affect the SRS. For more details, refer to [3].

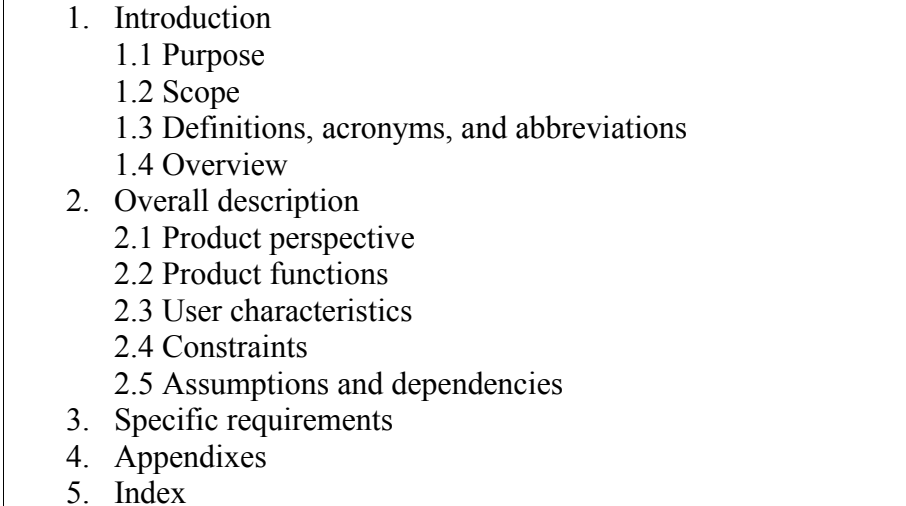
- 
1. Introduction
 - 1.1 Purpose
 - 1.2 Scope
 - 1.3 Definitions, acronyms, and abbreviations
 - 1.4 Overview
 2. Overall description
 - 2.1 Product perspective
 - 2.2 Product functions
 - 2.3 User characteristics
 - 2.4 Constraints
 - 2.5 Assumptions and dependencies
 3. Specific requirements
 4. Appendixes
 5. Index

Figure 2: Outline of IEEE 830-1998 SRS template.

The next major section of the SRS is Specific Requirements, providing a structured format to list the functions/features of the software system. The IEEE 830-1998 standard provides eight options to organize the requirements by mode, user class, objects (classes), features, stimulus, functional hierarchy or a combination of these styles. All the styles are very similar in structure, with the organization emphasizing the main goal of the project. For example, for a web-based user interface project, the SRS author may choose to organize the requirements by user class. Figure 3 displays the sections organized by mode.

3.	Specific requirements
3.1	External interface requirements
3.1.1	User interfaces
3.1.2	Hardware interfaces
3.1.3	Software interfaces
3.1.4	Communications interfaces
3.2	Functional requirements
3.2.1	Mode 1
3.2.1.1	Functional requirement 1.1
3.2.1. <i>n</i>	Functional requirement 1. <i>n</i>
3.2.2	Mode 2
3.2.3	Mode <i>m</i>
3.2. <i>m</i> .1	Functional requirement <i>m</i> .1
3.2. <i>m</i> . <i>n</i>	Functional requirement <i>m</i> . <i>n</i>
3.3	Performance requirements
3.4	Design constraints
3.5	Software system attributes
3.6	Other requirements

Figure 3: Specific requirements section organized by mode.

The Requirement Compiler mainly uses the assumptions section and the functional requirements section. The primary areas necessary for the translation process are functional requirements, listed in section three of an SRS. Each functional requirement is decomposed into such items as a unique index, name, purpose, set of input parameters, an output parameter, set of actions, set of exceptions that might occur during the realization of this requirement, set of remarks for the designers and finally a set of cross references. Figure 4 illustrates a sample requirement written in compliance with the IEEE specification.

Index:	ATM.2
Name:	Deposit
Purpose:	To deposit an amount into an account
Input parameters:	account number, amount
Output parameter:	None
Action:	Ensure that account number exists. Ensure that amount is greater than zero. Update the balance in the account by adding amount to it.
Exceptions:	account number does not exist. amount is less than or equal to zero.
Remarks:	None

Figure 4: An IEEE 830-1998 compliant requirement named “Deposit.”

RC supports semi-automated analysis of an IEEE 830-1998 compliant requirements specification to produce an OO design. RC parses all functional requirements (and the assumptions section) and guides the user through a series of

steps to derive an OO design. The following sections in this chapter will summarize this methodology.

2.2. Extract data objects

The first step in the methodology is to identify and extract data objects from the functional requirements and assumptions sections in the SRS. A data object corresponds to an entity that carries significant permanent data in the system; these items will exist as member variables in the software system. While data objects are found primarily in the “Input parameters”, “Output parameter” and “Assumptions” sections of the requirements document, the designer may add further data objects deemed to be necessary and appropriate. Table 1 provides an illustration of three functional requirements (left most column) and a set of data objects extracted, listed in the next column to the right. The data in this table will be used in the description of the remaining steps of the rigorous method.

2.3. Classify data objects as simple or composite

The next step is to categorize the extracted data objects as simple or composite; this is done by the designer intuitively based on their domain knowledge. Simple data objects are those that can be directly represented by well-defined programming language data types such as integer or double. All remaining data objects are classified as composite data objects, which will be represented as classes in the generated design. The middle column in Table 1 displays the category assigned to each data object. Composite data objects will later be transformed into classes, containing structural components (attributes) and behavioral components (methods).

Functional Requirements	Extracted Data Objects	Category	Standard Name	Data Type
Open Account	account file	simple	acctFile	string
	account entry	composite	Account	object
	first name	simple	fName	string
Deposit	amount	simple	Amount	double
	transaction entry	composite	Transaction	object
View Balance	account num	simple	acctNum	integer
	account no	simple	acctNum	integer

Table 1: Data objects and their standard name, category and data type.

As part of this step, a standard name is assigned to each data object. The standard name consolidates data objects that represent similar entities, but exist in the requirement document as differently spelled items. The standardized name of “acctNum” has been assigned to data objects “account num” and “account no” (refer to functional requirement View Balance in Table 1). It is common for such

small differences to occur in a requirements document due the fact it is written in natural language (e.g., English) and there are often multiple individuals contributing to the effort of writing a functional requirements document, introducing the possibility of inconsistencies in naming. Lastly, a data type is identified for each simple data object. The standard name and data type will be carried forward to the resulting OO design as attributes in a class.

2.4. Define structure of each composite data object

The structural components (attributes) are selected for each class from the set of extracted simple and composite data objects, but the designer also has the freedom to introduce new data objects into the design. In this case, RC will guide the user through a series of steps to update the functional requirement accordingly, thereby ensuring consistency between the requirements and design; that is, all attributes are based on data objects described in the functional requirements. The rigorous method also introduces another class at this time that corresponds to the main program where program execution will begin, a class referred to as Root in Table 2. Table 2 lists all classes and attributes assigned to each class. Once again, the methodology relies upon the domain knowledge and expertise of the designer to decide which data object should be assigned to each class. The classes below are simplified to maximize clarity for these examples.

Classes (Composite Data Objects)	Attributes (Standard Name)	Data Type
Account	fName acctNum	String Integer
Transaction	Amount	Integer
Root	acctFile Account	String object

Table 2: Classes (composite data objects) with user-assigned attributes.

2.5. Distribute functional requirements to classes (as methods)

The next step automatically distributes the functional requirements to the available classes (as methods) based upon a set of distribution rules. The simplest case occurs when a data object in *one* functional requirement is mapped to an attribute in *one* class. This example is illustrated by the functional requirement View Balance in Figure 5; the data object "acctNum" is assigned as an attribute in the class Account. The presence of the data object implies that this functional requirement will be transformed as a method. The functional requirement View Balance is transformed to method viewBalance() in the class Account. It is important to note that the entire functional requirement View Balance will be

implemented by this method, and be contained in only one class. This type of distribution is the least common.

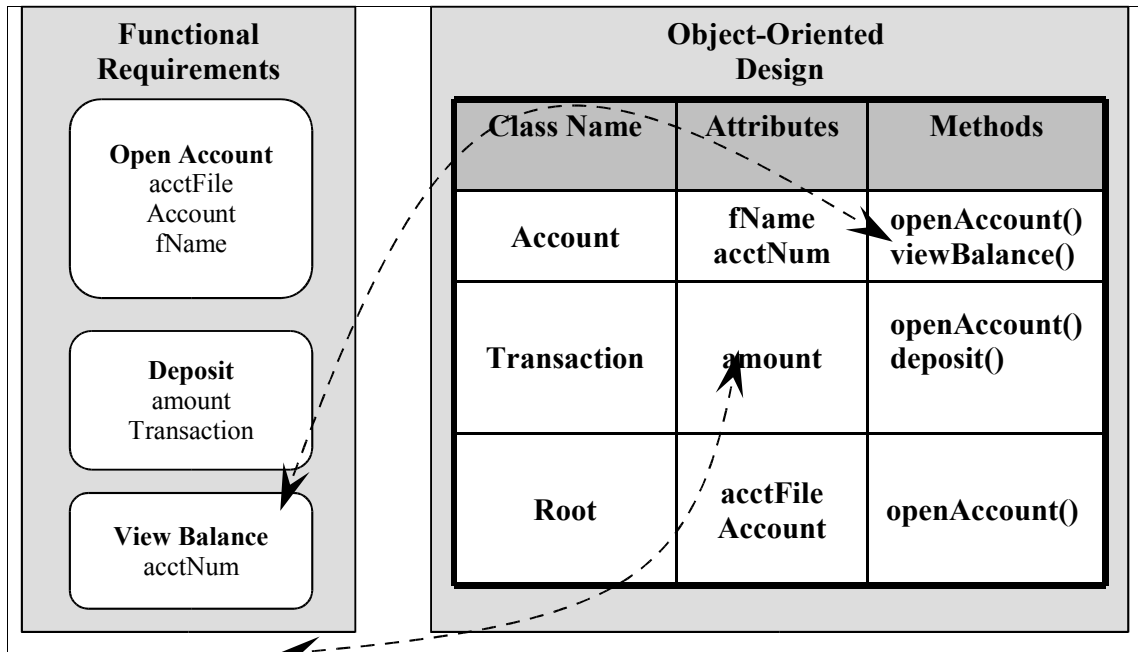
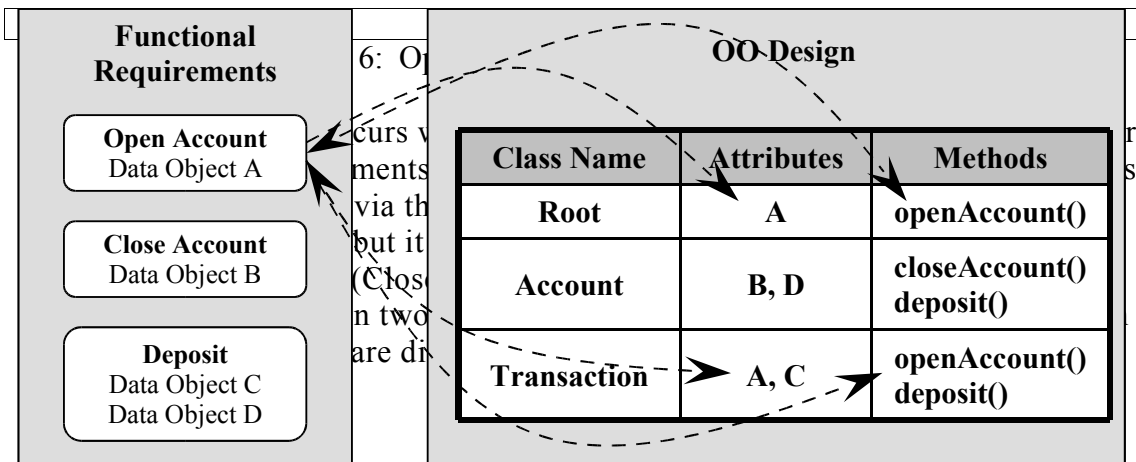


Figure 5: View Balance is distributed to class Account.

A more common case occurs when a data object in *one* functional requirement is distributed to *many* (two or more) classes. In this case, the functional requirement is distributed to multiple classes – that is, the methods to implement the functional requirement are distributed across multiple classes. This scenario is illustrated symbolically via the data object “A” – the arrows in Figure 6 indicate the mapping.



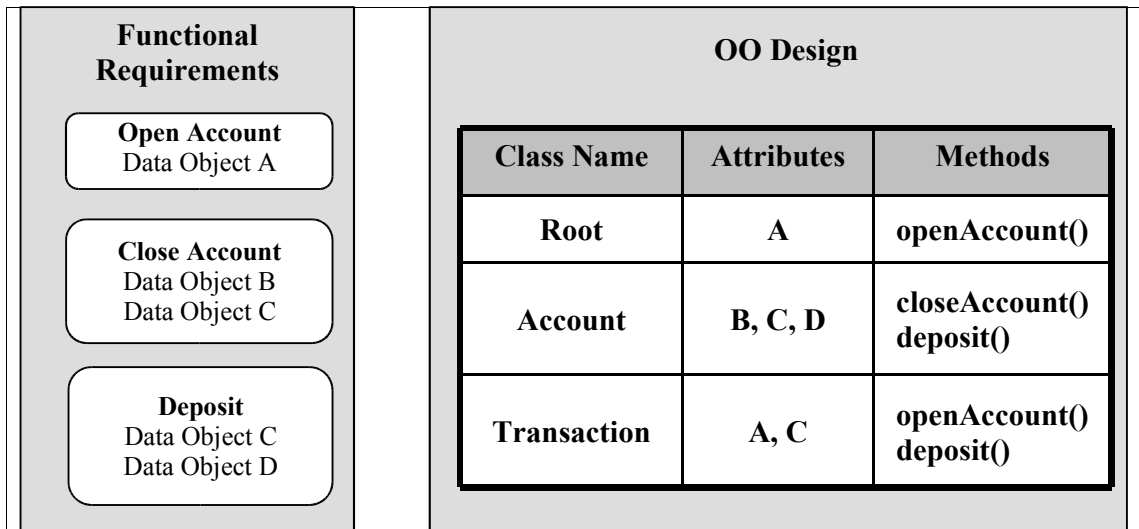


Figure 7: “Close Account” and “Deposit” mapped to classes Account, Transaction.

The justification for these rules comes from the fact that all data objects are manipulated in the same manner both in the functional requirement and in the design. The behavior of these objects in the design is thus consistent with their behaviors as stated in the functional requirements. Comparable approaches are described in [2, 5, 6].

2.6. Identify relationships

Finally, the designer is required to identify the relationships among the classes. The rigorous method automatically identifies aggregation relationships within a class based on the presence of composite data objects (i.e. classes) assigned as attributes. The designer is responsible to identify additional relationships.

These five steps result in an OO design implementing the functionalities enumerated in the requirements document. The designer has the freedom to edit this design or augment it with more detailed information. The five step process *ensures* that the functional requirements are represented in the object oriented design and the designer is guided through this structured process to think about how the requirements map to the design.

3. Requirements Compiler Overview

Figure 8 shows a schematic overview of the RC architecture. The Requirements Editor is a structured editor, guaranteeing that the resulting document is compliant with IEEE 830-1998. The editor performs many consistency checks, warning users of incomplete entries, ensuring that all functional requirements have unique identifiers and identifying ambiguous words.

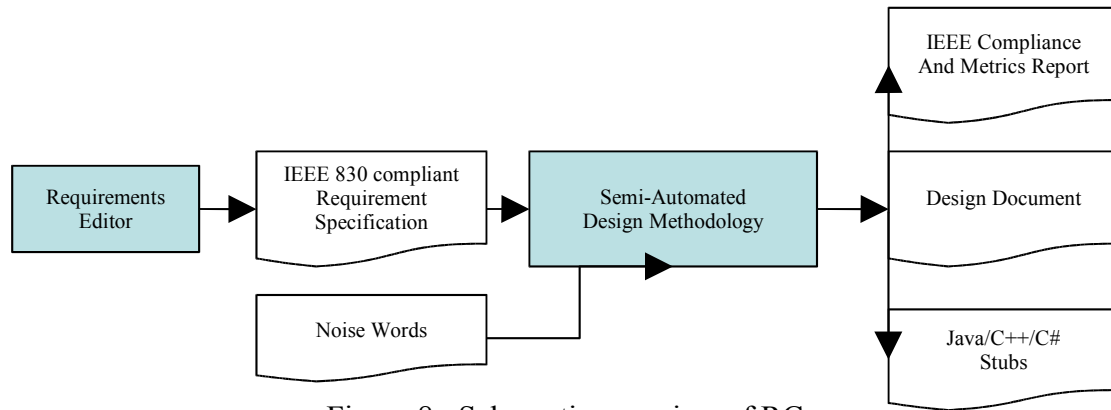


Figure 8: Schematic overview of RC.

RC then interactively guides the user in the derivation of an OO design by implementing the five steps described in the previous section. The OO design includes a set of classes, their structural components (attributes) and methods described in an ad-hoc HTML format. In addition, RC also generates source code stubs for each class in Java, C# or C++.

RC also generates a report for refining the requirements. This includes such items as a listing of all functional requirements that are not implemented in the design and a listing of classes and methods that are not traceable to a particular requirement. In addition, the application logs all designer actions that were performed during the compilation. Since such design decisions are made explicit, the review and modification of those design decisions is simplified.

Figure 9 shows the graphical user interface (GUI) for RC. It consists of two primary panes. The left pane enumerates the steps of the derivation process and changes dynamically in correspondence to the selected step. The right pane displays the functional requirements in a properly formatted fashion where particular words are color coded to allow the designer to quickly identify noise words (described in section 4.2), data objects and the like during the development process.

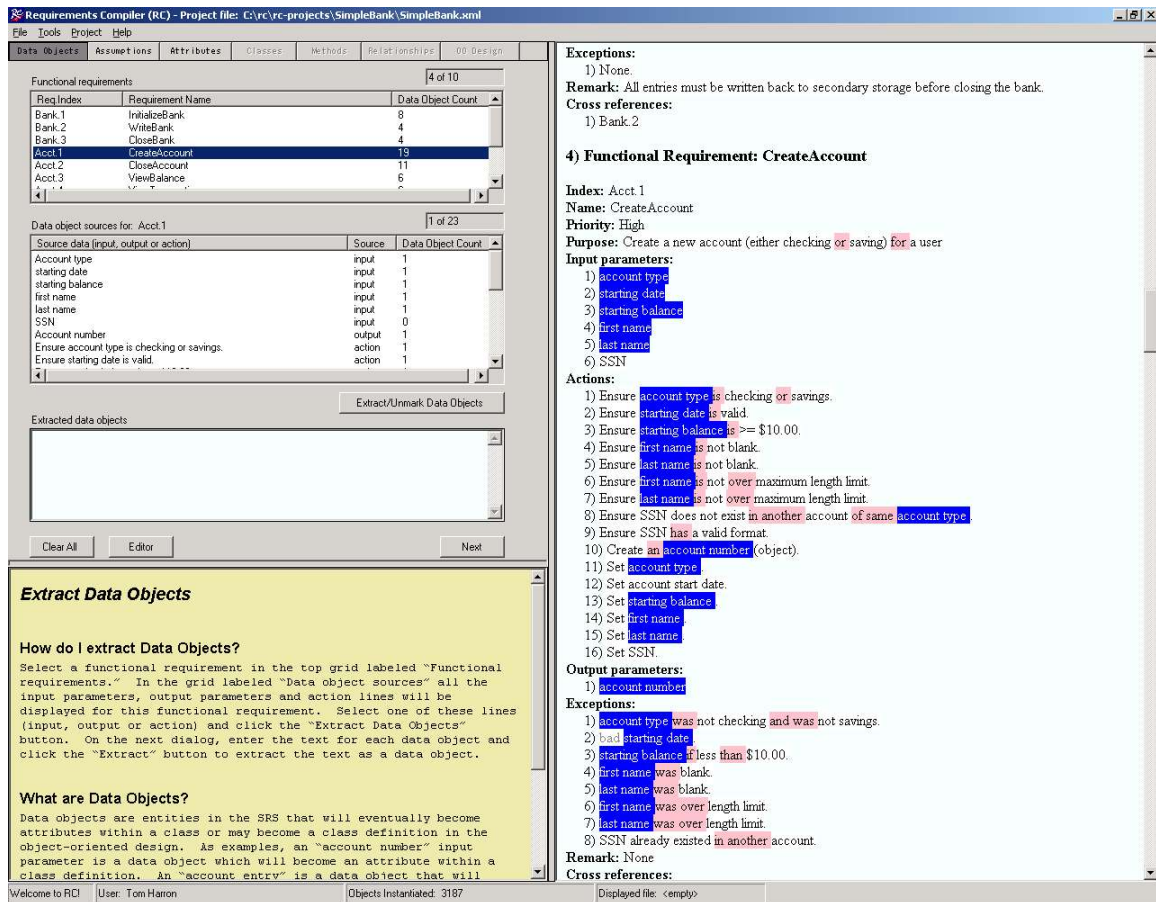


Figure 9: Screenshot of the RC software showing the Requirements Specification.

4. Scenario Walkthrough

This section will provide a high-level walk-through of the application. Due to space limitations, only the central aspects of each phase of design are highlighted here.

4.1. Create a requirements document

The designer must initially create a requirements document by using the supplied structured editor. The editor *ensures* that the requirements document structurally conforms to IEEE 830-1998. Since the RC application is able to generate actual code stubs from the eventual derived design, the designer must also select a target language. Languages currently supported include Java, C# and C++.

4.2. Extract data objects

Once the requirement document has been created, the user is ready to perform the translation from requirements document to object-oriented design. The first step in the process is to extract data objects in order to create a data dictionary that is drawn from the requirements specification. Figure 10 displays the extraction tool listing an entry from an input parameter from a particular functional requirement. As can be seen, the input into this tool is natural language (e.g. English); the user is responsible to identify data objects such as “amount” and “transaction type” in the figure below by highlighting and extracting them.

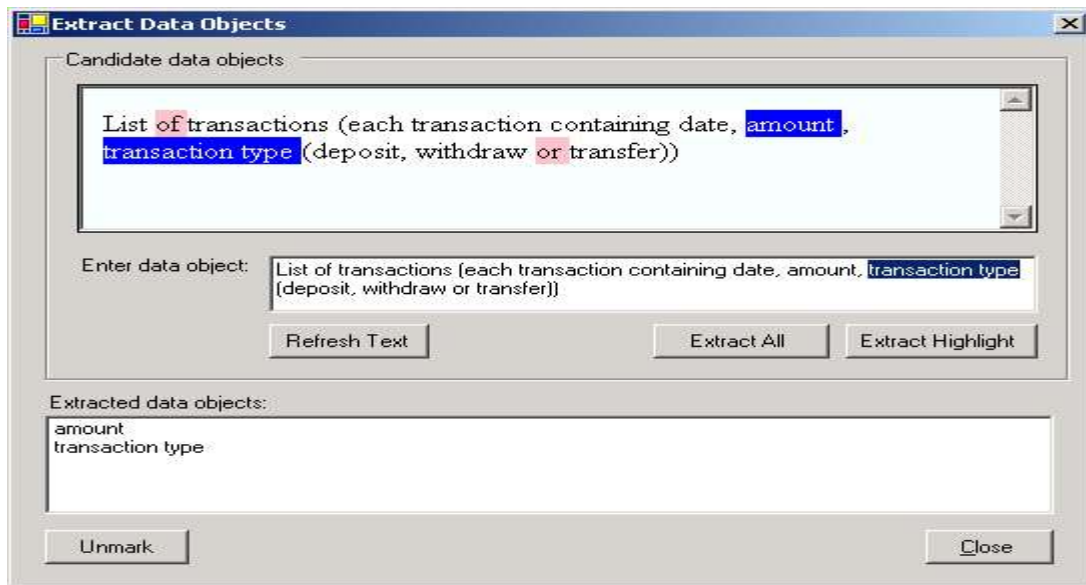


Figure 10: Data object extraction tool, showing highlighted data objects

After the user extracts a particular data object (i.e. “amount”), RC will automatically extract all other occurrences found in the document in every other functional requirement and/or assumptions.

4.3. Classify Data Objects

Once all data objects are extracted, they must be manually classified as simple or composite. When a data object is classified as *simple*, the designer identifies the underlying data type to represent the data object. RC provides a list of simple data types such as integer, double, and string depending on the target language that was selected as described in 4.1. When the designer identifies a data object as *composite*, RC creates a corresponding class definition that will be further refined in subsequent steps. This step is completely manual since the semantics of the problem domain cannot generally be embedded in the RC application.

Often, the same data object may be represented using different names within a requirements document. For example, the data object ‘start date’ might have been

included in “Input parameters” and referred as ‘starting date’ in an “Action” section. In this case, the designer should resolve the inconsistency by selecting a single phrase that will be uniformly applied throughout the requirements and design documents.

4.4. Identify the attributes for each class

This step allows the designer to manually define the structural components (attributes) of each class. Attributes are selected from the data dictionary. The designer selects a class and then assigns selected data objects as attributes of that class. It is possible that a data object may be assigned as an attribute of more than one class.

4.5. Identify the methods for each class

RC provides support for distributing requirements across class methods as described in Section 2. RC again provides low-level scribal support for this process by ensuring that all functional requirements are implemented by at least one method and that the input and output parameters are type-consistent.

After transforming a functional requirement into a method as described above, the designer may need to adjust each method depending on the context. This is because only a subset of actions in the functional requirement may correspond to the actions to be performed by the method. To illustrate this adjustment, consider a functional requirement called “verify a transaction” that is transformed into a method of the class “Account.” The only action required in the method may be to validate the account number. The functional requirement may list other actions such as verify user id, verify the transaction amount and so on. Therefore, the method name could be changed from “verify a transaction” to “validate account.” RC allows methods to be renamed while maintaining traceability and consistency among the various documents.

4.6. Defining relationships among classes

Classes may be related to each other in complex ways via aggregation, association or specialization. RC *automatically* detects aggregation relationships among classes based on the assignment of composite data objects as class attributes. The designer must *manually* identify associations and specializations.

4.7. Generation (design, code, reports)

RC is able to generate a design document, source code stubs and a report summarizing the consistency between the requirements specification and the derived design.

4.7.1. Design document

RC will generate a design document containing the definition of all classes, including their attributes and methods. Attributes are specified in terms of the underlying implementation choices and include access control modifiers. Method declarations for each class include the return type, method name and input arguments. The body of the method will contain comments based on the “Action” clauses from functional requirements. The relationships described in Section 2.6 are also included in the design document.

4.7.2. Code generation

RC generates source code for the initial design. The source code provides a starting point for the designer to begin implementation, taking advantage of the design decisions made while using RC. Much of the information contained in the design document is rendered in the target language chosen at project inception. Method bodies, however, consist of comments drawn from the various functional requirement clauses and indicate the expected behavior of the method.

4.7.3. Report generation

RC is able to provide information and metrics on the derived design by generating a diagnostic report. The report provides information about a requirement document’s compliance to IEEE standards and overall quality. RC will report on duplicate requirement names, cross-reference errors, missing fields within a requirement (e.g., no “Actions” field for a particular requirement) or blank fields (e.g., the “Output Parameter” field is blank). Other quality checks consist of ensuring that data objects are used appropriately within a functional requirement. If, for example, a data object exists as an “Input Parameter” of a functional requirement, that data object must be referenced by some “action” or “exception” in the same functional requirement.

Secondly, the report provides metrics derived from an analysis of the functional requirements and design documents. The report includes information such as total number of: functional requirements; input and output parameters per requirement; actions per requirement; data objects associated with a functional requirement; and others. In particular the relationships between design and requirements are explicitly maintained and therefore traceable.

5. Conclusion

This paper has described the basic features of a software environment that supports the derivation of an OO design by creating and analyzing a functional requirements specification. The RC application derives an OO design from the functional requirements using a step-by-step process that is highly visible to the designer and explicitly enforced and supported by the software environment. A log of each design decision is recorded for later review and modification and serves to emphasize the important technical decisions inherent in an OOD process. Moreover, the designer can visualize the relationships between the entities in the requirements and those in the design thus providing traceability of requirements from design. Since the software is interactive it enables the designer to dynamically reconfigure the design and ensure consistency between the design and requirements.

It is the author's belief that classroom use of such an application will 1) teach common fundamental aspects of OO design 2) teach the importance of structured requirement specifications, and 3) alleviate much of the burdensome work required for deriving a consistent OO design from a set of procedural requirements.

References

- [1] Joint Task Force on Computing Curricula, "Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering", 2004.
- [2] S. Liu and N. Wilde, "Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery", *Proceedings of IEEE Conference on Software Maintenance*, San Diego, 1990, pp. 266-271.
- [3] Recommended Practice for Software Requirements Specification, *IEEE Standard 830-1998*, IEEE Computer Society Press, 1998.
- [4] V.S. Alagar and K. Periyasamy, "A Methodology for Deriving an Object-Oriented Design from Functional Specifications", *Software Engineering Journal*, Vol. 7, No. 4, July 1992, pp. 247-263.
- [5] Cimitile et al., "Identifying Objects in Legacy Systems", *5th International Workshop on Program Comprehension*, Dearborn, MI, 1997, pp. 138-147.
- [5] J. George and B.D. Carter, "A Strategy for Mapping from Function-Oriented Software Models to Object-Oriented Software Models", *ACM Software Engineering Notes*, Vol. 21, No. 2, March 1996, pp. 56-63.
- [6] J. George and B.D. Carter, "A Strategy for Mapping from Function-Oriented Software Models to Object-Oriented Software Models", *ACM Software Engineering Notes*, Vol. 21, No. 2, March 1996, pp. 56-63.