Requirements Compiler

A Manuscript

Submitted to

the Department of Computer Science

and the Faculty of the

University of Wisconsin-La Crosse

La Crosse, Wisconsin

by

Thomas W. Harron

in Partial Fulfillment of the

Requirements for the Degree of

Master of Software Engineering

May, 2006

REQUIREMENTS COMPILER

By Thomas W. Harron

We recommend acceptance of this manuscript in partial fulfillment of this candidate's requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

_____	_____
Dr. Kasi Periyasamy Examination Committee Chairperson	Date
_____	_____
Dr. Kenny Hunt Examination Committee Member	Date
_____	_____
Dr. David Riley Examination Committee Member	Date
_____	_____
Dr. Vijendra Agarwal Director, University Graduate Studies	Date

ABSTRACT

Harron, Thomas, W., “Requirements Compiler”, Master of Software Engineering, May 2006, Advisors: Dr. Kasi Periyasamy, Dr. Kenny Hunt.

The process of developing an object-oriented software design from a functional requirements specification is a fundamental issue in software engineering. Important issues such as discovering data elements present in the requirements, deriving design objects, specifying object relationships, and others can be obscured as significant low-level clerical effort is expended to produce consistent, traceable, standards-compliant requirements and design documentation. This manuscript describes a software tool, Requirements Compiler (RC), which assists in the generation of quality functional requirements and guides the user through a well defined process to derive an object-based design. The tool explicitly exposes the user to central processes governing a formal software design methodology, alleviates much of the associated low level clerical effort and ensures consistency between the requirements and the corresponding design. RC provides an editor for composing and editing IEEE-compliant functional requirement documents, ensuring the requirements specified are structured correctly (i.e. syntax). RC also provides an interactive GUI environment which guides the user through a derivation process to obtain an object-based design for a selected requirements document. In addition to the generated design, the tool provides a detailed statistical report on the derivation process and source code stubs for each class specified in the design. The generated source code (Java, C# or C++) contains declarations for the attributes and signatures and pseudo-code for the methods, all based upon information parsed from the requirements document.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my project advisors Dr. Kasi Periyasamy and Dr. Kenny Hunt for their valuable guidance. I spent many hours with Kasi discussing design decisions regarding the rigorous method and clarifying project priorities and scope; Dr. Hunt provided feedback on the existing design and offered many elegant alternatives, especially leveraging the power of XML. Kasi and Kenny encouraged me to write papers for publication and spent numerous hours coaching me in this process and provided invaluable feedback; I thank them for their time and expertise. I would like to thank David Dobson for taking the time to meet regularly and provide valuable feedback during the project. I would also like to express my thanks to the Computer Science Department, Murphy Library and the University of Wisconsin-La Crosse for providing the computing environment for my project. Finally, I wish to thank my wife Heide and my daughters Natalie and Olivia for their patience and encouragement during the tenure of this project. I truly appreciate the sacrifices they made to enable me to complete this degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

Aggregation

Also called a “has a” relationship, aggregation between two classes defines that an object of one class includes an object of the other class.

Attribute

An attribute is a structural component of a class**.**

DFD

Data Flow Diagram is a graphic that illustrates the movement of data between external entities and the processes and data stores within a system.

Data Object

A data object corresponds to an entity that carries significant permanent data in the system. A data object will likely be represented as an attribute in the implementation of a software system.

Entity-Relationship Diagram

Entity-Relationship Diagram is a diagram that depicts an entity relationship model’s entities, attributes, and relations. It also displays connectivity and cardinality.

Functional Requirements

These are statements of services the system should provide, how the system should react to particular set of inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.

HTML

HyperText Markup Language. A markup language designed for creating web pages and other information to view in a web browser.

IEEE

An acronym for Institute of Electrical and Electronics Engineers, Inc., which is an international organization whose constitution describes their purpose as "scientific and educational, directed toward the advancement of the theory and practice of several engineering fields including computer science."

IEEE 830-1998 Standard

This is a recommended practice describing approaches for the specification of software requirements. It emphasizes the development of unambiguous and complete requirements specification documents.

Method

Used mainly in object-oriented programming, this term refers to a piece of code that is exclusively associated either with a class (called class methods or static methods) or with an object (called instance methods). Like a procedure in procedural programming languages, a method usually consists of a sequence of statements to perform an action, a set of input parameters, and possibly an output value (called return value) of some kind.

Object-based design

A limited version of object-oriented design where one or more of the following restrictions apply: there is no implicit inheritance or there is no polymorphism.

Object-oriented design

A design method in which a system is modeled as a collection of cooperating objects and individual object are treated as instances of a class within a class hierarchy.

PDF

An Adobe technology representation for formatting documents so that they can be viewed and printed using the Adobe Acrobat reader.

RC

Requirements Compiler, which is the name of software tool described in this report.

SRS

Software Requirements Specification is a document format supplied by the IEEE 830-1998 Standard for specifying the requirements of a software system.

UML

Unified Modeling Language™ (UML) is an industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems standardized by the Object Management Group. UML simplifies the complex process of software design by using “blueprints” for software construction.

XML

XML (Extensible Markup Language) is a W3C initiative that allows information and services to be encoded with meaningful structure and semantics which computers and humans can understand. XML is great for information exchange, and can easily be extended to include user-specified and industry-specified tags.

1. Introduction

The Requirement Compiler (RC) tool is based on a rigorous methodology published in [14]. The methodology prescribes a sequence of actions to translate a set of functional requirements into an object-based design. Section 2 of this report will describe the methodology thoroughly. Conceptually, this translation can be compared to the function of a programming language compiler [10]. That is, a compiler takes an 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 object-based design document. Figure 1.1 illustrates this concept.

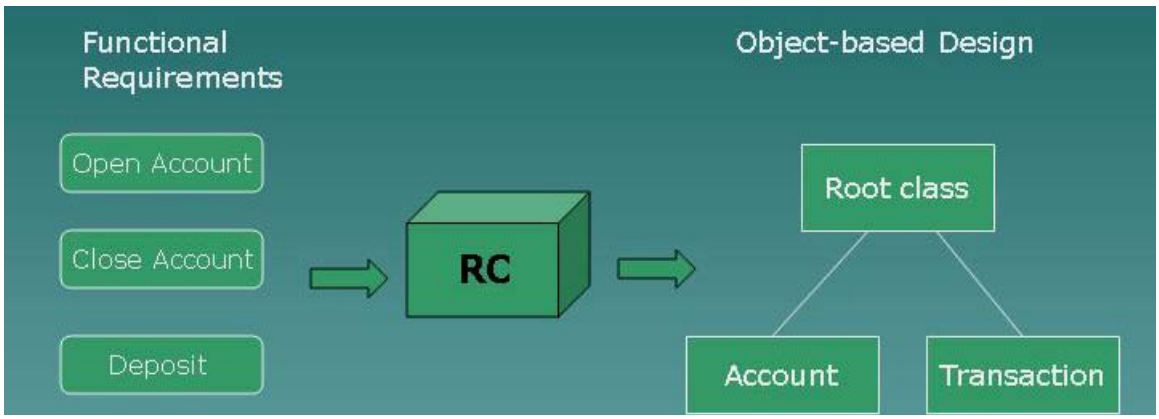


Figure 1.1. Conceptual model of Requirements Compiler

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

The design derived from this methodology is object-based, meaning it employs a subset of possible object-oriented design concepts. The object-based design is written to an "object-oriented design document", with the expectation that the designer will extend

the design with the use of encapsulation, inheritance, polymorphism and/or other object-oriented design concepts.

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 teaching 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 in the software engineering industry. The tool has value from a perspective of enforcing software engineering principles (requirements

writing, object-based design generation, etc.) and relieving software engineers from the previously described low-level clerical details of constructing requirement and design documents. Also, enforcing a rigorous requirement engineering process (i.e. one tool for creating and editing requirements) will enable software developers to move between projects without learning new tools and techniques. The object-based design derived via RC will map back to the requirements document providing testers with a skeleton traceability matrix also.

2. Rigorous Methodology

Alagar and Periyasamy [14] developed a methodology to derive an object-based design from a functional specification described using VDM-SL notation. Their methodology describes a formal process for deriving a design that is consistent with the functional requirements expressed in the formal specification. This method has limited pedagogical use since the functional requirements are expressed in a formal notation. The Requirements Compiler (RC) 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 [7]. RC also assists in the creation of requirements specifications compliant with the IEEE 830-1998 standard and interactively guides the user through the derivation process.

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 2.1. 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 on the introduction and overall description sections, refer to [7].

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 2.1. Outline of IEEE 830-1998 SRS template

The next major section of the SRS describes the details of the functional 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 2.2 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.*n* Functional requirement 1.*n*
  - 3.2.2 Mode 2
  - 3.2.3 Mode *m*
    - 3.2.*m*.1 Functional requirement *m*.1
    - 3.2.*m.n* Functional requirement *m.n*
- 3.3 Performance requirements
- 3.4 Design constraints
- 3.5 Software system attributes
- 3.6 Other requirements

Figure 2.2. Specific requirements section organized by mode

Performance considerations and other non-functional requirements such as maintainability, security, reliability and portability can be listed in the software system attributes.

The Requirement Compiler makes use of (i.e. parses) the information described in the functional requirements and the list of assumptions contained within the requirements document. The necessary inputs to the translation process are the functional requirements, listed in section 3 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 2.3 illustrates a simplified version of 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 2.3. An IEEE 830-1998 compliant requirement named “Deposit”

RC supports semi-automated analysis of an IEEE 830-1998 compliant requirements specification to produce an object-based design. RC parses all functional requirements (and the assumptions section) and guides the user through a series of steps to derive a 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 may later be transformed 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 2.1 provides an illustration of three functional requirements (leftmost column) and a set of data object 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 2.1 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 account entry first name	simple composite simple	acctFile Account fName	string object string
Deposit	amount transaction entry	simple composite	amount Transaction	double object
View Balance	account num account no	simple simple	acctNum acctNum	integer integer

Table 2.1. Data objects and their category, standard name and data type

As part of this step, a standardized name is assigned to each data object. The standardized name consolidates data objects that represent the same entity, but

spelled differently in other portions of the requirement. As an example, in Table 2.1, the standardized name of “acctNum” has been assigned to the data objects “account num” and “account no” (refer to functional requirement View Balance in Table 2.1). It is common for such small differences to occur in a requirements document due to the fact that it is written in natural language (e.g., English). In addition, there are often multiple individuals contributing to the effort of writing a functional requirements document, introducing the possibility of additional inconsistencies in naming. Lastly, a data type is identified for each simple data object. The standardized name and data type will be carried forward to the resulting design.

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 2.2. Table 2.2 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(s) 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 2.2. 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 as an attribute in **one** class. This example is illustrated by the functional requirement View Balance in Figure 2.4; the data object "acctNum" is assigned as an attribute in the class Account. As defined by the rigorous method, a mapping takes place when a data object of a functional requirement is assigned as an attribute of a class. The methodology asserts that this mapping implies that the functional requirement will be implemented by the mapped class(es). In this example, 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.

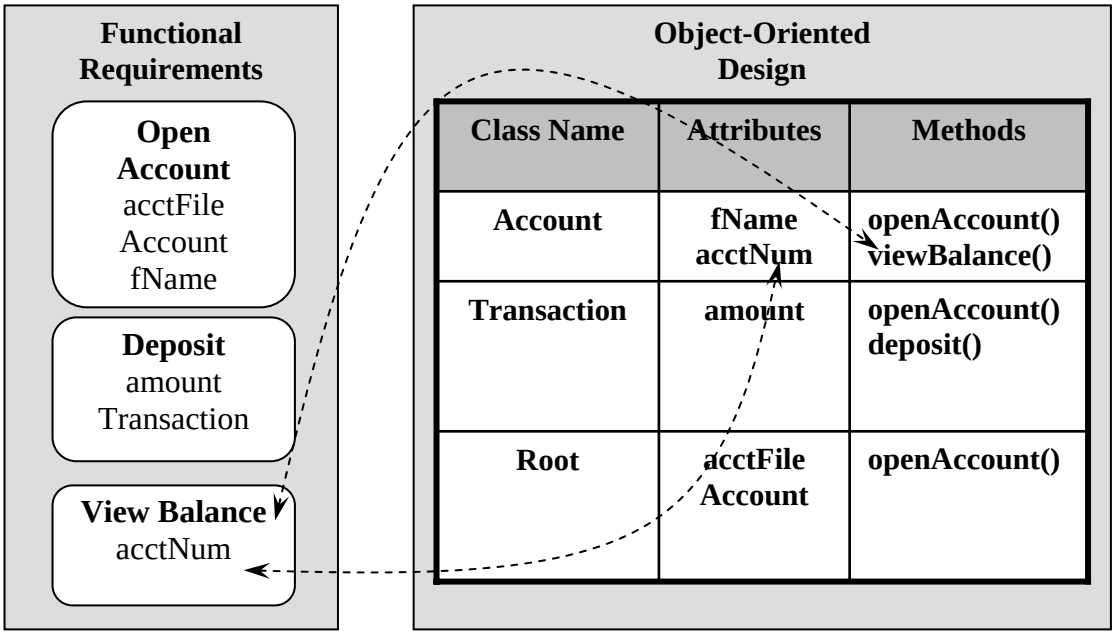


Figure 2.4. View Balance is distributed to class Account

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" in Figure 2.5; the arrows indicate the mapping.

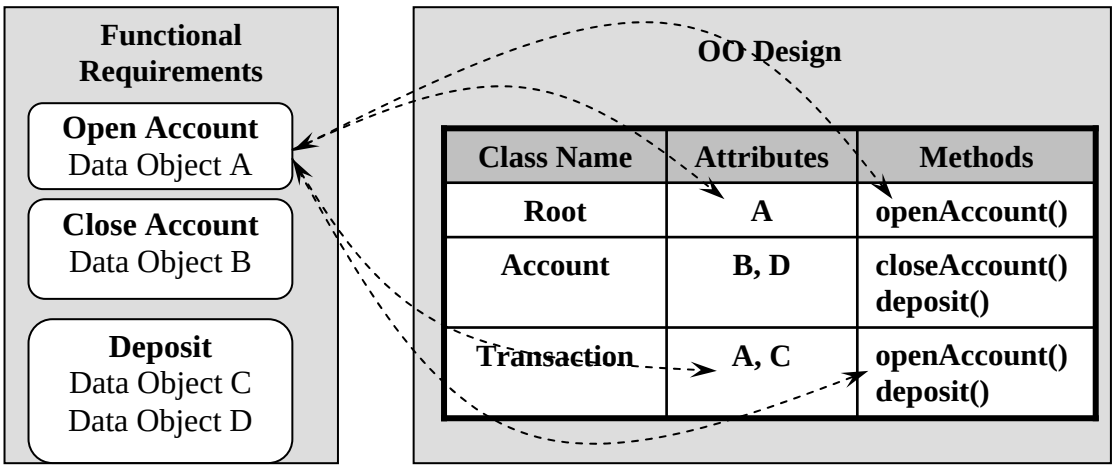
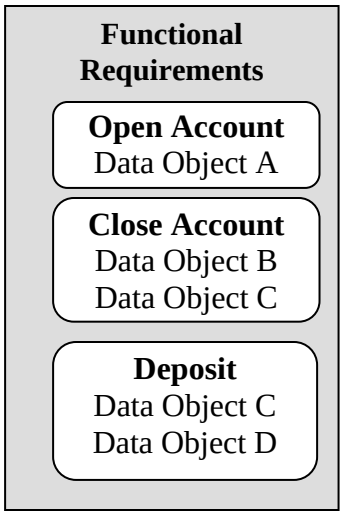


Figure 2.5. Open Account distributed to two classes

A more complex case occurs when the same data object is present in **many** (two or more) functional requirements and is mapped to **many** (two or more) classes. This is illustrated in Figure 2.6 via the data object "C." Arrows are not drawn to avoid complicating the figure, but it can be seen that the data object "C" is present in two functional requirements (Close Account and Deposit) and that the data object is assigned as an attribute in two classes (Account and Transaction) – therefore both functional requirements are distributed to both classes. This is also a common case.



OO Design

Class Name	Attributes	Methods
Root	**A**	**openAccount()**
Account	**B, C, D**	**closeAccount()** **deposit()**
Transaction	**A, C**	**openAccount()** **deposit()**

Figure 2.6. "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 [3, 9, 13]. Liu and Wilde [13] explain two techniques to identify objects in procedural programming languages such as C or Pascal. The first method identifies objects by locating all functions that reference global data definitions. An object consists of the global data definition and its related functions. The second method involves grouping functions that share either argument data types and/or return types in the function signature. While these methods are not searching functional requirements for data objects, the approach of

finding data objects (global data definition) and its associated actions (functions) is similar to the rigorous method. Cimitile [9] explains another approach for recovering an object-oriented decomposition from a procedurally oriented legacy system. This approach identifies objects (classes) by locating data stores such as files and tables in databases. Object methods are derived from program and subroutines. A relationship between the objects and methods are formed by the number of times each program and/or subroutine accesses the data store. Finally, George and Carter [9] discuss a mapping strategy for identifying objects based a function-oriented model described by a data flow diagram (DFD), an entity-relationship diagram (ERD) and an accompanying data dictionary. The authors describe six phases resulting in an object structure and message diagram. This strategy most closely matches the concept of the rigorous method described in this manuscript. The major difference is that the Requirements Compiler takes as input an IEEE-compliant functional requirement document (an SRS) instead of the combination of the DFD, ERD and data dictionary.

Figure 2.7 illustrates a more detailed view of the mapping process. The functional requirement “Initialize Bank” has been mapped to the method name “initializeBank()” in the class Account. This mapping was done due to the fact the data object “account file name” exists both in the functional requirement and as an attribute of class “Account.” The attribute name “m_acctFile” was specified as the standardized data object name by the designer as part of the derivation process as described in section 2.3. The input parameter(s) (in the functional requirement) becomes an argument in the method signature; the data type “string” was assigned by the designer during the methodology (also, described in section 2.3). The two action lines and one exception line are mapped into the method body as pseudo-code due to the presence of the data object “account file name” and finally, the output parameter is mapped to the method’s return type. In this case, there is no output parameter specified in the functional requirement, resulting in a “void” return type. If a data object had been specified, the data type assigned by the designer (as described in section 2.3) would have been used as the method’s return type.

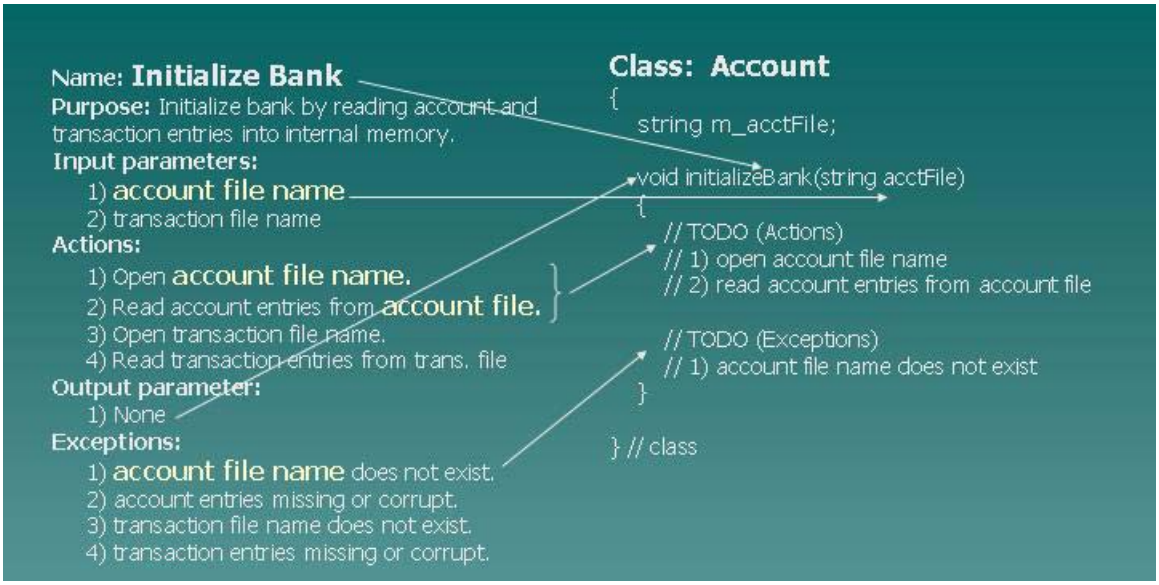


Figure 2.7. Detailed view of a functional requirement mapped to a method

At this time, it is advisable to refine the method names in each class. The derivation process mechanically transforms the functional requirement names to method names but only a portion of the functional requirement may be implemented in this method. It is common to refine the method names based on the actions included as part of this functional requirement. Figure 2.7 illustrates that only two of four action lines (and only one of four exception lines) are mapped to the method initializeBank(). The designer would likely refine the method name from initializeBank() to initializeAccountFile() based on the information mapped from the functional requirement document. It is expected the remaining action and exception lines will be mapped to a different method (and possibly in different classes) depending upon where the data objects transaction file name, account entries and transaction entries are mapped. A point worth observing is that functional requirements are often distributed to multiple methods and/or classes; it is uncommon for one functional requirement to be entirely implemented as one method within one class.

2.6. Identify relationships

Finally, the designer is required to identify the relationships among the classes. These relationships fall into three categories – association, aggregation and specialization. The rigorous method records 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 the other two types of relationships since the information corresponding to these relationships is not generally found in the requirements document. The designer will be responsible to identify additional aggregation relationships or change them to association relationships.

These five steps result in an object-based design implementing the functionalities described 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-based design and the designer is guided through this structured process to think about how the requirements map to the design.

Class Name	**Attributes**	**Methods**
Account	string m_fName int m_acctNum	openAccount() viewBalance()
Transaction	double m_acctNum	openAccount() deposit()
Root	string m_acctFile Account m_Account	openAccount()

Table 2.3. Object-based design

3. Requirements Compiler Overview

Figure 3 shows a schematic overview of RC architecture. The Requirements Editor is a structured editor, guaranteeing that the resulting document is compliant with the IEEE 830-1998 standard. The editor performs many consistency checks, warning users of incomplete entries, ensuring that all functional requirements have unique identifiers and identifying ambiguous words. The editor will be described in more detail in section 4.3.

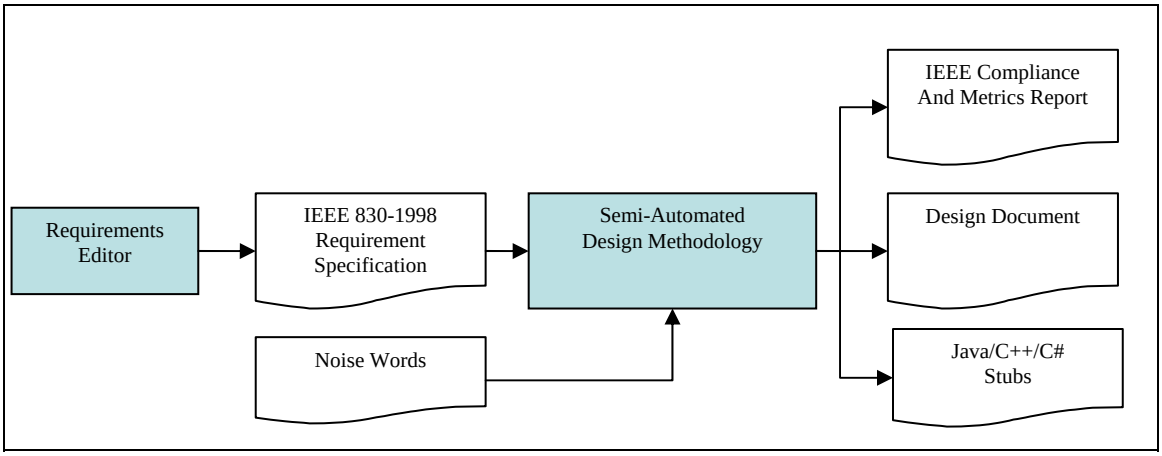


Figure 3.1. Schematic overview of RC

RC then interactively guides the user in the derivation process by implementing the five steps described in the previous section. The object-based design includes a set of classes, their structural components (attributes) and behavioral components (methods) generated in an ad-hoc HTML format [6]. In addition, RC also generates source code stubs for each class in Java, C++ or C#. The source code generated is formatted according to the syntax rules for each programming language and can be loaded into its respective development environment for compilation and subsequent development efforts.

RC also generates a report providing diagnostic information captured during the design derivation process. 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. The report provides statistics to the user in many perspectives: data object, functional requirements, classes, methods, etc.

Figure 3.2 shows the graphical user interface (GUI) for RC; it consists of three primary panes. The upper left pane enumerates the steps of the derivation process and changes dynamically in correspondence to the selected step. The steps are listed across the top of this pane (row of buttons): Data Objects, Assumptions, Attributes, Classes, Methods, Relationships and lastly, Design. These steps match the steps of the rigorous method described in the previous section. The lower left pane provides online context-sensitive help; as the user navigates through the software, this pane will dynamically update information displayed to match the activity the user is currently performing. The right pane displays the functional requirements in a readable format where particular words are color coded to enable the designer to quickly identify data objects (described in section 4.4), noise words and the like during the development process.

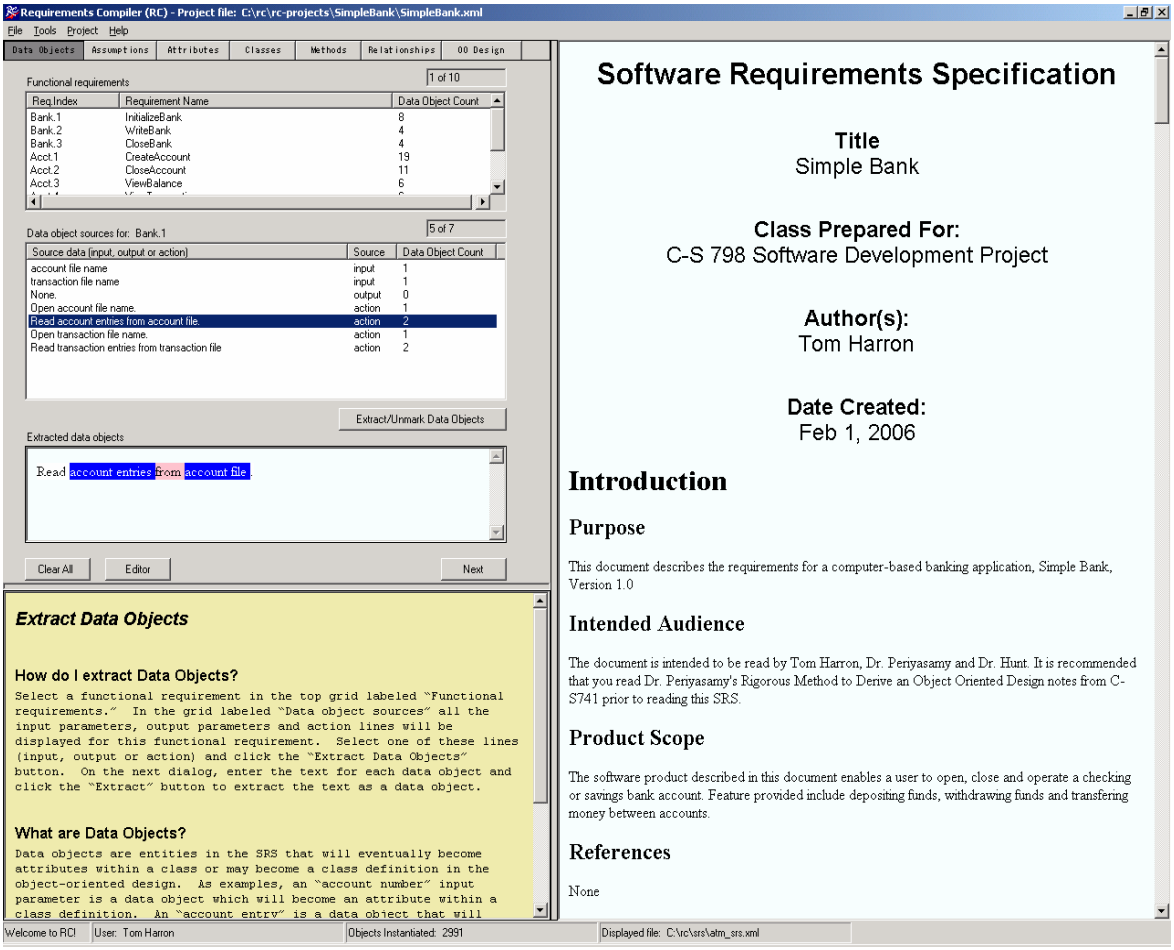


Figure 3.2. RC displaying an IEEE 830-1998 compliant SRS

The next chapter will walk the reader through the RC at a high level and correlate the capabilities in the program to the concepts of the underlying rigorous methodology.

4. Scenario Walkthrough

4.1. Managing projects

Figure 4.1 displays the project creation dialog; the most significant item specified is the selected SRS document. The user must also specify the name of the root class, that is, the class in which program execution will begin. Since RC generates actual code stubs from the eventual derived design, the designer must also select a target language. Languages currently supported include Java, C# and C++.

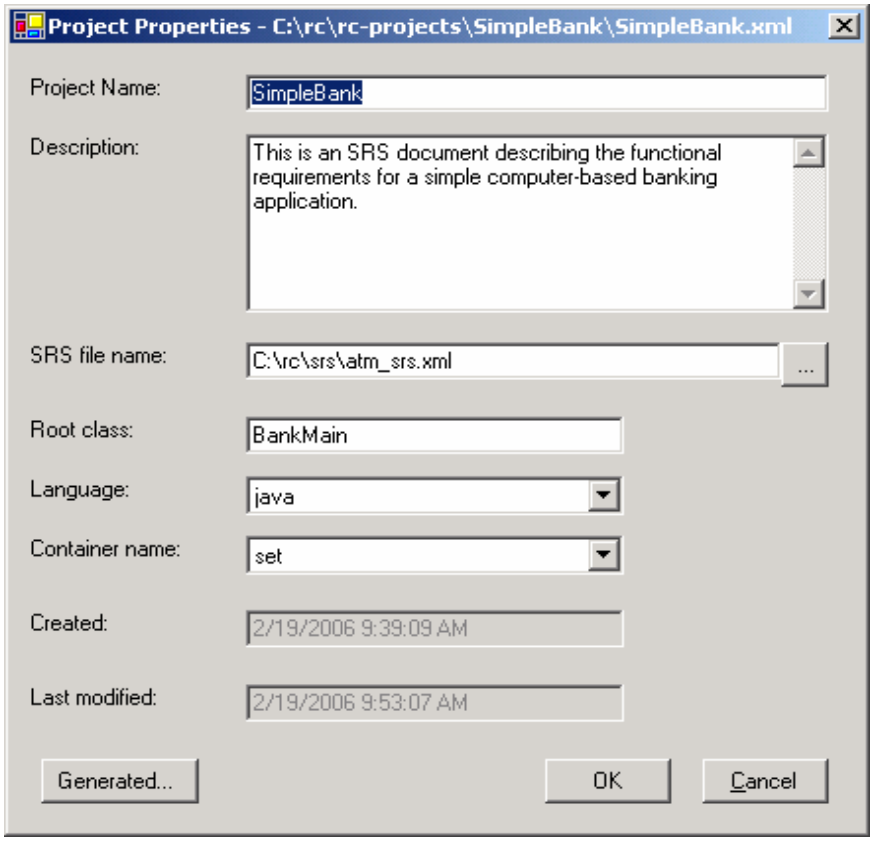


Figure 4.1. Project properties

4.2. Tools

4.2.1. Verification

While creating/editing an SRS document or progressing through the translation steps to derive an object-based design, the user may view warnings and errors. Warnings generally describe conflicting or missing data. In Figure 4.2 below, several warnings are displayed alerting the user to missing information in the functional requirement document.

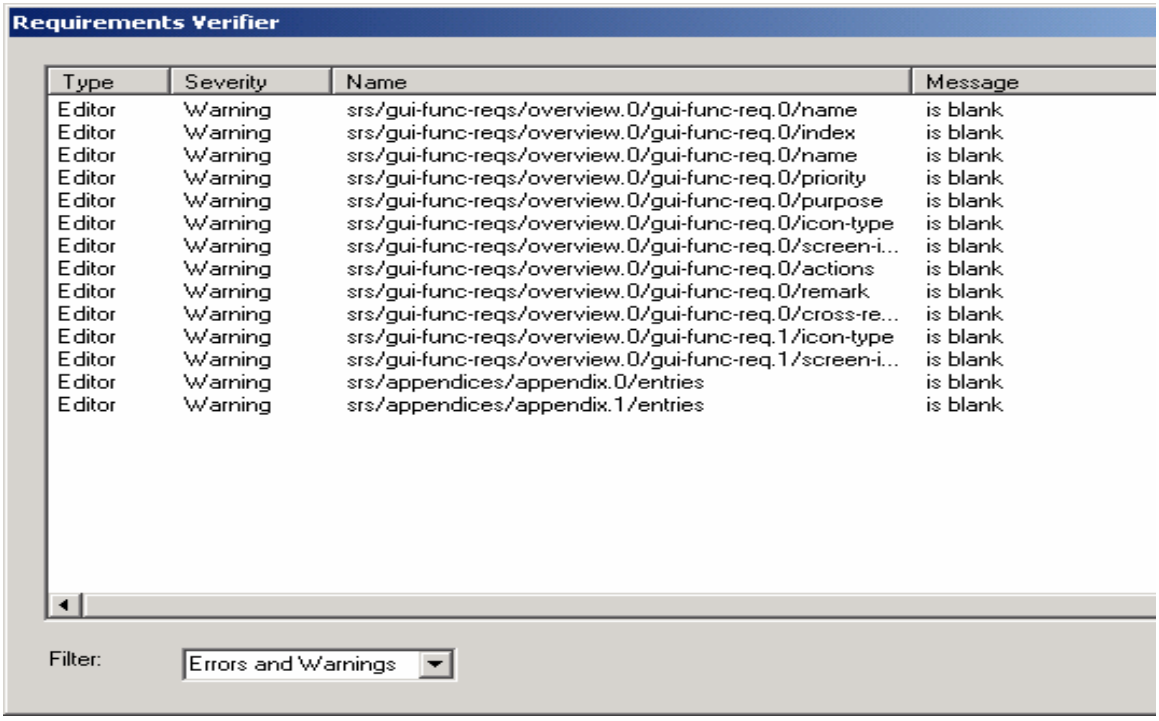


Figure 4.2. RC verifier display

4.2.2. Reference data

RC provides extensibility by enabling the user to define/edit reference data used by the application, thereby providing a means to control choices available within the GUI. For example, the words defined in the reference data for simple Java data types (int, long, double, etc.) are displayed in drop-down boxes within RC to enable the user to select the data type for class attributes. If the user wishes to extend (or constrain) the data types available within the program, they may edit the reference data through a facility supported in RC. Table 4.1 lists the categories of information which can be similarly controlled by the user. Noise words are words such as: the, for and other; these words are inconsequential in terms of design derivation process. Similarly, the user may define ambiguous words, that is; words that are generally considered ambiguous in a functional requirement document and should be avoided. Examples include: easy and faster. These words are highlighted to draw attention to them, so they can be removed from the document.

User editable reference data is a very important facility as each user will likely have definitions particular to their institution or application domain of their project, etc.

Composite data types – Java	Simple data types – Java
Composite data types – C#	Simple data types – C#
Composite data types – C++	Simple data types – C++
Noise words	Ambiguous words
Relationship words	Container words

Table 4.1. Reference data categories

4.2.3. System configuration

Finally, Figure 4.3 displays settings for RC system including the location of project files, SRS documents, etc. An interesting point on this screen is the word colors – the

user has the ability to control the color of various types of words (noise, data objects, ambiguous) that are displayed in the editor. Combined with the mechanism to edit reference data described in the last section, the user has the ability to control the definition of various types of words and the color in which they are displayed in RC environment.

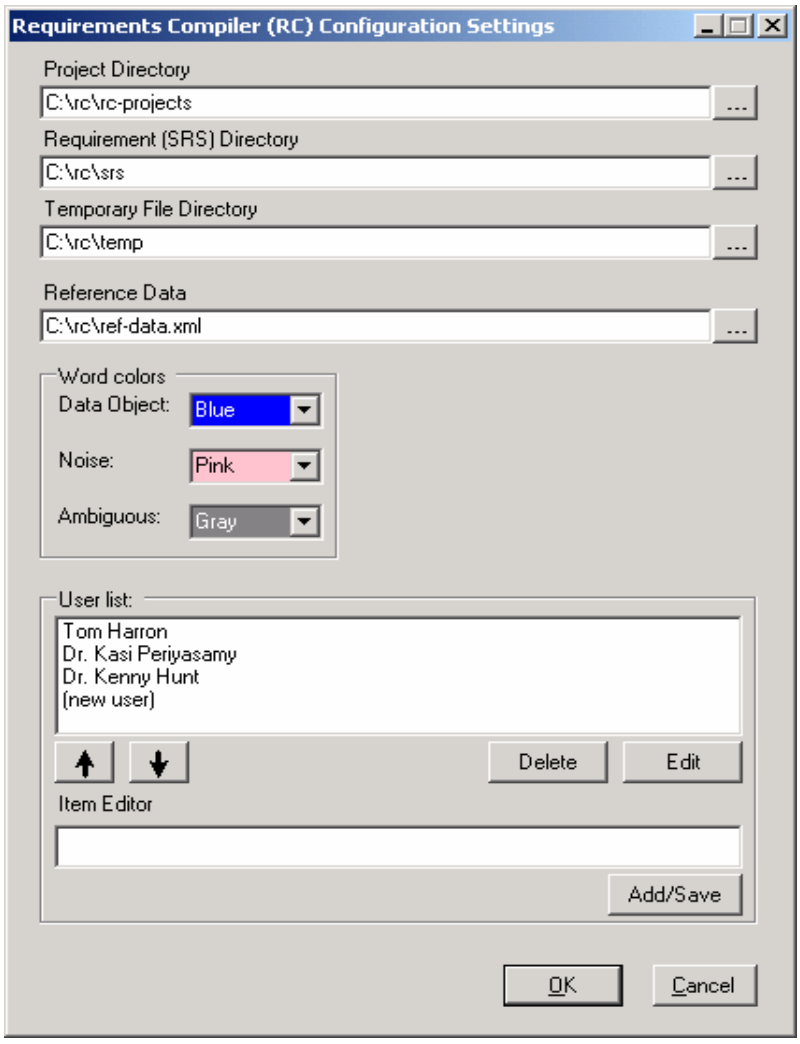


Figure 4.3. RC system settings

4.3. Creating 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 standard. As can be seen in Figure 4.4, all the sections defined in the standard are represented in the tree control. The user simply navigates the nodes defined within the tree control and fills in the required information in the pane immediately to the right (of the tree control). In Figure 4.4, the user is being prompted for input parameters for the functional requirement "Deposit Funds." As the user specifies the information, it is automatically rendered in the rightmost panel in document form, with noise and ambiguous words being highlighted in the chosen color. Online context-sensitive help is provided for each individual element (i.e. each tree node in the GUI) of the SRS document providing guidance for the information expected throughout the SRS.

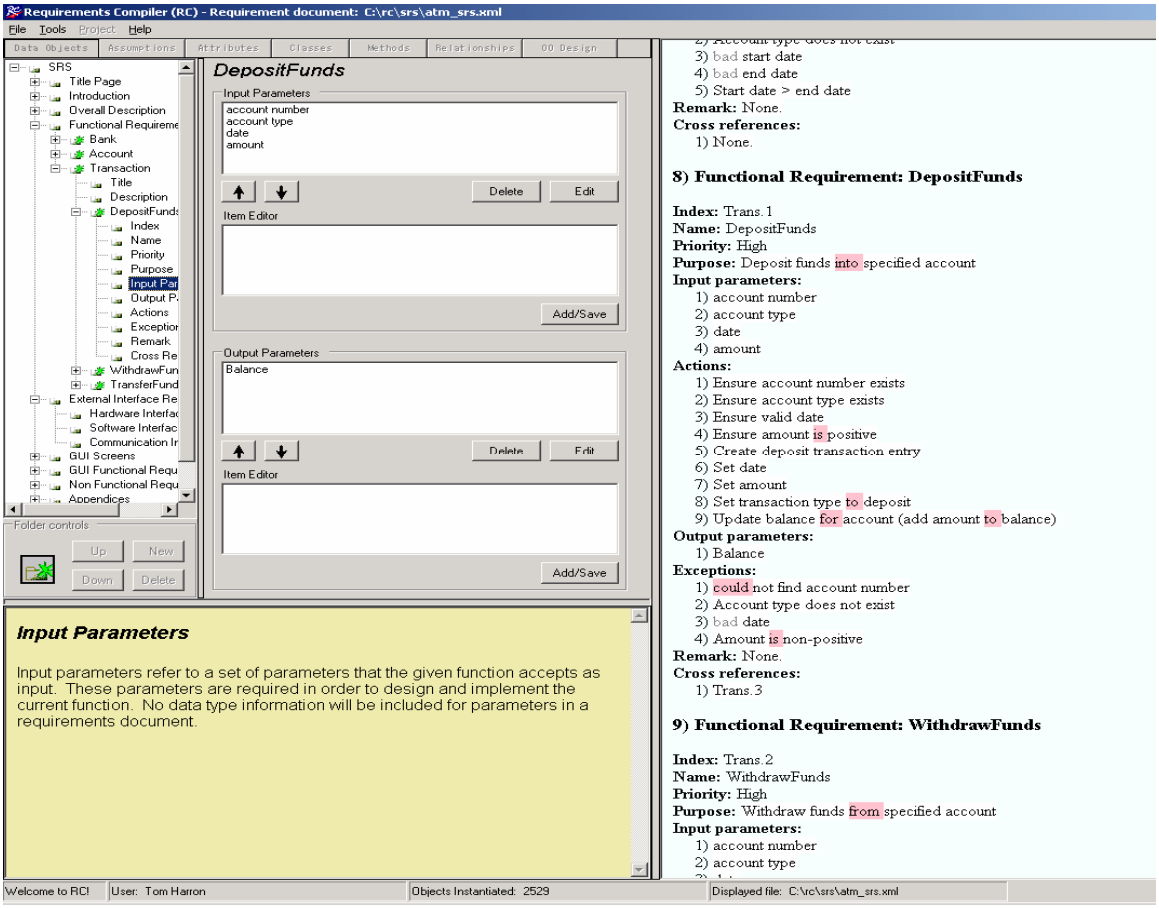


Figure 4.4. Requirement document editor

4.4. Extracting data objects

Once the requirement document has been created, the user is ready to perform the translation from requirements document to object-based design. The first step in the process is to extract data objects in order to create a data dictionary drawn from the requirements specification. Figure 4.5 displays the extraction tool listing an entry from an input parameter from a particular functional requirement. As can be seen, the input is in natural language (e.g. English); the user is responsible to identify data objects (such as “amount” and “transaction type”) by highlighting them and then instructing the application to extract them. The example below shows the user extracting the data object “transaction type.”

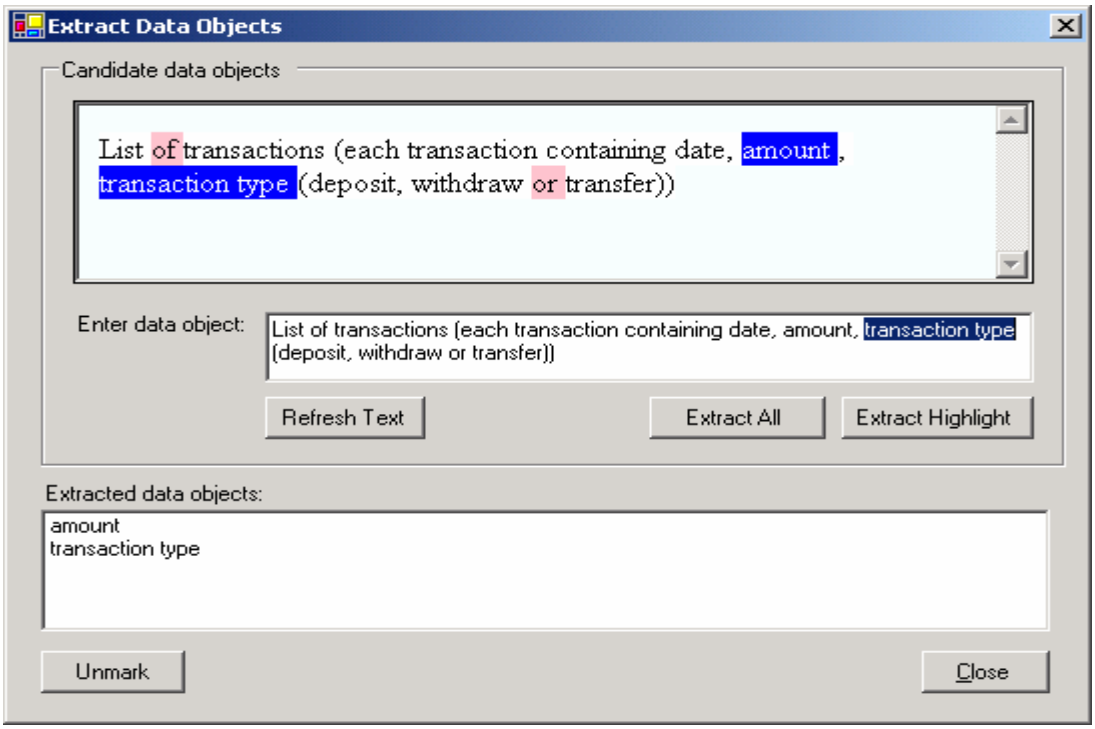


Figure 4.5. Data object extraction tool, showing highlighted data objects

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 assumption.

Figure 4.6 provides a display of the highlighted data objects extracted and also indicates the noise words (grayed out). The color coding enables the user to easily scan the entire SRS document for additional words describing the same data object.

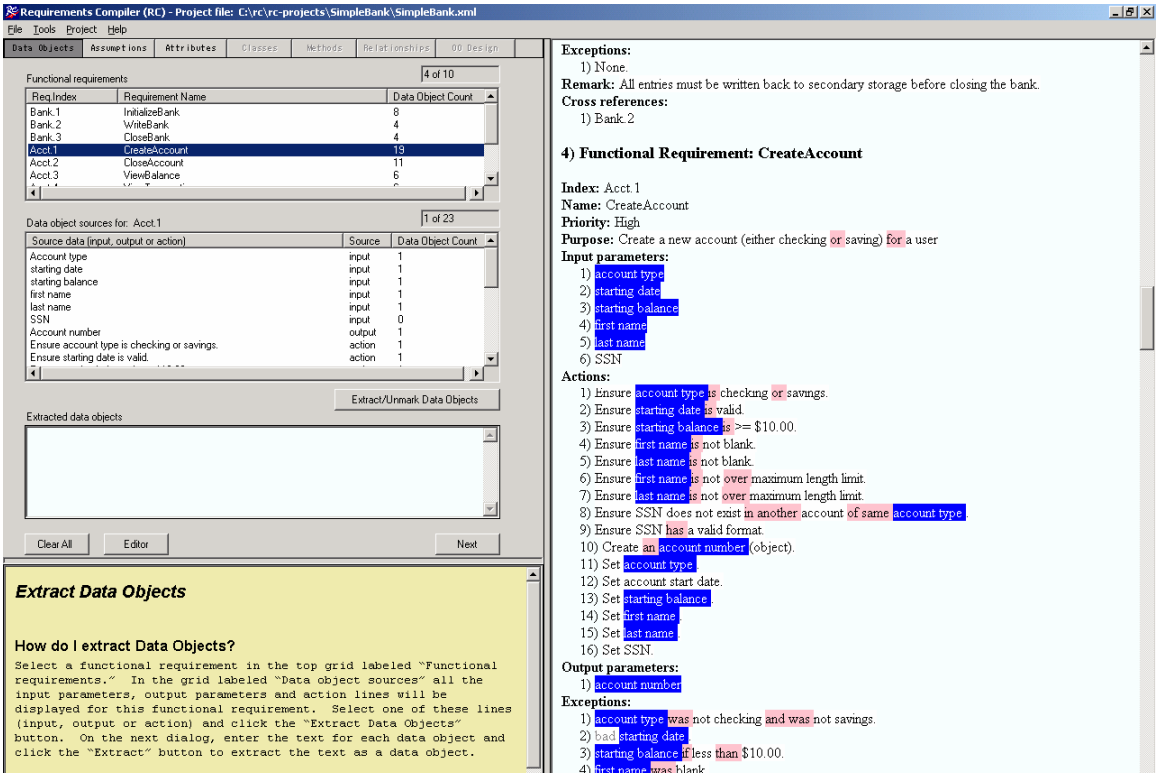


Figure 4.6. Extracting data objects from functional requirement

Similarly, Figure 4.7 provides a display of the assumptions section of the SRS document and the data objects extracted therein.

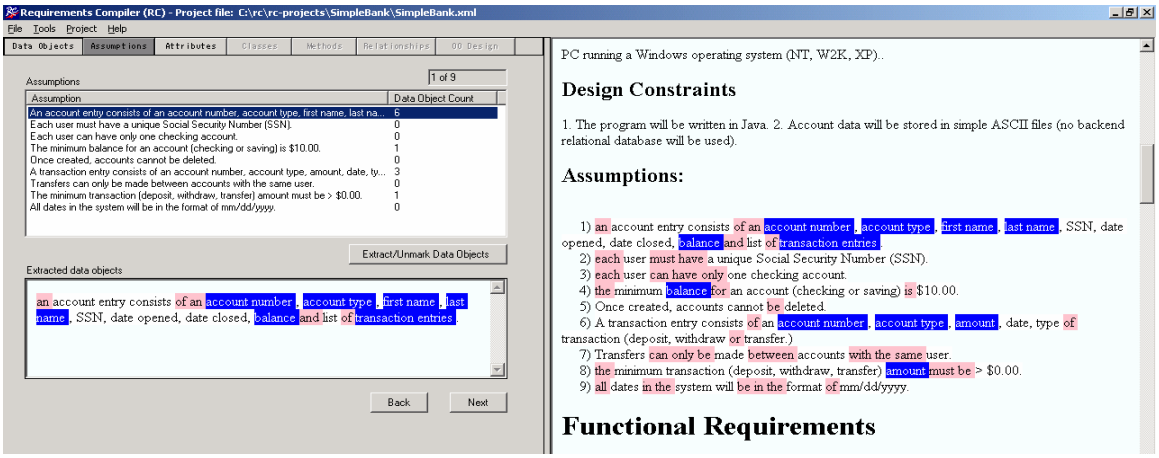


Figure 4.7. Extracting data objects from assumptions

4.5. Classifying data objects

Once all data objects have been extracted, they must be classified as simple or composite. This step is done manually by the designer. 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 section 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 RC. Figure 4.8 displays the data objects extracted from the SRS document and the assigned category (simple or composite) and data type.

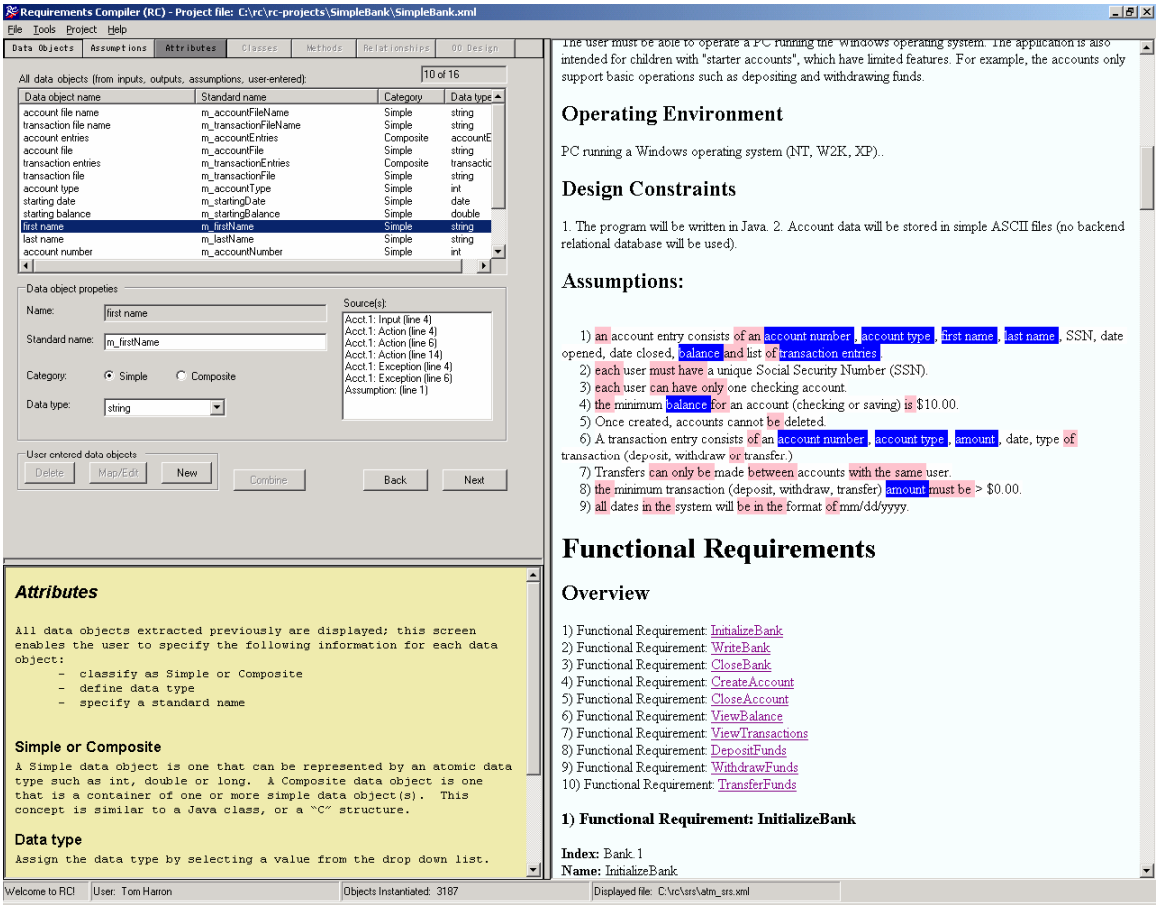


Figure 4.8. Classifying data objects, assigning standard name and data type

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 standardized phrase that will be uniformly applied throughout the requirements and design documents. The standardized name specified by the user will “correct” the inconsistency present in the SRS will be used as attribute names in the design.

4.6. Identifying the attributes of a class

This step allows the designer to define the structural components (attributes) of each class. This step must also be performed manually by the designer. Attributes are selected from the data dictionary built by the extracted data objects. 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. Figure 4.9 displays the attributes selected for a particular class; the user has the ability to control various aspects of each attribute such as the modifier (private, public, protected).

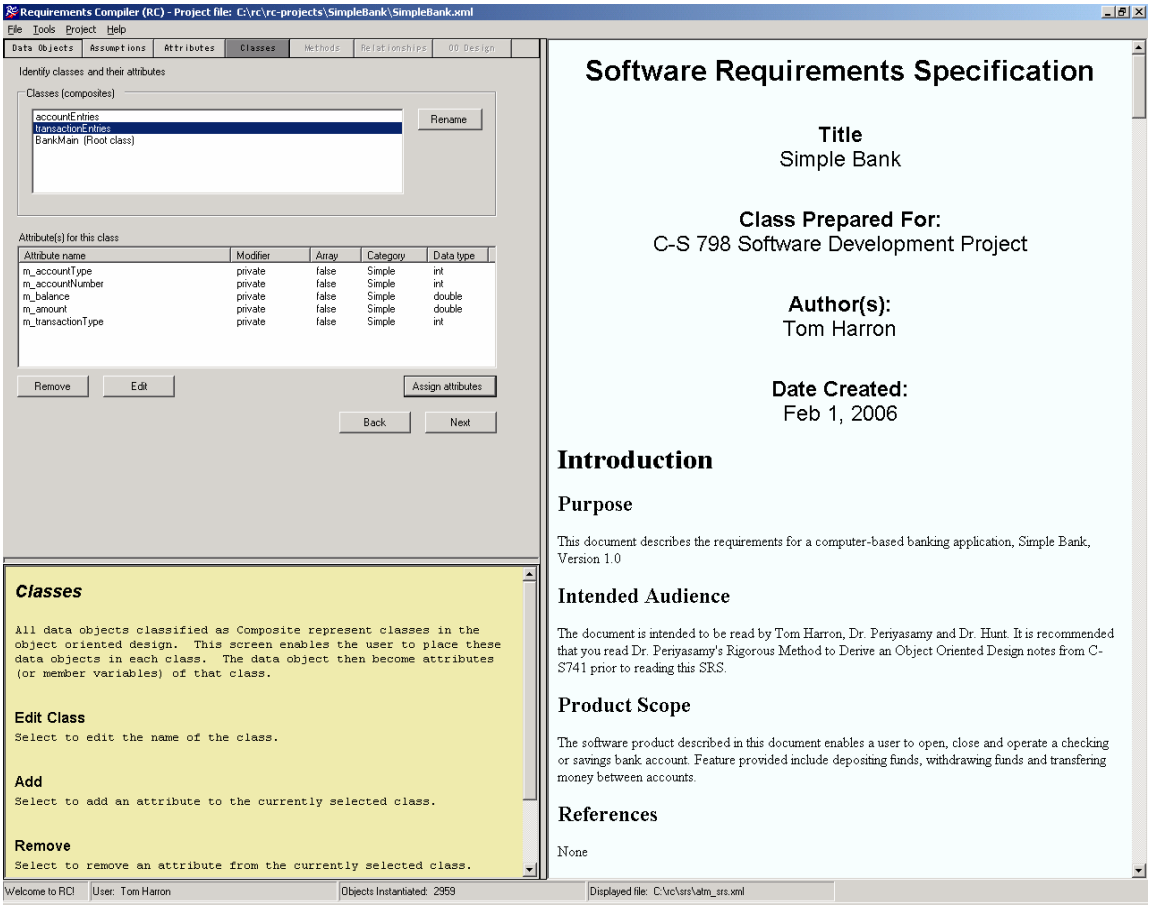


Figure 4.9. Assigning attributes to classes

4.7. Identifying the methods of a class

RC provides support for distributing requirements across classes as 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. See Figure 4.10 for an example of this concept.

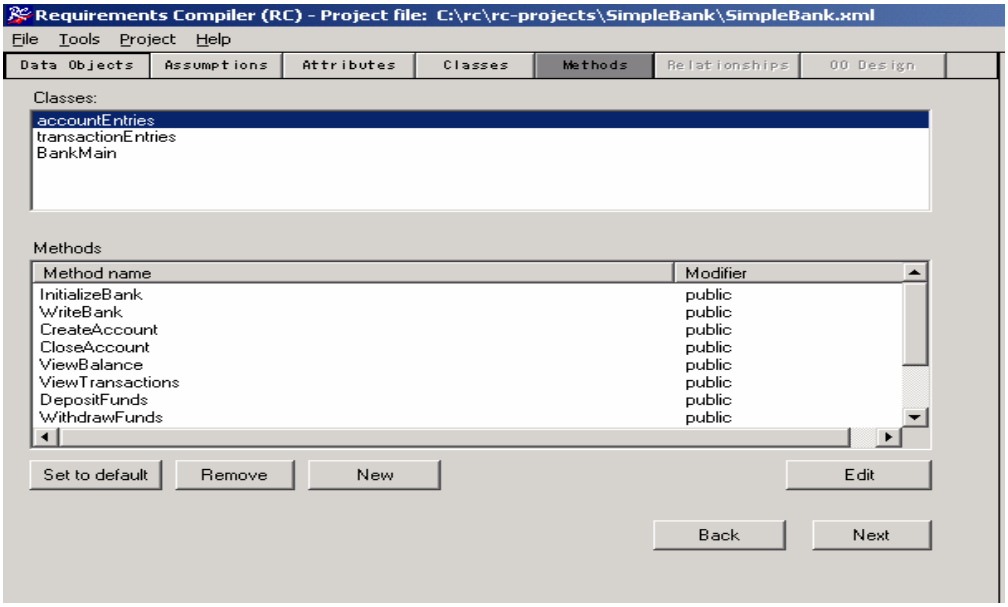


Figure 4.10. Methods distributed to class AccountEntries

4.8. Defining relationships between the classes

There are three types of relationships between the classes in an object-oriented system: aggregation, association or specialization. RC records aggregation relationships among classes based on the assignment of composite data objects as class attributes; an example can be viewed in Figure 4.11. The designer must *manually* identify the other two types of relationships. The designer may also elect to change the aggregation relationships to associations.

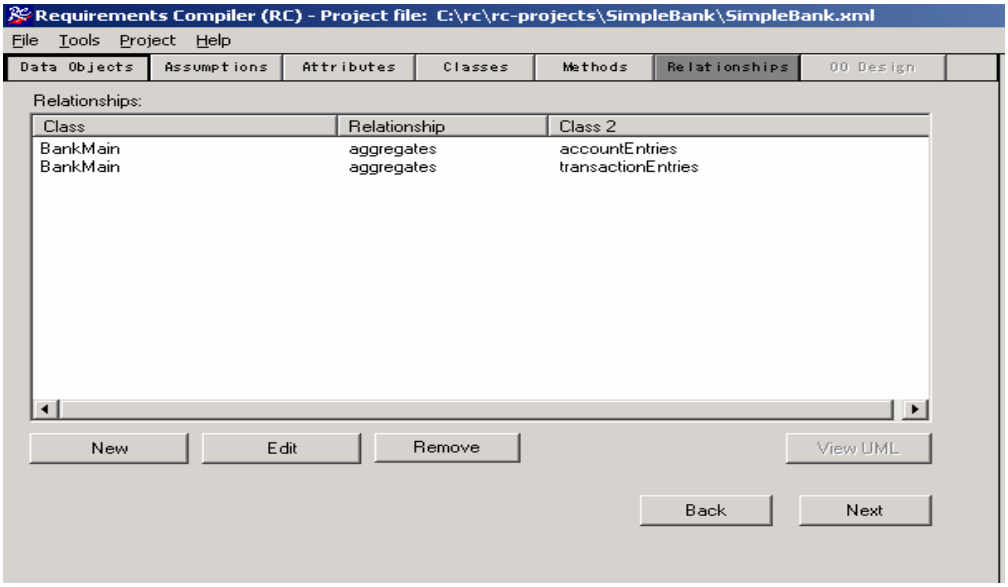


Figure 4.11. Aggregation relationships recorded by RC

4.9. Generating design, code, and reports

RC is able to generate an object-based design document, source code stubs and a report summarizing the consistency between the requirements specification and the derived design. Figure 4.12 displays all of the artifacts generated (left hand pane); the user has the ability to scroll through each of these and view their contents in the file viewer in the rightmost pane of the GUI. The rightmost pane in Figure 4.12 is rendering the generated object-based design document.

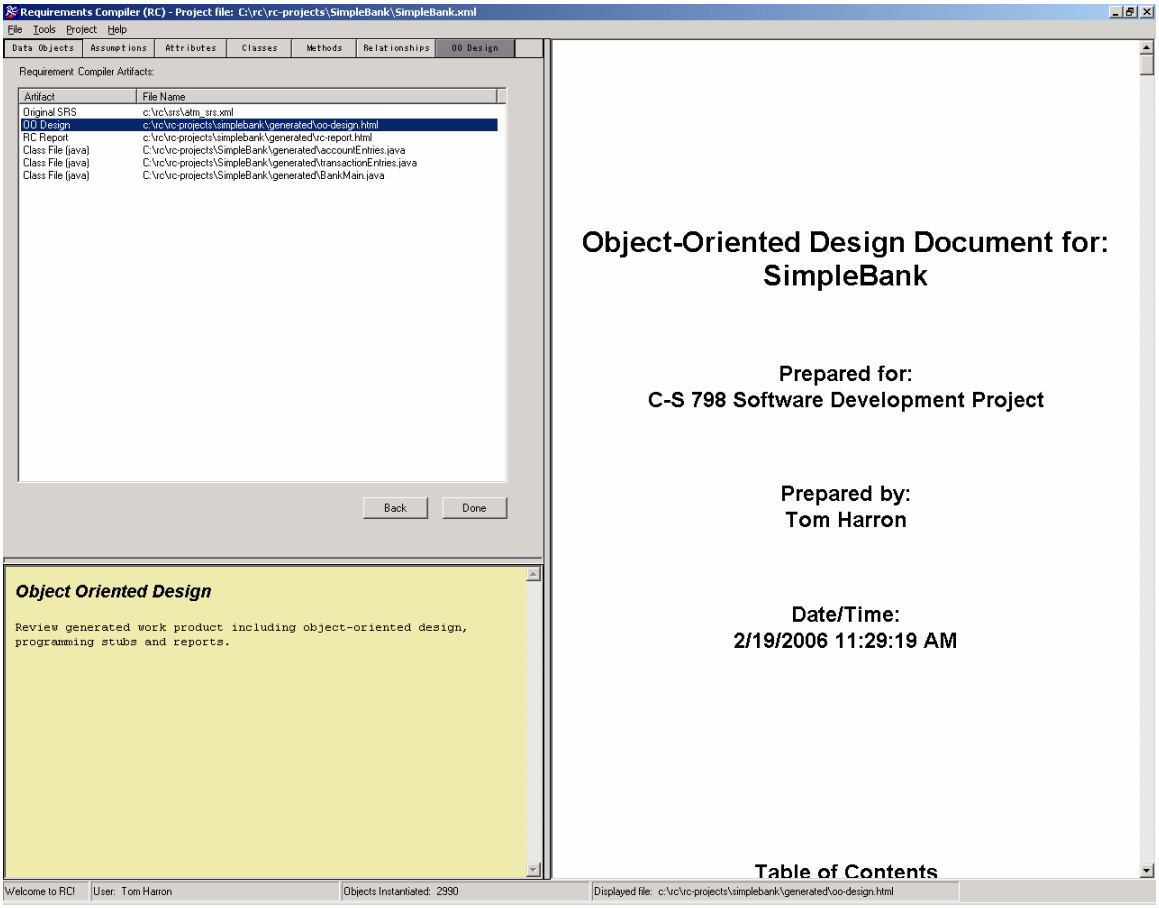


Figure 4.12. Object-based design generated by RC

4.9.1. 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 (i.e. pseudo-code) based on the "Action" clauses from functional requirements. The relationships described in Section 4.8 are also included in the design document. Table 4.2 lists the specific sections generated for each design document and the contents for each. A complete design document generated by RC can be seen in Appendix D.

Design document section	Information specified in section
1.0 Introduction	Introductory information about the design document. RC automatically inserts a reference to the SRS which this design is based upon. It is up to the designer to fill in more descriptive information.
2.0 Design Decisions	The designer must list design decisions taken on the project. RC leaves this section blank.
3.0 UML Class Diagram	A UML class diagram. This section lists the aggregation relationships detected in textual format. The designer is responsible to represent this information in a diagram format.
4.0 Format of a Class Definition	A template section listing the sections provided for each class. RC creates this section in italics text.
5.0 Class Definitions	RC generated design information for each class. This information includes the class name, list of attributes and methods. For each method the calling signature and pseudo-coded is provided.
6.0 References	RC places a reference to the SRS document.
7.0 Appendix A	All design documents should have at least one appendix, therefore RC creates one.

Table 4.2. Design document sections

4.9.2. Code generation

RC generates skeletons of 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.

Figure 4.13 provides an example of code generated in the Java programming language.

```
  /*
   * Class:  accountEntries.java
   *
   * Generated:  2/19/2006 11:27:53 AM
   * User:  Tom Harron
   *
   */
  import javax.swing.*
  import java.util.*;
  import java.awt.*;

  package SimpleBank;

  /**
   * TODO - Put summary of this class here.
   *
   * @author Tom Harron
   * @version 1.0
   */
  public class accountEntries
  {
     // member variable(s)
     private int m_accountType;
     private date m_startingDate;
     private double m_startingBalance;
     private string m_firstName;
     private string m_lastName;
     private int m_accountNumber;
     private date m_closeDate;
     private double m_balance;

     /**
     * @author Tom Harron
     * @version 1.0, 2/19/2006 11:27:53 AM
     *
     * @param int m_accountType (Func.Req.: Acct.3 - ViewBalance,
Input (line 2))
     * @param int m_accountNumber (Func.Req.: Acct.3 - ViewBalance,
Input (line 1))
     *
     * @return double
     *
     */
```

```
     public double ViewBalance(final int m_accountType, final int
m_accountNumber)
     {
        // TODO: Implement ACTIONS for attribute:  m_accountType
        // Line 2) Ensure account type exists

        // TODO: Implement EXCEPTIONS for attribute:  m_accountType
        // Line 2) Account type does not exist
     }
```

Figure 4.13. Generated class stub (Java)

4.9.3. Report generation

RC provides project summary information, SRS document statistics 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. Table 4.4 provides a sample of verification warnings.

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

The following Tables (4.3, 4.4, 4.5 and 4.6) provide examples of a project summary, errors/warnings, functional requirements statistics and extracted data object, respectively.

Project Name	SimpleBank
Report Date/Time:	2/19/2006 11:29:19 AM
Project File	C:\rc\rc-projects\SimpleBank\SimpleBank.xml
Description	This is an SRS document describing the functional requirements for a simple computer-based banking application.
Project File	C:\rc\rc-projects\SimpleBank\SimpleBank.xml
SRS File	C:\rc\srs\atm_srs.xml
Programming language	Java
Root class name	BankMain

Table 4.3. Project summary

Editor	Warning	srs/gui-func-reqs/overview.0/gui-func-req.0/cross-references	is blank
Editor	Warning	srs/gui-func-reqs/overview.0/gui-func-req.1/icon-type	is blank
Editor	Warning	srs/gui-func-reqs/overview.0/gui-func-req.1/screen-ident	is blank
Editor	Warning	srs/appendices/appendix.0/entries	is blank
Editor	Warning	srs/appendices/appendix.1/entries	is blank

Table 4.4. Verification errors and warnings

Index	**Name**	**Data objs. extracted**	**Number of input parms**	**Number of output parms**	**Number of action lines**	**Number of exception lines**
Bank.1	InitializeBank	2	2	1	4	4
Bank.2	WriteBank	2	2	1	2	1
Bank.3	CloseBank	2	2	1	3	1
Acct.1	CreateAccount	6	6	1	16	8
Acct.2	CloseAccount	4	3	1	7	3
Acct.3	ViewBalance	3	2	1	3	2
Acct.4	ViewTransactions	4	4	1	6	5
Trans.1	DepositFunds	4	4	1	9	4
Trans.2	WithdrawFunds	4	4	1	10	5
Trans.3	TransferFunds	8	5	2	16	5
Totals	n/a	39	34	11	76	38

Table 4.5. Functional requirement statistics

Extracted name	**Standard name (name of attribute/class)**	**Category**	**Data type**	**Source(s)**	**Class(es) data object is member of (as attribute)**
account file name	m_accountFileName	Simple	string	Bank.1: Input (line 1), Bank.1: Action (line 1), Bank.1: Exception (line 1)	AccountEntries, TransactionEntries
transaction file name	m_transactionFileName	Simple	string	Bank.1: Input (line 2), Bank.1: Action (line 3), Bank.1: Exception (line 3)	BankMain

Table 4.6. Extracted data objects

5. Limitations

The Requirements Compiler application is a good start in the direction of its intended goals: Enforcing a disciplined approach in the creation of software requirements specification documents, translating those requirements into an object-based design and the generation of reports and code stubs. Despite this good start, there are several significant enhancements required to make it a commercially viable tool – some of these can be seen in the next section of this manuscript. Besides these missing features, general limitations will be listed here.

The object-based designs generated by RC are derived from a mechanical process and therefore are rather simplistic; only aggregation relationships are generated (no inheritance, no polymorphism, etc.) In addition, the tool does not leverage industry standard design patterns (improving cohesion, reducing coupling) common to various application domains. Due to these limitations, the designer will likely need to evolve the design from the starting point generated by RC. Great care must be taken while changing the design (adding classes, attributes, methods, relationships, etc.) to maintain the consistency (i.e. mappings) with the requirements document.

Requirements Compiler assumes a waterfall development methodology, in which is it assumed that requirements are defined very well prior to the start of the project [8]. Therefore RC will have limited value in iterative development methodologies such as prototyping methods.

Requirements Compiler also assumes a project starting from scratch. There is currently no facility to import the functional requirements and object-oriented design of an existing legacy system and then extend the project by adding additional functional requirements and updating an existing design with changes generated by RC. In general, RC has no capabilities to handle “version 2” of a project initially designed with or without the tool.

RC also expects the input (SRS document) to have been written using the supplied structure editor. The editor produces a requirements document in XML format that is read by the compiler portion of the program [4, 16]. RC does not import requirements documents in other formats such as Microsoft Word.

6. Continuing Work

The Requirements Compiler tool is ready for beta testing at this time. While the product is quite robust, the product covers a wide array of software engineering topics (functional requirements engineering, object-based design, generating source code (Java, C#, C++), etc.). As a result, there is plenty of room for enhancement and product evolution. Listed below are several categories for additional work, most of which would be required to commercialize the product.

Rigorous method extensions

- Enable the user to edit the design derived by RC. As the user edits the design (introducing additional classes, modifying class relationships, etc.), maintain the consistency with the SRS document.

Usability

- Continue testing with additional SRS documents; all testing completed thus far have been with SRS documents containing less than 20 requirements.
- Add additional verification rules and error/warning messages to enhance product robustness and inform the user of the ramification of design decisions.
- Enhance interaction between editing an RC project (i.e. the rigorous method steps) and moving back and forth to the editor. Currently RC enables the user to append additional functional requirements to the SRS, even if you've already started the rigorous method steps. However, the user can only append functional requirements (to the list of existing functional requirements); they may not insert functional requirements between existing requirements, nor may they delete or

edit existing requirements. The addition of these extra features would also require additional warnings, in the case the user deletes significant information that is already mapped in the rigorous method, etc.

- Improve user interface in general (support drag-n-drop, enter data directly in grid controls (instead of separate windows)); complete all online context-sensitive help.

Code generation

- Provide more options to enable developers to control generated source code: hungarian notation, doxygen comments, copyright lines, enum's, etc.
- Add constructors, destructors and other support methods (ToString(), etc.).

Document generation (Report, Design, SRS)

- Render files in .pdf format (rather than HTML). The .pdf format supports page numbers, section headers (1.1, 1.2, etc.) and table of content capabilities much more elegantly than HTML. XSLT could be employed to support numerous formats easily [12].
- Write class relationship information to a file format that can be opened and viewed by industry leading UML tools [15].
- Add more information to the RC diagnostic report. There are numerous categories of information that developers may find useful in the report; it is expected that beta testers will drive the requirements for these enhancements.

7. Conclusion

This manuscript describes the basic features of a software environment that supports the derivation of an object-based design by creating and analyzing a functional requirements specification. RC derives an object-based design from 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 RC is interactive it enables the designer to dynamically reconfigure the design and ensure consistency between design and requirements. Work is also underway to augment RC with software design document in a format that can be viewed directly using commercially available UML tools.

It is the author's belief that classroom use of such an application will 1) teach a subset of fundamental aspects of object-oriented design 2) teach the importance of structured requirement specifications, and 3) alleviate much of the burdensome work required for deriving a design consistent with a set of procedural requirements. Work is underway to provide this tool in the classroom setting to measure its value in educational benefits.

Due the limitations listed in Section 5, currently RC has limited scope in industrial applications, but through further work addressing these issues it is hoped it will become more viable.

8. Bibliography

[1] Joint Task Force on Computing Curricula, "Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering", 2004.

[2] C. Petsold, *Programming Microsoft Windows with C#, Microsoft Press,* 2002.

[3] Cimitile et al., "Identifying Objects in Legacy Systems", *5th International Workshop on Program Comprehension*, Dearborn, MI, 1997, pp. 138-147.

[4] D. Esposito, *Applied XML Programming for Microsoft .NET,* Microsoft Press, 2002.

[5] E. Brown, *Windows Forms Programming in C#*, Manning Publications, 2002.

[6] E. Castro, *HTML for the World Wide Web,* Peachpit Press, 1996.

[7] IEEE Recommended Practice for Software Requirements Specification, *IEEE Standard 830-1998 (Revision of IEEE Std 830-1993),* IEEE Computer Society Press, 1998.

[8] I. Sommerville, *Software Engineering 6th Edition,* Addison Wesley, 2001.

[9] 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.

[10] J. Levine, T. Mason and D. Brown, *lex and yacc*, O'Reilly, 1995.

[11] J. Liberty, *Programming C#, Third Edition*, O'Reilly, 2003.

[12] M. van Otegem, *Sams Teach Yourself XLST in 21 Days*, Sams, 2002.

[13] 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.

[14] 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.

[15] T. Pender, *UML Weekend Crash Course*, Wiley Publishing, Inc., 2002.

[16] W. Stanek, *XML Pocket Consultant*, Microsoft Press, 2002.

Appendix A: RC High-level Class Diagram

Figure A.1 presents a UML class diagram depicting the classes and relationships of the Requirements Compiler project.

Program execution begins in RCMain; this class is responsible for managing RC projects (RCProject), system settings (RCOptions), reference data (RCRefSet, RCRefEntry) and verification (RCVerifier).

RCReqDoc class holds the parsed software requirements specification (SRS) document. The requirement document is composed of a collection of RCFuncReq objects, each representing a single functional requirement. The additional classes of RCFuncReqDetail, RCDataObj and RCPhase break down the functional requirements into more details, extracted data objects and marked phrases respectively. RCPhrase class is used to hold any marked item: ambiguous words, noise words and data object phrases.

The last remaining major component of the design is RCRigMethod class which holds all entities associated with the rigorous method. As the user progress through the steps in the rigorous method, the aggregated classes in the design are populated. RCRigMethod aggregates RCAttribute class directly; this is a list of all extracted data objects after they have been categorized as either simple or composite data objects. The data stored in these objects was copied by the collection of RCDataObj classes aggregated by RCReqDoc class – this is a list of the actual data objects extracted from the requirements document. The remaining classes (RCClass, RCAttribute, RCMethod, RCRelationship, etc.) hold the information pertaining to the classes formed by the rigorous method.

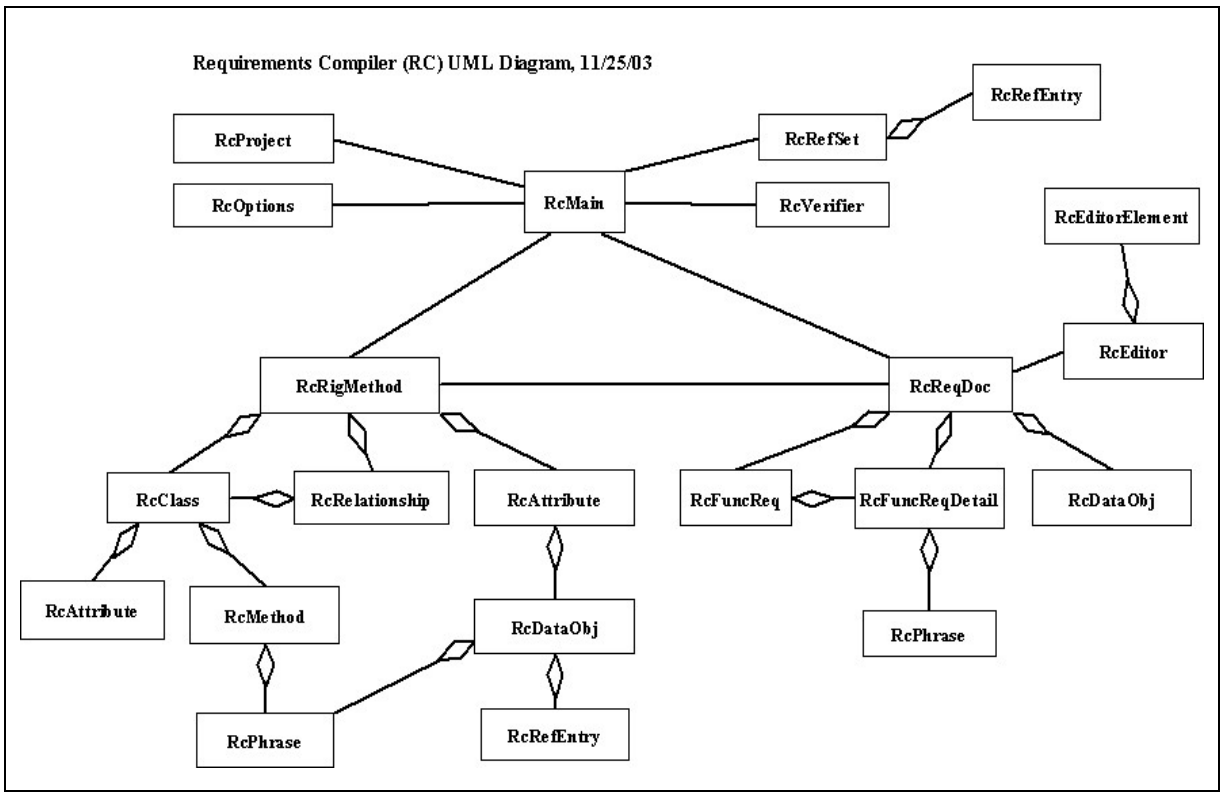


Figure A.1. RC high level class diagram

Appendix B: Implementation Notes

The Requirements Compiler project was implemented using C# programming language in the Microsoft Visual Studio .NET environment [2, 5, 11]. All external files (RC project files, configuration settings, reference data, etc.) are stored in XML format. C# GUI programming matches Java almost entirely (i.e. Panel instead JPanel, etc.)

The Requirements Compiler project was implemented to be extensible (i.e. data-driven). Table B.1 lists the significant files in the project; by editing these files (either via facilities provided in the RC or directly in an XML editor) the user has the ability to control the contents of drop-down boxes, etc. The Purpose column explains how the data in these files is utilized within the application.

RC file(s)	**Purpose**
rc-config.xml	RC system settings. These values are editable via the Tools | Options menu item.
ref-data.xml	RC reference data file holding words for all categories (simple data types, composite data types, noise words, ambiguous words, relationship words and priority words). The data in this file is editable via the Tools | Reference menu item.
rc-editor.xml	This file controls the display of the data entry screens in RC editor. While editable, the user must take extreme care as there are many dependencies required for proper definitions.
help*.html	Help directory contains online context-sensitive help files for application. The user may edit these files to customize online help for their organization.

Table B.1. RC external files

The next three figures (B.1, B.2 and B.3) display the structure of RC project files, the reference data file and one element within the RC editor configuration file, respectively.

```
<rc-config>
  <project-path>C:\rc\rc-projects</project-path>
  <srs-path>C:\rc\srs</srs-path>
  <reference-file>C:\rc\ref-data.xml</reference-file>
  <temp-path>C:\rc\temp</temp-path>
  <language>java</language>
  <container>array</container>
  <noise-color>Pink</noise-color>
  <ambiguous-color>Gray</ambiguous-color>
  <data-object-color>Blue</data-object-color>
  <all-users>
    <user>Tom Harron</user>
    <user>Dr. Kasi Periyasamy</user>
    <user>Dr. Kenny Hunt</user>
    <user>(new user)</user>
  </all-users>
  <last-user>Tom Harron</last-user>
  <last-project>C:\rc\rc-projects\SimpleBank\SimpleBank.xml</last-
project>
</rc-config>
```

Figure B.1. RC system settings file rc-config.xml

```
  <Noise>
      <data>after</data>
  </Noise>
  <Relationship>
      <data>aggregation</data>
  </Relationship>
  <Simple-Java>
      <data>bool</data>
      <data>date</data>
      <data>double</data>
  </Simple-Java>
```

Figure B.2. Reference data file ref-data.xml

```
    <element>
      <!-- Introduction, Product Scope -->
      <title>Product Scope</title>
      <control>text</control>
      <height>15</height>
```

```
      <scr-id>12</scr-id>
      <scr-order>1</scr-order>
      <repeat>no</repeat>
      <help-file>ed-product-scope.html</help-file>
      <xml-path>srs/introduction/product-scope</xml-path>
      <xml-list-tag></xml-list-tag>
      <xml-parent>srs/introduction</xml-parent>
      <verif-rule>bl-warn</verif-rule>
    </element>
```

Figure B.3. Editor configuration file rc-editor.xml

Figure B.4 depicts the code growth over the course of the project. The lower line depicts the amount of application code written (i.e. classes shown in Figure A.1), approximately 15,000 lines of code. The upper line represents all code written for the project; this is counting all the GUI code (dialogs and forms) in addition to the application code. A significant portion of the GUI code was generated by the C# programming language. The total lines of code for RC project is 31, 783.

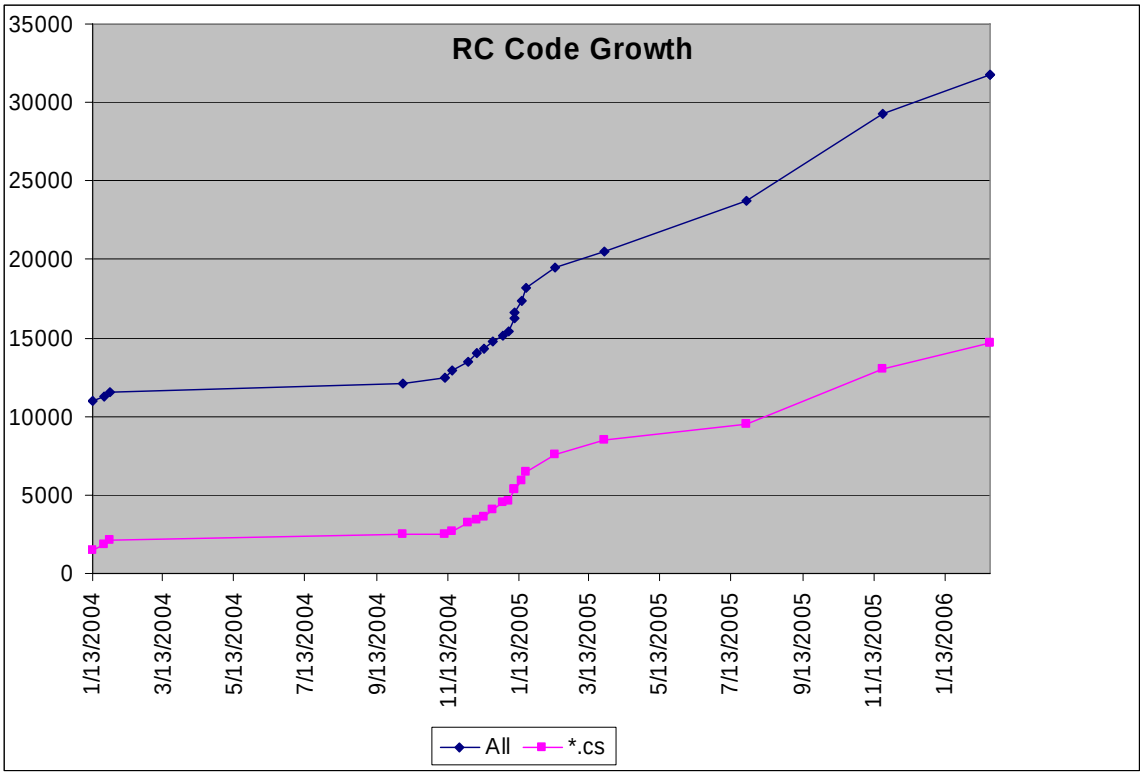


Figure B.4. Code growth for Requirements Compiler project

Appendix C: Sample SRS document

This appendix contains an SRS document created by the structured editor supplied by RC. Some of the content has been removed from this document (external hardware interfaces, GUI screens, GUI functional requirements, non-functional requirements) to keep the size manageable for this manuscript. This SRS served as a test case during RC development and is rather simplistic; for example the actions/exceptions listed in each functional requirement are quite terse.

Software Requirements Specification

Title
Simple Bank

Class Prepared For:
C-S 798 Software Development Project

Author(s):
Tom Harron

Date Created:
Feb 1, 2006

Introduction

Purpose
This document describes the requirements for a computer-based banking application, Simple Bank, Version 1.0

Intended Audience
The document is intended to be read by Tom Harron, Dr. Periyasamy and Dr. Hunt. It is recommended that you read Dr. Periyasamy's Rigorous Method to Derive an Object-oriented Design notes from C-S 741 prior to reading this SRS.

Product Scope
The software product described in this document enables a user to open, close and operate a checking or savings bank account. Feature provided include depositing funds, withdrawing funds and transferring money between accounts.

References
None

Overall Description

Product Perspective
This document will be used as a case study for Tom Harron's C-S 798 capstone project. Tom will use this SRS to exercise the rigorous method to derive an object-oriented design. The scope of this project is kept extremely simple in order to have a small case study to exercise the rigorous method.

Product Functions
1) Open bank account
2) Close back account
3) Deposit funds
4) Withdraw funds
5) Transfer funds

User Classes
The user must be able to operate a PC running the Windows operating system. The application is also intended for children with "starter accounts", which have limited features. For example, the accounts only support basic operations such as depositing and withdrawing funds.

Operating Environment
PC running a Windows operating system (W2K or XP).

Design Constraints
1. The program will be written in Java. 2. Account data will be stored in simple ASCII files (no backend relational database will be used).

Assumptions

1) an account entry consists of an account number, account type, first name, last name, SSN, date opened, date closed, balance and list of transaction entries.
2) each user must have a unique Social Security Number (SSN).
3) each user can have only one checking account.
4) the minimum balance for an account (checking or saving) is $10.00.
5) Once created, accounts cannot be deleted.
6) A transaction entry consists of an account number, account type, amount, date, type of transaction (deposit, withdraw or transfer.)
7) Transfers can only be made between accounts with the same user.
8) the minimum transaction (deposit, withdraw, transfer) amount must be > $0.00.
9) all dates in the system will be in the format of mm/dd/yyyy.

Functional Requirements

Overview

1) Functional Requirement: InitializeBank
Index: Bank.1
Name: InitializeBank
Priority: High
Purpose: Initialize bank by reading account and transaction entries from secondary storage into internal memory.
Input parameters:
1) account file name
2) transaction file name
Actions:
1) Open account file name.
2) Read account entries from account file.
3) Open transaction file name.
4) Read transaction entries from transaction file
Output parameters:
1) None.

Exceptions:

1) Account file name does not exist.
2) Account entries missing or corrupt.
3) Transaction file name does not exist.
4) Transaction entries missing or corrupt

Remark: This requirement is clarifies the information required to initialize the bank.
Cross references:

1) None.

2) Functional Requirement: WriteBank

Index: Bank.2
Name: WriteBank
Priority: High
Purpose: to write all account and transaction entries to secondary storage. All data should be written out faster than last version of software.
Input parameters:

1) account entries
2) transaction entries

Actions:

1) Write account entries to secondary storage.
2) Write transaction entries to secondary storage.

Output parameters:

1) None.

Exceptions:

1) Disk full while writing information to secondary storage

Remark: None.
Cross references:

1) None.

3) Functional Requirement: CloseBank

Index: Bank.3
Name: CloseBank
Priority: Medium
Purpose: Close all files and terminate bank program.
Input parameters:

1) Name of the account file
2) name of transaction file

Actions:

1) Close account file.
2) Close transaction file.
3) Terminate bank program.

Output parameters:

1) None.

Exceptions:

1) None.

Remark: All entries must be written back to secondary storage before closing the bank.
Cross references:
1) Bank.2

4) Functional Requirement: CreateAccount

Index: Acct.1
Name: CreateAccount
Priority: High
Purpose: Create a new account (either checking or saving) for a user
Input parameters:
1) Account type
2) starting date
3) starting balance
4) first name
5) last name
6) SSN

Actions:
1) Ensure account type is checking or savings.
2) Ensure starting date is valid.
3) Ensure starting balance is >= $10.00.
4) Ensure first name is not blank.
5) Ensure last name is not blank.
6) Ensure first name is not over maximum length limit.
7) Ensure last name is not over maximum length limit.
8) Ensure SSN does not exist in another account of same account type.
9) Ensure SSN has a valid format.
10) Create an account number (object).
11) Set account type.
12) Set account start date.
13) Set starting balance.
14) Set first name.
15) Set last name.
16) Set SSN.

Output parameters:
1) Account number

Exceptions:
1) Account type was not checking and was not savings.
2) starting date.
3) Starting balance if less than $10.00.
4) First name was blank.
5) Last name was blank.
6) First name was over length limit.
7) Last name was over length limit.
8) SSN already existed in another account.

Remark: None
Cross references:
1) None

5) Functional Requirement: CloseAccount

Index: Acct.2
Name: CloseAccount
Priority: High
Purpose: Close an existing account (either checking or saving) for a user
Input parameters:
1) Account number
2) account type
3) close date

Actions:
1) Ensure account number exists
2) Ensure account type exists
3) Ensure valid close date
4) Set account close date
5) Set balance to zero
6) Clear account type (closing this account)
7) Return previous balance

Output parameters:
1) Previous balance

Exceptions:
1) Account number does not exist
2) Account type does not exist
3) close date

Remark: The balance amount returned (if positive) can be used to cash out the account.
Cross references:
1) None.

6) Functional Requirement: ViewBalance

Index: Acct.3
Name: ViewBalance
Priority: High
Purpose: View current balance of specified account
Input parameters:
1) account number
2) account type

Actions:
1) Ensure account number exists
2) Ensure account type exists
3) Return current balance to user

Output parameters:

1) Balance

Exceptions:

1) could not find account number
2) Account type does not exist

Remark: None.

Cross references:

1) None.

7) Functional Requirement: ViewTransactions

Index: Acct.4

Name: ViewTransactions

Priority: High

Purpose: View account activity (transactions) for specified account between two dates

Input parameters:

1) account number
2) account type
3) start date
4) end date

Actions:

1) Ensure account number exists
2) Ensure account type exists
3) Ensure start date valid
4) Ensure end date valid
5) Ensure start date less than or equal to end date
6) Return list of transactions to user

Output parameters:

1) List of transactions (each transaction containing date, amount, transaction type (deposit, withdraw or transfer))

Exceptions:

1) could not find account number
2) Account type does not exist
3) start date
4) end date
5) Start date > end date

Remark: None.

Cross references:

1) None.

8) Functional Requirement: DepositFunds

Index: Trans.1

Name: DepositFunds

Priority: High

Purpose: Deposit funds into specified account

Input parameters:

1) account number
2) account type
3) date
4) amount

Actions:

1) Ensure account number exists
2) Ensure account type exists
3) Ensure valid date
4) Ensure amount is positive
5) Create deposit transaction entry
6) Set date
7) Set amount
8) Set transaction type to deposit
9) Update balance for account (add amount to balance)

Output parameters:

1) Balance

Exceptions:

1) could not find account number
2) Account type does not exist
3) date
4) Amount is non-positive

Remark: None.

Cross references:

1) Trans.3

9) Functional Requirement: WithdrawFunds

Index: Trans.2
Name: WithdrawFunds
Priority: High
Purpose: Withdraw funds from specified account
Input parameters:

1) account number
2) account type
3) date
4) amount

Actions:

1) Ensure account number exists
2) Ensure account type exists
3) Ensure valid date
4) Ensure amount is positive
5) Ensure account balance >= (amount to be withdrawn + $10.00)
6) Create withdraw transaction entry
7) Set date
8) Set amount

9) Set transaction type to withdraw
10) Update balance for account (subtract amount from balance)

Output parameters:

1) Balance

Exceptions:

1) could not find account number
2) Account type does not exist
3) date
4) Amount is non-positive
5) Resulting balance is less than $10.00.

Remark: None.

Cross references:

1) Trans.3

10) Functional Requirement: TransferFunds

Index: Trans.3

Name: TransferFunds

Priority: High

Purpose: Transfer funds from one account type to another. The account number must be identical.

Input parameters:

1) account number
2) source account type
3) destination account type
4) date
5) amount

Actions:

1) Ensure account number exists
2) Ensure source account type exists
3) Ensure destination account type exists
4) Ensure valid date
5) Ensure amount is positive
6) Ensure source account type balance >= amount to be transferred
7) Create transaction entry for source account type
8) Set date
9) Set amount
10) Set transaction type to transfer-withdraw
11) Update balance for account type (subtract amount from balance)
12) Create transaction entry for destination account type
13) Set date
14) Set amount
15) Set transaction type to transfer-deposit
16) Update balance for account type (add amount to balance)

Output parameters:

1) balance of source account type
2) balance of destination account type

Exceptions:

1) could not find account number
2) Account number does not have specified source account type
3) Account number does not have specified destination account type
4) date
5) Amount is non-positive

Remark: A transfer is performed by withdrawing the funds from the 'source' account type and then depositing the fund in the 'destination' account type

Cross references:

1) Trans.1
2) Trans.2

External Hardware Interfaces
GUI Screens
GUI Functional Requirements
Non-Functional Requirements
Appendicies

Appendix 1

Title: Appendix C: Definitions, Acronyms and Abbreviations

Overview: The following definitions, acronyms and abbreviations are used throughout this document.

Entries:

1) CS741 Computer Science class at UW-La Crosse: Software Engineering Principles.
2) CS798 Computer Science class at UW-La Crosse: Master's of Software Engineering Capstone Project.
3) IEEE Institute of Electrical and Electronics Engineers
4) Object-oriented design is a document specifying the objects to be implemented. The document will specify attributes (member variables) and behaviors (methods) for each object.

Revision History

Name:tomh
Date-Time:9/26
Description:initial draft
Version:1.01

Appendix D: Sample Object-Oriented Design

This appendix contains an Object-oriented design generated by RC, which derived from the SRS in appendix C. Some of the content has been removed from this document (all classes except one) to keep the size manageable for this manuscript.

Object-Oriented Design Document for: SimpleBank

Prepared for:
C-S 798 Software Development Project

Prepared by:
Tom Harron

Date/Time:
2/20/2006 10:40:25 AM

Table of Contents

1.0 Introduction

TODO: Review and complete this information!

This document describes an object-oriented design for... The requirements for this project are given in [1].

The object-oriented design is described using a collection of class definitions where each class definition includes structural and behavioral properties. The classes corresponding to functional behavior are described separately from the classes that describe the graphical user interface (GUI). In this way, a designer has the freedom to change the GUI without affecting the functional behavior.

2.0 Design Decisions

The following decisions are made during the design process.

TODO: List the design decisions for this project.

1.

3.0 UML Class Diagram

UML design diagram for RC project: SimpleBank

The following relationships are defined:

BankMain aggregates accountEntries

BankMain aggregates transactionEntries

TODO: Convert this information into a diagram.

4.0 Format of a Class Description

Each class definition is given in the following format.

Class Names

Must be unique within entire document.

Attributes (or instance variables)

Each attribute is given in the following format:

<modifier><data type><name>

where <modifier> is "public" or "private."

Methods

Each method will be classified as "public" or "private."

Each method is given in the following structural format:

Name of the method - must be unique within the class.

Synopsis of the method - calling syntax for the method.

Purpose of the method - a short description of the functionality implemented by this method.

Modifier (visibility) - public or private.

Input parameters - a set of parameters in <data type><name> format.

Output parameters - a set of parameters in <data type><name> format.

Local variables - a set of variables used in describing the pseudocode (given next) for this method.

Pseudocode - an algorithmic (structured) description of the method.

Exceptions - a set of exceptions that might arise in executing this method and their corresponding corrective actions.

Remarks - additional information about this method and hints for the programmers; typically, in may include the design decisions taken and choices of implementation that the programmer may consider.

5.0 Class Definitions

Class name: accountEntries

Attributes:

private int m_accountType

private date m_startingDate

private double m_startingBalance

private string m_firstName

private string m_lastName

private int m_accountNumber

private date m_closeDate

private double m_amount

Methods:

-------*-------*-------*--------*-------

Name: InitializeBank
Synopsis: public void InitializeBank()
Purpose: **TODO - fill this in**
Modifier (visibility): public
Input parameters:
Output parmaters:
Local variables: None
Pseudocode:

```
/**
* @author Tom Harron
* @version 1.0, 2/20/2006 10:40:25 AM
*
*
* @return void
*
*/
public void InitializeBank()
{

return;

} // InitializeBank()
```

Exceptions:
Remarks: None

-------*-------*-------*--------*-------

Name: WriteBank
Synopsis: public void WriteBank(final accountEntries m_accountEntries)
Purpose: **TODO - fill this in**
Modifier (visibility): public
Input parameters: final accountEntries m_accountEntries
Output parmaters:
Local variables: None
Pseudocode:

```
/**
* @author Tom Harron
* @version 1.0, 2/20/2006 10:40:25 AM
*
* @param accountEntries m_accountEntries (Func.Req.: Bank.2 - WriteBank, Input (line 1))
*
* @return void
*
*/
public void WriteBank(final accountEntries m_accountEntries)
{

    // TODO: Implement ACTIONS for attribute: m_accountEntries
    // Line 1) Write account entries to secondary storage.


     return;

} // WriteBank()
```

Exceptions:
Remarks: None

-------*-------*-------*--------*-------

Name: CreateAccount
Synopsis: public int CreateAccount(final int m_accountType, final date m_startingDate, final double m_startingBalance, final String m_firstName, final String m_lastName)
Purpose: **TODO - fill this in**
Modifier (visibility): public
Input parameters:

final int m_accountType

final date m_startingDate

final double m_startingBalance

final String m_firstName

final String m_lastName

Output parmaters:
Local variables: None
Pseudocode:

```
/**
* @author Tom Harron
* @version 1.0, 2/20/2006 10:40:25 AM
*
* @param int m_accountType (Func.Req.: Acct.1 - CreateAccount, Input (line 1))
* @param date m_startingDate (Func.Req.: Acct.1 - CreateAccount, Input (line 2))
* @param double m_startingBalance (Func.Req.: Acct.1 - CreateAccount, Input (line 3))
* @param string m_firstName (Func.Req.: Acct.1 - CreateAccount, Input (line 4))
* @param string m_lastName (Func.Req.: Acct.1 - CreateAccount, Input (line 5))
*
* @return int
*
*/
public int CreateAccount(final int m_accountType, final date m_startingDate, final double m_startingBalance,
final String m_firstName, final String m_lastName)
{

   // TODO: Implement ACTIONS for attribute: m_accountType
   // Line 1) Ensure account type is checking or savings.
   // Line 8) Ensure SSN does not exist in another account of same account type.
   // Line 11) Set account type.

   // TODO: Implement EXCEPTIONS for attribute: m_accountType
   // Line 1) Account type was not checking and was not savings.

   // TODO: Implement ACTIONS for attribute: m_startingDate
   // Line 2) Ensure starting date is valid.

   // TODO: Implement EXCEPTIONS for attribute: m_startingDate
   // Line 2) starting date.

   // TODO: Implement ACTIONS for attribute: m_startingBalance
   // Line 3) Ensure starting balance is >= $10.00.
   // Line 13) Set starting balance.

   // TODO: Implement EXCEPTIONS for attribute: m_startingBalance
   // Line 3) Starting balance if less than $10.00.
```

```
    // TODO: Implement ACTIONS for attribute: m_firstName
    // Line 4) Ensure first name is not blank.
    // Line 6) Ensure first name is not over maximum length limit.
    // Line 14) Set first name.

    // TODO: Implement EXCEPTIONS for attribute: m_firstName
    // Line 4) First name was blank.
    // Line 6) First name was over length limit.

    // TODO: Implement ACTIONS for attribute: m_lastName
    // Line 5) Ensure last name is not blank.
    // Line 7) Ensure last name is not over maximum length limit.
    // Line 15) Set last name.

    // TODO: Implement EXCEPTIONS for attribute: m_lastName
    // Line 5) Last name was blank.
    // Line 7) Last name was over length limit.

    // TODO: Implement ACTIONS for attribute: m_accountNumber
    // Line 10) Create an account number (object).

    // TODO: This method must return data type: int
    return; // specify return value of type int

} // CreateAccount()
```

Exceptions:

Remarks: None

-------*-------*-------*--------*-------

Name: CloseAccount
Synopsis: public void CloseAccount(final int m_accountType, final int m_accountNumber, final date m_closeDate)
Purpose: **TODO - fill this in**
Modifier (visibility): public
Input parameters:

final int m_accountType

final int m_accountNumber

final date m_closeDate

Output parmaters:
Local variables: None
Pseudocode:

```
/**
* @author Tom Harron
* @version 1.0, 2/20/2006 10:40:25 AM
*
* @param int m_accountType (Func.Req.: Acct.2 - CloseAccount, Input (line 2))
* @param int m_accountNumber (Func.Req.: Acct.2 - CloseAccount, Input (line 1))
* @param date m_closeDate (Func.Req.: Acct.2 - CloseAccount, Input (line 3))
*
* @return void
*
*/
public void CloseAccount(final int m_accountType, final int m_accountNumber, final date m_closeDate)
```

```
{

    // TODO: Implement ACTIONS for attribute: m_accountType
    // Line 2) Ensure account type exists
    // Line 6) Clear account type (closing this account)

    // TODO: Implement EXCEPTIONS for attribute: m_accountType
    // Line 2) Account type does not exist

    // TODO: Implement ACTIONS for attribute: m_accountNumber
    // Line 1) Ensure account number exists

    // TODO: Implement EXCEPTIONS for attribute: m_accountNumber
    // Line 1) Account number does not exist

    // TODO: Implement ACTIONS for attribute: m_closeDate
    // Line 3) Ensure valid close date
    // Line 4) Set account close date

    // TODO: Implement EXCEPTIONS for attribute: m_closeDate
    // Line 3) close date

    return;

} // CloseAccount()
```

Exceptions:

Remarks: None

-------*-------*-------*--------*-------

Name: ViewBalance
Synopsis: public void ViewBalance(final int m_accountType, final int m_accountNumber)
Purpose: **TODO - fill this in**
Modifier (visibility): public
Input parameters:

final int m_accountType

final int m_accountNumber

Output parmaters:
Local variables: None
Pseudocode:

```
/**
* @author Tom Harron
* @version 1.0, 2/20/2006 10:40:25 AM
*
* @param int m_accountType (Func.Req.: Acct.3 - ViewBalance, Input (line 2))
* @param int m_accountNumber (Func.Req.: Acct.3 - ViewBalance, Input (line 1))
*
* @return void
*
*/
public void ViewBalance(final int m_accountType, final int m_accountNumber)
{

    // TODO: Implement ACTIONS for attribute: m_accountType
    // Line 2) Ensure account type exists

    // TODO: Implement EXCEPTIONS for attribute: m_accountType
```

```
    // Line 2) Account type does not exist

    // TODO: Implement ACTIONS for attribute: m_accountNumber
    // Line 1) Ensure account number exists

    // TODO: Implement EXCEPTIONS for attribute: m_accountNumber
    // Line 1) Could not find account number

    return;

} // ViewBalance()
```

Exceptions:
Remarks: None

-------*-------*-------*--------*-------

Name: ViewTransactions
Synopsis: public double ViewTransactions(final int m_accountType, final int m_accountNumber)
Purpose: **TODO - fill this in**
Modifier (visibility): public
Input parameters:

final int m_accountType

final int m_accountNumber

Output parmaters:
Local variables: None
Pseudocode:

```
/**
* @author Tom Harron
* @version 1.0, 2/20/2006 10:40:25 AM
*
* @param int m_accountType (Func.Req.: Acct.4 - ViewTransactions, Input (line 2))
* @param int m_accountNumber (Func.Req.: Acct.4 - ViewTransactions, Input (line 1))
*
* @return double
*
*/
public double ViewTransactions(final int m_accountType, final int m_accountNumber)
{

    // TODO: Implement ACTIONS for attribute: m_accountType
    // Line 2) Ensure account type exists

    // TODO: Implement EXCEPTIONS for attribute: m_accountType
    // Line 2) Account type does not exist

    // TODO: Implement ACTIONS for attribute: m_accountNumber
    // Line 1) Ensure account number exists

    // TODO: Implement EXCEPTIONS for attribute: m_accountNumber
    // Line 1) Could not find account number



    // TODO: This method must return data type: double
    return; // specify return value of type double
```

```
} // ViewTransactions()

Exceptions:
Remarks: None
        -------*-------*-------*--------*-------
Name: DepositFunds
Synopsis: public void DepositFunds(final int m_accountType, final int m_accountNumber, final double
m_amount)
Purpose: TODO - fill this in
Modifier (visibility): public
Input parameters:
        final int m_accountType
        final int m_accountNumber
        final double m_amount
Output parmaters:
Local variables: None
Pseudocode:

/**
* @author Tom Harron
* @version 1.0, 2/20/2006 10:40:25 AM
*
* @param int m_accountType (Func.Req.: Trans.1 - DepositFunds, Input (line 2))
* @param int m_accountNumber (Func.Req.: Trans.1 - DepositFunds, Input (line 1))
* @param double m_amount (Func.Req.: Trans.1 - DepositFunds, Input (line 4))
*
* @return void
*
*/
public void DepositFunds(final int m_accountType, final int m_accountNumber, final double m_amount)
{

    // TODO: Implement ACTIONS for attribute: m_accountType
    // Line 2) Ensure account type exists

    // TODO: Implement EXCEPTIONS for attribute: m_accountType
    // Line 2) Account type does not exist

    // TODO: Implement ACTIONS for attribute: m_accountNumber
    // Line 1) Ensure account number exists

    // TODO: Implement EXCEPTIONS for attribute: m_accountNumber
    // Line 1) Could not find account number

    // TODO: Implement ACTIONS for attribute: m_amount
    // Line 4) Ensure amount is positive
    // Line 7) Set amount
    // Line 9) Update balance for account (add amount to balance)

    // TODO: Implement EXCEPTIONS for attribute: m_amount
    // Line 4) Amount is non-positive


    return;

} // DepositFunds()
```

Exceptions:
Remarks: None

-------*-------*-------*--------*-------

Name: WithdrawFunds
Synopsis: public void WithdrawFunds(final int m_accountType, final int m_accountNumber, final double m_amount)
Purpose: **TODO - fill this in**
Modifier (visibility): public
Input parameters:

final int m_accountType

final int m_accountNumber

final double m_amount

Output parmaters:
Local variables: None
Pseudocode:

```
/**
* @author Tom Harron
* @version 1.0, 2/20/2006 10:40:25 AM
*
* @param int m_accountType (Func.Req.: Trans.2 - WithdrawFunds, Input (line 2))
* @param int m_accountNumber (Func.Req.: Trans.2 - WithdrawFunds, Input (line 1))
* @param double m_amount (Func.Req.: Trans.2 - WithdrawFunds, Input (line 4))
*
* @return void
*
*/
public void WithdrawFunds(final int m_accountType, final int m_accountNumber, final double m_amount)
{

   // TODO: Implement ACTIONS for attribute: m_accountType
   // Line 2) Ensure account type exists

   // TODO: Implement EXCEPTIONS for attribute: m_accountType
   // Line 2) Account type does not exist

   // TODO: Implement ACTIONS for attribute: m_accountNumber
   // Line 1) Ensure account number exists

   // TODO: Implement EXCEPTIONS for attribute: m_accountNumber
   // Line 1) Could not find account number

   // TODO: Implement ACTIONS for attribute: m_amount
   // Line 4) Ensure amount is positive
   // Line 8) Set amount
   // Line 10) Update balance for account (subtract amount from balance)

   // TODO: Implement EXCEPTIONS for attribute: m_amount
   // Line 4) Amount is non-positive


   return;

} // WithdrawFunds()
```

Exceptions:
Remarks: None

-------*-------*-------*--------*-------

Name: TransferFunds
Synopsis: public void TransferFunds(final int m_accountType, final int m_accountType, final int m_accountNumber, final double m_amount)
Purpose: **TODO - fill this in**
Modifier (visibility): public
Input parameters:

final int m_accountType

final int m_accountType

final int m_accountNumber

final double m_amount

Output parmaters:
Local variables: None
Pseudocode:

```
/**
* @author Tom Harron
* @version 1.0, 2/20/2006 10:40:25 AM
*
* @param int m_accountType (Func.Req.: Trans.3 - TransferFunds, Input (line 2))
* @param int m_accountType (Func.Req.: Trans.3 - TransferFunds, Input (line 3))
* @param int m_accountNumber (Func.Req.: Trans.3 - TransferFunds, Input (line 1))
* @param double m_amount (Func.Req.: Trans.3 - TransferFunds, Input (line 5))
*
* @return void
*
*/
public void TransferFunds(final int m_accountType, final int m_accountNumber, final double m_amount)
{

   // TODO: Implement ACTIONS for attribute: m_accountType
   // Line 2) Ensure source account type exists
   // Line 3) Ensure destination account type exists
   // Line 6) Ensure source account type balance >= amount to be transferred
   // Line 7) Create transaction entry for source account type
   // Line 11) Update balance for account type (subtract amount from balance)
   // Line 12) Create transaction entry for destination account type
   // Line 16) Update balance for account type (add amount to balance)

   // TODO: Implement EXCEPTIONS for attribute: m_accountType
   // Line 2) Account number does not have specified source account type
   // Line 3) Account number does not have specified destination account type

   // TODO: Implement ACTIONS for attribute: m_accountNumber
   // Line 1) Ensure account number exists

   // TODO: Implement EXCEPTIONS for attribute: m_accountNumber
   // Line 1) Could not find account number
   // Line 2) Account number does not have specified source account type
   // Line 3) Account number does not have specified destination account type

   // TODO: Implement ACTIONS for attribute: m_amount
   // Line 5) Ensure amount is positive
   // Line 6) Ensure source account type balance >= amount to be transferred
   // Line 9) Set amount
   // Line 11) Update balance for account type (subtract amount from balance)
   // Line 14) Set amount
```

```
    // Line 16) Update balance for account type (add amount to balance)

    // TODO: Implement EXCEPTIONS for attribute: m_amount
    // Line 5) Amount is non-positive

    return;

} // TransferFunds()

Exceptions:
Remarks: None
            -------*-------*-------*--------*-------
Name: Assumption1
Synopsis: public void Assumption1()
Purpose: TODO - fill this in
Modifier (visibility): public
Input parameters:
Output parmaters:
Local variables: None
Pseudocode:

/**
* @author Tom Harron
* @version 1.0, 2/20/2006 10:40:25 AM
*
*
* @return void
*
*/
public void Assumption1()
{

    // TODO: Implement ASSUMPTIONS for attribute: m_accountType
    // Line 1) An account entry consists of an account number, account type, first name, last name, SSN, date
opened, date closed, balance and list of transaction entries.

    // TODO: Implement ASSUMPTIONS for attribute: m_firstName
    // Line 1) An account entry consists of an account number, account type, first name, last name, SSN, date
opened, date closed, balance and list of transaction entries.

    // TODO: Implement ASSUMPTIONS for attribute: m_lastName
    // Line 1) An account entry consists of an account number, account type, first name, last name, SSN, date
opened, date closed, balance and list of transaction entries.

    // TODO: Implement ASSUMPTIONS for attribute: m_accountNumber
    // Line 1) An account entry consists of an account number, account type, first name, last name, SSN, date
opened, date closed, balance and list of transaction entries.

    return;

} // Assumption1()

Exceptions:
Remarks: None
            -------*-------*-------*--------*-------
Name: Assumption6
Synopsis: public void Assumption6()
Purpose: TODO - fill this in
```

Modifier (visibility): public
Input parameters:
Output parmaters:
Local variables: None
Pseudocode:

```
/**
* @author Tom Harron
* @version 1.0, 2/20/2006 10:40:25 AM
*
*
* @return void
*
*/
public void Assumption6()
{

    // TODO: Implement ASSUMPTIONS for attribute: m_accountType
    // Line 6) A transaction entry consists of an account number, account type, amount, date, type of
transaction (deposit, withdraw or transfer.)

    // TODO: Implement ASSUMPTIONS for attribute: m_accountNumber
    // Line 6) A transaction entry consists of an account number, account type, amount, date, type of
transaction (deposit, withdraw or transfer.)

    // TODO: Implement ASSUMPTIONS for attribute: m_amount
    // Line 6) A transaction entry consists of an account number, account type, amount, date, type of
transaction (deposit, withdraw or transfer.)

    return;

} // Assumption6()
```

Exceptions:
Remarks: None

-------*-------*-------*--------*-------

Name: Assumption8
Synopsis: public void Assumption8()
Purpose: **TODO - fill this in**
Modifier (visibility): public
Input parameters:
Output parmaters:
Local variables: None
Pseudocode:

```
/**
* @author Tom Harron
* @version 1.0, 2/20/2006 10:40:25 AM
*
*
* @return void
*
*/
public void Assumption8()
{

    // TODO: Implement ASSUMPTIONS for attribute: m_amount
    // Line 8) The minimum transaction (deposit, withdraw, transfer) amount must be > $0.00.
```

```
    return;

} // Assumption8()

Exceptions:
Remarks: None
            -------*-------*-------*--------*-------
```

6.0 References

This section list documents referenced earlier in this document.

1. Tom Harron, "Software Requirements Specification (SRS) for SimpleBank", 2/19/2006 9:39:09 AM.

TODO: List any other documents referenced by this object-oriented design document.

7.0 Appendix A

TODO: You may optionally place additional information about design in this appendix. Information such as external file formats, configuration information, etc.