Softstar Research, Inc.

Requirements Artifacts

Revision History

Date	Version	Description	Author
	<1.0>	Initial draft	David Rubin

Table of Contents

INTRODUCTION

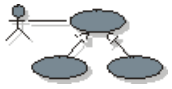
This document provides an overview of artifacts considered 'requirements' within the Softstar Research engineering environment. Actual requirements for any particular project will leverage one or more of these artifacts when being formally documented within an elaboration phase.

There are seven primary requirements artifacts, which are:

Glossary. The glossary is a repository of common terms and definitions used throughout the project. There should be only one glossary for the project.

Actor Inventory. This is a list containing all of the actors and their definitions for the system. An actor defines a coherent set of roles that users of the system can play when interacting with it. A user can either be an individual or an external system or entity (i.e. clock).

Use Case Diagrams. The use case diagram is a model of the system's intended functions and its environment, and serves as a contract between the customer and the developers.



Use Case. A use case defines a set of use case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor.

Storyboard. A storyboard is a logical and conceptual description of how a use case is provided by the user interface, including the interaction required between the actor(s) and the system.

User Interface Prototype. The user interface prototype will enable the customer to visualize how the system will work. The aim is not to create a product, but to demonstrate certain properties of the system.

Supplementary Specifications. The Supplementary Specifications capture the system and business requirements that are not captured in the use cases, use case model or other requirement artifacts. These will include separate documentation for Business Rules, as well as User Interviews, Use case Notes, and GUI metaphor [look and feel].

GLOSSARY

Introduction

This document is used to define terminology specific to the problem domain, explaining terms that may be unfamiliar to the reader of the use case descriptions and other project documents.

The glossary may be used as an informal data dictionary. Data definitions can be captured so that use case descriptions and other project documents can focus on what the system must do with the information, rather than defining terms.

There will only be **one** Glossary for the project. This will ensure that the terms are defined in such a way that they are applicable across the project. If multiple glossaries are created, often a term may be defined more than once with slightly different definitions, or multiple terms may be used for the same definition.

Symbol

Properties

Entries in the Glossary will contain at a minimum the following fields:

- **Term** The word or phrase for which a glossary entry is being created.
- **Category** The category to which the term belongs. This will make it easier to search for or locate a term.
- **Definition** The detailed definition of the term.
- **Acronym** The acronym used for this term within Softstar.
- **Found on** The date on which the term was found.
- **Found by** The name of the person who found the term (if applicable).
- **Found in** The document or source in which the term was found (if applicable).

ACTOR INVENTORY

Introduction

The actor inventory is a listing of all the actors participating in the system. This inventory will contain an entry for each type of actor.

An actor defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor has one role for each use case with which it communicates.

Actors are different from users. The user is the actual person who uses the system, whereas an actor represents a certain role that a user can play. An actor is regarded as a class; the user is an instance of that class. An example of a 'non-user' actor often defined is the actor 'time'.

A user may play the roles of several different actors. For example, a system may have the actor's Clerk and Pilot. A user may sometimes act as a pilot and sometimes as a clerk. A user will perform different use cases depending upon the role being played.

Symbol

Properties

Entries in the Actor Inventory can contain the following fields:

- **Name** Name of the actor.
- **Description** The description of the actor's sphere of responsibility, and what the actor needs and/or will use the system for.
- **Characteristics** For human actors the following details should be captured:
  - The physical environment of the actor
  - The number of users the actor represents
  - The actor's level of domain knowledge
  - The actor's level of computer experience
  - Other applications the actor is using
  - Other general characteristics

USE CASE DIAGRAMS

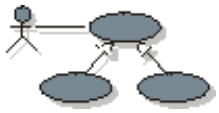
Introduction

The use Case diagram is a graphical representation of the actors and their interactions with the system, via graphical objects representing the Use Cases. The diagrams show actors and use cases together, along with their relationships.

The use case model is a model of the system's intended functions and its environment, and serves as a contract between the customer and the developers.

Use case diagrams are often equated to 'Level 0 Context Diagrams' in the Information Engineering world. Level 0 context diagrams often represent a high-level view of the system as a 'box' that external entities interact with.

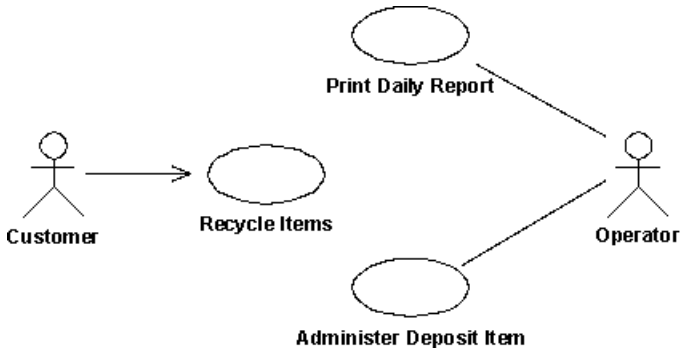
Symbol



Explanation

A use case model is a model of the system's intended functions and its surroundings, and serves as a contract between the customer and the developers. Use cases serve as a unifying thread throughout system development.

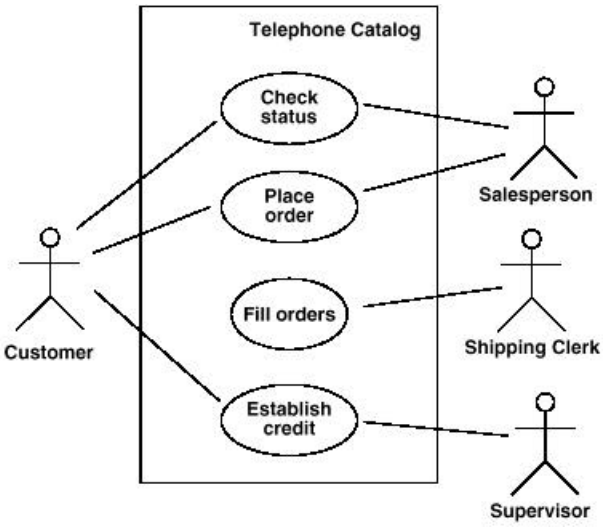
The diagram below shows a part of a use case model for the Recycling-Machine System.



- A use case diagram, showing an example of a use case model with actors and use cases.

There are many ways to model a system, each of which may serve a different purpose. However, the most important role of a use case model is to communicate the system's behavior to the customer or end user. Consequently, the model must be easy to understand.

The users and any system that may interact with the system are the actors. Because they represent system users, actors help delimit the system and give a clearer picture of what it is supposed to do. Use cases are developed on the basis of the actors' needs. This ensures that the system will turn out to be what the users expected.



- A second use case diagram example

USE CASE

Introduction

A use case is a requirements document, which describes what the system does for a user. All of the use cases taken together completely describe the system from the user's perspective. Users, analysts and developers should be able to understand and agree on what the system does by reading the use cases.

An example of a use case might be "Add Linked Account" which describes how a user uses the system to add a linked account to their portfolio.

Symbol

Definition

The classic definition of Use Cases comes from Ivar Jacobson's *'Object-Oriented Software Engineering' (*OOSE) which states: "A Use Case is a sequence of transactions in a system, whose task is to yield a measurable value to an individual actor of the system."

Here is the definition stated a little differently, with explanation:

The Rational Unified Process defines two key concepts: use case and actor:

A *use case* is a sequence of actions a system performs that yields an observable result of value to a particular actor.

An *actor* is someone or something outside the system that interacts with the system.

Therefore, we have the system under consideration, which is surrounded by actors (people or other systems) that interact with it, and we have use cases that define these interactions.

In reviewing these definitions you should consider several key items.

Actions
An action is a computational or algorithmic procedure that is invoked when the actor provides a signal to the system or when the system gets a time event. An action may imply signal transmissions to either the invoking actor or other actors. An action is atomic, which means it is performed either entirely or not at all.

A sequence of actions

The sequence referred to in the definition is a specific flow of events through the system. Many flows of events are possible, and many of them may be very similar. To make a use case model understandable, you group similar flows of events into a single use case.

The system performs

This means that we are concerned with what the system does in order to perform the sequence of actions. The use case helps us to define a firm boundary around the system and what it does to separate it from the outside world. In this way, it helps us bound the scope of the system.

An observable result of value
The sequence of actions must yield something that has value to an actor of the system. An actor should not have to perform several use cases in order to achieve something useful. Focusing on useful value provided to an actor ensures that the use case has relevance and is at a level of granularity that can be understood by the user.

A particular actor

Focusing on a particular actor forces us to isolate the value provided to specific groups of users of the system, ensuring that the system does what they need it to. It avoids building use cases that are too big. It also prevents us from losing focus and building systems that try to satisfy the needs of everyone but in the end satisfy no one.

The description of a use case defines what happens in the system when the use case is performed. The functionality of a system is defined by a set of different use cases, each of which represents a specific flow of events.

To define Use Cases from a different perspective, there's a technique taught many years ago regarding the design of software systems. This technique was *to "Write the users manual"*, before designing the actual program. With that technique in mind, Use Cases can be thought of as a technique for writing the users manual of a system, before actually designing the system itself.

Further material can be found in the white paper *'Uses of Use Cases'* available at www.softstar-inc.com.

Guidelines

As a general guideline, there should be about 25 use cases for any large-scale application. More than 30 means that the Use Cases are too detailed, fewer than 10 probably means that some major part of the application has been missed. As far as detail within the Use Case, a good starting point is that a Use Case should be about three pages long.

When describing a Use Case, always try to keep in mind the distinction between the 'user's' problem and context, and the systems implementation domain. In other words, keep implementation details out of the Use Case. Implementation details discovered during this phase should be documented in the use case notes at the end of each use case. As use cases are refined, this detail may be pulled out and placed in other more relevant documents.

Additional details will also be discovered and documented separately. For example, business rules and issues will probably be discovered during the requirements and should be documented in the Business Rules document and also the Issues Database.

The level of detail will evolve over the life of the project. Use Cases begin as outlines with little detail. Later, the use cases will be reorganized to pull out commonalities. Finally detail will be added. Some Use Cases may be too simple to warrant much detail and may remain outlines provided that they are clear enough for users, analysts and developers.

Properties

ID	A unique identifier for the Use Case.
TITLE	The title of the use case, usually no more than a few words long. Should be in form of Verb-Object.

LAST UPDATED DATE	Date the Use Case was last updated.
IDENTIFIED BY	Name of person/analyst who originally identified the use case.
PROCESSES	Lists any processes, which use this use case to automate some steps in the process.
SUMMARY	A concise paragraph that defines the nature of the 'Use Case'. The summary may also include a section with three points, and can be documented as three separate bullet items. The three points are the trigger, the action, and the result. These points can be stated using the following templates for sentences: This Use Case begins when … This Use Case does … This Use Case concludes (or ends) when … Sentences within the summary generally contain information about time sequences, actors, actions and constraints.
ACTORS	A list of one or more actors participating in this Use Case. These actors should appear in the Actor Inventory.
ASSUMPTIONS	Any assumptions made in researching and documenting this Use Case.
CONSTRAINTS	Statements that affect or modify the system's behavior. List any business rules which constrain the running of the Use Case or which constrain the results of the Use Case.
BASIC COURSE	An ordered list of steps taken by actors and responses by the system. This is

OF ACTION	the normal sequence of steps usually taken in the Use Case. While this can contain optional steps, this should describe what the actors do most frequently. Variations and exceptions should be documented as alternate courses. Actions are generally described from the actors' perspective, not the analysts' perspective. Try to avoid any details that are implicit implementation considerations. Each step in the basic course should be numbered, so that the steps can be referenced easily elsewhere.
ALTERNATE COURSES OF ACTION	Any alternate set of steps that are exceptions to the basic course. The name of an alternative course should identify where and why the branch occurs. The "where" refers back to a numbered action in the basic course, the "why" describes the situation that causes the actor to take the alternate course? E.g. "Alternate 4A, Linked Account Institution does not exist", says that the alternate course branches after step 4 when the institution a user selects does not exist. In general, alternate courses will not reenter the basic course. Once an alternate course is started, the user is typically committed to finishing the alternate course. If an exception to this is identified, it should be discussed with the lead analyst and designers. Alternate courses of action can be a mechanism to reduce the total number of Use Cases defining a system.
USE CASE NOTES	Any extra or supporting information about the use case.

STORYBOARD

Introduction

A **use case storyboard** is a logical and conceptual description of how a use case is provided by the user interface, including the interaction required between the actor(s) and the system.

Use case storyboards are used to understand and reason about the requirements of the user interface, including usability requirements. They represent a high-level understanding of the user interface, and are much faster to develop than the actual user interface. The use case storyboards can thus be used to create and reason about several versions of the user interface before it is prototyped, designed, and implemented.

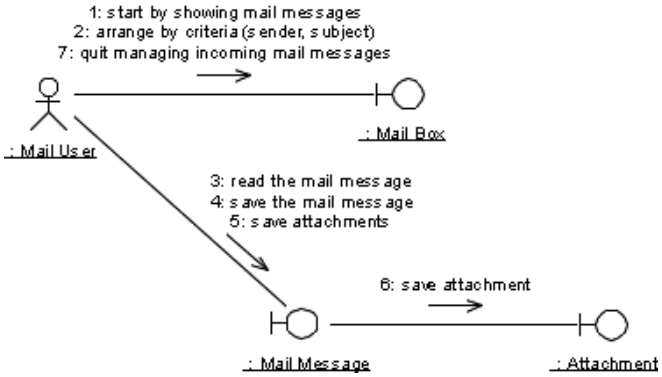
Normally, a use case storyboard is described in terms of boundary classes and their static and dynamic relationships, such as aggregations, associations, and links [i.e. a collaboration diagram]. Each boundary class is in turn a high-level representation of a window or similar construct in the user interface.

For Softstar, storyboards can either be in the format of a collaboration diagram, or they can take a form that would represent a high-level perspective of the user interface element [i.e. screens], and the association between the user interface elements to satisfy a particular Use Case.

Symbol

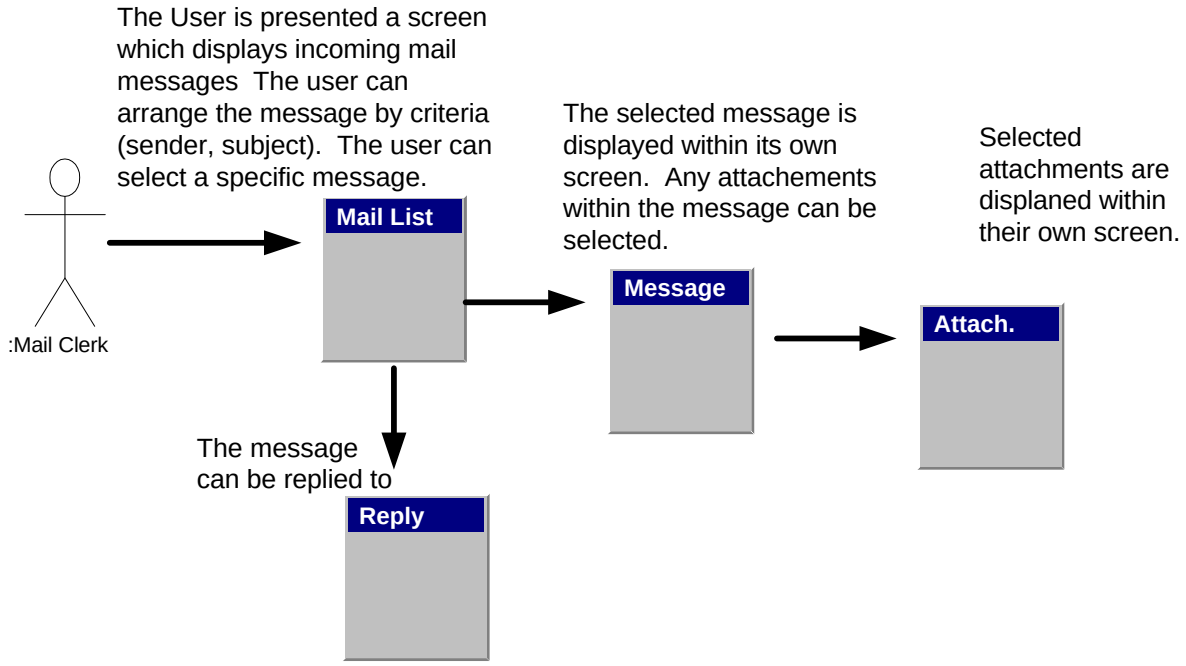
Example Storyboard 1

The following is a collaboration diagram owned by the use case storyboard corresponding to the Manage Incoming Mail Messages use case.



Example Storyboard 2

The following is an example of a storyboard representing User Interface elements and association flows between them to satisfy a Use Case.



USER INTERFACE PROTOTYPE

Introduction

The **user interface prototype** will enable the customer to visualize how the system will work. The aim is not to create a product, but to demonstrate certain properties of the system. The intention of the prototype may be to prove out technical or functional decisions made in the project.

The prototype can serve as a means of communication between the customer and the developer. It is much easier to express a view about something tangible – something that can be demonstrated and used, rather than express an opinion about a specification. A specification cannot capture the dynamics of a system in the same way as a working prototype.

If your prototype becomes your application, one of them is wrong!

Only a small part of the system will be used for the prototype and in the majority of cases the prototype is throwaway.

The prototype can address questions such as:

- What do the screens look like?
- What is the guiding metaphor for navigation through the application?
- What are the common widgets/buttons the user would like to see to perform operations and/or commands?

Symbol

SUPPLEMENTARY SPECIFICATIONS

Introduction

The Supplementary Specifications capture system requirements that are not captured in use cases, the use case model, or other requirement's documents. These will include separate documentation and other information such as User Interviews, Business Rules, and the GUI look and feel (GUI metaphor).

User Interviews

Any documentation created from meetings with the users of the system should be included as 'reference material' in the supplementary specification section of the requirement's document.

Use Case Notes

It's inevitable that when capturing information for a Use Case, that non-'Use Case' types of information will be discussed (i.e. implementation details, or algorithmic equations). While this information does not belong within the Use Case description, it is important information regarding the 'user' requirements of the system and should not be discarded.

Therefore, a 'use Case Notes' document is created when information is identified that is relevant to the system requirements, but not relevant to the use case being described. Often this document is in the form of notes, which in the header reference the Use Case that generated the information.

Business Rules

Overview

Business rules are statements that describe what is allowed or not allowed in the business. They lay out the boundaries and structure of the business. They constrain and influence the business itself, rather than define how the business is conducted. Business rules derive from the policies of the business. When rules are clearly stated, the policies become explicit and available.

Different groups have different needs for and different ways of looking at business rules. Business experts use business rules to define their business logically. Analysts use business rules to define the system. Designers map business rules to the parts of the system, which will implement the rules. Developers use business rules as specifications for the code, which implements the rules.

These different perspectives are managed by capturing business rules in one form, then transforming them, step by step, into executable code. The initial form for business rules should be clear, unambiguous and understandable for business experts and analysts. Each transformation after that needs to be clear enough for the groups that needs to use the rules in that form.

Definition

A business rule is "a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behavior of the business." [GUIDE Business Rule Project, Final Report, 1995]

Business rules are declarative statements: they assert facts. They do not describe when facts are asserted or in what order. The following is an example of a business rule as declarative statement:

- If a client does not sign [electronically] the NASDAQ subscriber agreement, then the client is not eligible for real-time quotes.

It is often tempting to write business rules as procedures or recipes. This temptation must be resisted to keep rules independent from each other. However, it is often useful to clarify rules with a procedure.

Documenting Business Rules

Business rules are documented in Word documents using the template, presented in a later section. When a business rule is identified, the Id, Name, Category, Type, Abstract, Analyst and Date should be filled in and the document checked into version control. As the rule is detailed, the document will be updated and expanded.

Properties

The following properties of business rules are captured during the requirements definition phase:

- **Id** A unique identifier for the business rule which can be used in processes or use cases to refer to the rule.
- **Name** Name of the business rule. Should describe briefly what the rule does.
- **Category** Multiple keywords which categorize the rule in the business.
- **Type** The type of rule.
- **Abstract** The concise, unambiguous statement of the business rule.
- **Processes** Processes, which use the business rule.

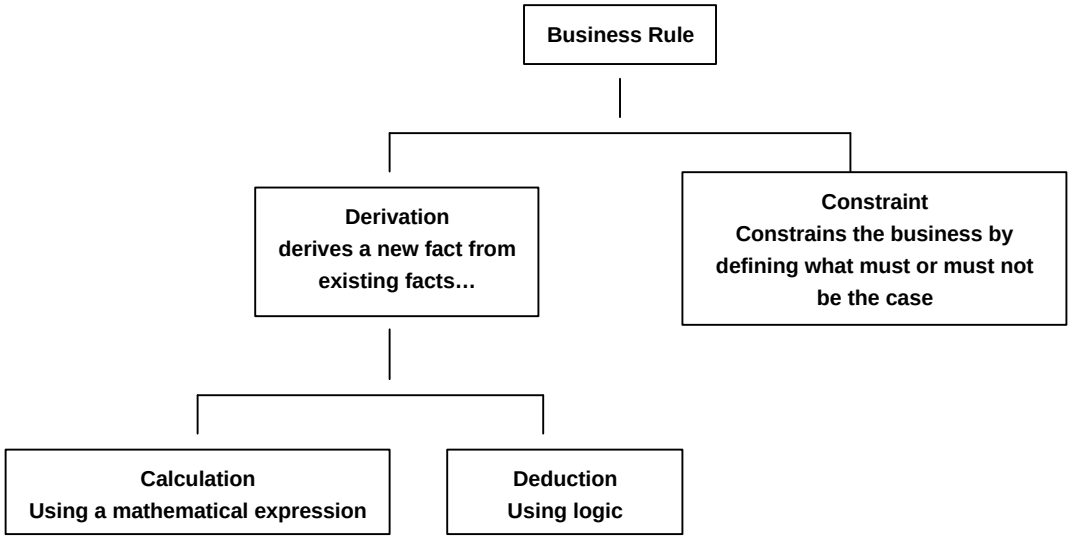
- **Use Cases** Use cases, which use the business rule.
- **Analyst** The analyst who found the business rule.
- **Date Found** When the rule was found.
- **Status** That current status of the rule.
- **Source** Where the rule was found.
- **Stability** How frequently the rule changes.
- **Policy** Policies from which this rule derive.
- **Clarification** Examples and explanations of the rule. As many examples as are necessary to show how the rule is used in the business. Explanations can be in the form of procedures with references to other rules. Note that this is not the implementation of the rule; it is to help understand the rule.

Types of Business Rules

Business rules are classified to ensure that the architecture of the system has conceptual integrity. The correct classification of a rule helps understand the rule's purpose. This helps determine the language of the rule and also allows different people to come to an understanding of what the rule does, more quickly. This classification can also be used by designers to map the rule to the system's design.

The current scheme for classifying business rules has two main types: Derivations and Constraints. There are two subtypes of Derivations: Calculations and Deductions. (See picture) This yields three types of business rules:

- **Calculation** Derives a fact about the business using a mathematical expression.
- **Deduction** Derives a fact about the business using logic.
- **Constraint** Constrains the business by defining what must or must not be the case.



Guidelines

Natural language should be used in the abstract and clarification of business rules. Terms in a rule statement should come from the glossary. As rules are written, the glossary should be updated with the terms used.

As rules are structured, more formal language will be used to make the rules completely unambiguous. This formal language will allow designers and developers to implement the rules as intended.

Template and Examples

TBD

GUI Expectations

This defines the User expectations of the GUI that the actors will interact with. These expectations should include the overall system metaphor(s), specifically 'look and feel' and 'flow control'.

Non Functional Requirements

Overview

This defines any non-functional requirements identified that aren't specified in the other requirements' artifacts. Non-functional requirements specify properties of the system, whereas functional requirements specify what the system does. Examples include requirement's identifying performance, scalability, reliability, operational requirements, technical implementation specifics, and so on.

Properties

- **Name** Unique, descriptive name of the requirement.
- **Type** Type of non-functional requirement. Current list of types includes…
  - **Implementation** How the system must be implemented.
  - **Interface** How users are able to interact with the system.
  - **Performance** How fast, how much, how frequently…
  - **Physical** Where or on what something must be done.
  - **Reliability** How robust the system is to errors and failures.
  - **Supportability** How easy the system is to keep running.
  - **Usability** How easy the system is to use.
- **Description** A statement of the requirement.
- **Found On** Date on which requirement was found.
- **Found By** Name of analyst who found the requirement.

Appendix

Requirements Characteristics [Individual]

In the article “Writing Quality Requirements”, Karl Wiegers writes that individual requirements statements should exhibit six characteristics. It’s these characteristics that often differentiate good requirements from problematic requirements. The following list summarizes these six good characteristics.

Characteristic 1: They must be correct.

Each requirement must accurately describe the functionality to be delivered. The reference for correctness is the source of the requirement, such as an actual customer or a higher-level system requirement specification. Any software requirement that conflicts with a corresponding system requirement is not correct. A developer that caught saying [or thinking] “That doesn’t make sense. This is probably what this requirement meant.” Is exhibiting a response to an incorrect requirement.

Characteristic 2: They must be feasible.

The requirement being defined must be implimentable within the known capabilities and limitations of the system and it’s environment. Developers can often provide a final sanity Q/A check on the technical feasibility of a requirement.

Characteristic 3: They must be necessary for the project.

Each requirement should document something that the customers [users] actually need, or something that is required to conform to an external system, an external interface, or an external standard. Unnecessary requirement statements are the leading causes of ‘scope creep’. A requirement that is ‘necessary’ should be able to be linked back to the business process that identified the need for the requirement.

Characteristic 4: They must be prioritized.

Each requirement should have an implementation priority associated with the requirement. If all requirements are not prioritized [or of equal priority], there’s no way to react to changes and/or additions during the life of the project. In addition, it is often the priority of the requirements that helps define the iteration plan for the development of the system.

Characteristic 5: They must be unambiguous.

The reader of any requirements should only draw one conclusion from it. In addition, multiple readers of the requirements should arrive at the same interpretation of the same requirement. Natural language is highly prone to ambiguity. To minimize this effect, subjective terms should be avoided when possible. These include terms like user-friendly, several, state-of-the-art, improved, maximize, and minimize. These types of term often violate characteristic # 6, that of being verifiable.

Characteristic 6: They must be verifiable.

There should be a way to derive testes or use other verification mechanisms, such as inspection or demonstration, to verify the implementation of a requirement. If there isn’t a quantifiable way of verifying a requirements, determining whether it was implemented correctly is high subjective [a matter of opinion].

Requirements Characteristics [Combined]

The combination of all of the artifacts identified in this document make up the complete Software Requirements Specification. This complete specification should exhibit the following four characteristics.

Characteristic 1: It is complete.

The specification shouldn't be missing any requirements. In other words, if it isn't written down as a requirement within the context of the requirement specification, it isn't getting done.

Characteristic 2: It is consistent.

Consistent requirements do not conflict with other requirements. Any disagreements about a requirement, or any conflicts between two or more requirements should be addressed early in the development life cycle [versus later]. Often, inconsistent requirements are caused by a specific requirement being referenced multiple times in different contexts.

Characteristic 3: It is modifiable.

The complete requirement specification must be able to be revised and traced. An automated way [both through procedures and tools] must be established to provide both history and trace-ability of requirements. It's inconceivable to believe that the requirements of the project won't change from initial inception through development to deployment.

Characteristic 4: It is traceable.

When we're finished, we should be able to link every software requirement to it's source, down through the design and coding that implemented the particular requirement. The characteristics identified in appendix A especially apply to these types of requirements.