

Software Requirement Specification ("Skill-Driven") - Form, Content and Guidelines

By Avner Ben
Draft #7

Contents

Software Requirement Specification ("Skill -Driven") - Form, Content and Guidelines ...	1
Introduction.....	2
SRS Objectives	3
Position in project life cycle	4
Document Validation	5
Document structure	6
Definitions.....	8
Appendices.....	10
Appendix 1. Software Development Life Cycle (Dataflow)	11
Appendix 2. Form	13
Appendix 3. Methodical guidelines	14

Introduction

1. **Product Name:** SOFTWARE REQUIREMENT SPECIFICATION *DOCUMENT*.
2. **Acronym:** SRS.
3. **Scope.** The SRS document is the sole product of the “Requirement Analysis” function in the software development life cycle.
These “SRS guidelines” describe (1) the objectives of the SRS document, (2) its internal structure, (3) the methodology to guide its preparation process and (4) its interfaces with neighboring document products. The latter are three: Base project specification (e.g. MRD), (2) the TLD document and (3) the PM document. These terms are explained below.
4. **Problem statement:**
 - 4.1. In practice, the software development process produces and consumes a number of documents, most notably – MRD, SRS, TLD, PM and STD. At present, there is much duplication of information among these documents, and it is not kept up-to-date. E.g. SRS documents duplicate use cases from the MRD.
 - 4.2. Various SRS documents demonstrate conflicting understanding of the difference between “functional” and “non-functional” requirements.
 - 4.3. SRS documents do not adhere to the organizational SRS template (or other standard), even if available.
 - 4.4. There is no formal definition for the interface between the SRS and neighboring documents.
 - 4.4.1. There is no convention as to which requirements to repeat, which to reference and which to safely ignore.
 - 4.4.1.1. SRS documents do not reference specific items in their base documents. There is only a general reference to the entire document.
 - 4.4.2. There is no convention as to which items belong in the SRS function and which belong in the High-Level Design functions.
5. **Authors.**
The SRS document is prepared by programming personnel, typically professional team leader and/or a specialized software architect. Often, these are also the people that will design the software (see TLD) and implement it.
6. **Audience.**
The SRS is based upon a MRD (or other document), is used to prepare the TLD document and supplies information for the preparation of the STD document. It is expected to be critically reviewed by the authors of these documents.

SRS Objectives

1. To terminate the functions of requirement collection and start the functions of production.
This means in particular:
 - 1.1. To concentrate all requirements of the software product in a repository that will effectively buffer the software development team from the product owner.
 - 1.2. To facilitate withstanding sensible requirement extension and changes without a major rewrite. The methodology and structure of the SRS must be robust, built to meet extension and change.
 - 1.2.1. *On the one hand*, the product owner is expected to make all effort to produce a *healthy requirement specification* and, *on the other hand*, the SRS authors must iterate on extracting and understanding the requirements, returning to the product owner as many times as it takes to ensure completeness and coherence of the SRS.
 - 1.2.2. Experience shows that most requirement changes only met during hard-core development could be prevented by methodical analysis of the initial requirements. Contrary with common programmer belief, only a small part of destructive requirement changes are indeed the result of completely unforeseen commercial events. Research (e.g. the "[CHAOS](#)" survey of 1995 by Boehm et al) indicates that as much as 80% of reported software project failures are due to *incomplete understanding* of the problem domain!
2. To assess the feasibility of the requirements, given platform and current software and hardware technology.
 - 2.1. This is the last stop, in the development process, for assessing the feasibility of the requirements without causing *physical* damage to the product!

Position in project life cycle

See also Appendix – Software Development Life Cycle Dataflow

1. Inputs.

The SRS is based upon one or more historically -preceding documents, from which it *extracts the base requirements* – functional and non-functional. This “Product Requirement Specification” document is prepared by the *product owner*, typically by non-programmers.

1.1. We will refer to this document(s) as the “Base Document”.

1.1.1. An example Base Document in the world of telephony is a “UI” document, typically featuring *use cases*.

1.1.2. An example Base Document in the world of gaming is a “Game Flow” document, typically featuring *flow charts*.

1.2. The SRS is formally based upon the Base Document, i.e. it was conceived (in the greater part) by its analysis and it *contains references* to it, which must be kept up-to-date.

1.2.1. Administrative measures must be taken to facilitate this requirement (or the SRS effort is rendered useless).

1.3. In the absence of a product-owner-supplied base document, a substitute may be prepared by the SRS authors.

1.3.1. Caution is required to prevent such document from becoming prematurely programmatic. Pure requirement definitions are best written by the product owner or representatives.

1.4. For the SRS author, the base document is reduced to so many functional requirement items to reference (e.g. “use cases” and “user-interaction flow charts”).

1.4.1. The items must be methodically sound, i.e. *allow exhaustive formal analysis*.

1.4.2. The items must be organized in a numbering system.

1.4.2.1. The numbering system must be reasonably stable, so that updating the base document does not invalidate all references to it in the SRS.

2. Outputs.

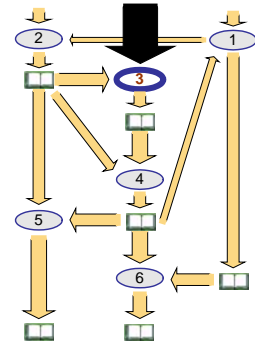
2.1. The SRS is the main source for compiling the TLD (Top-Level Design) document.

2.1.1. The TLD mainly translates the SRS into a programmable model.

2.1.2. The SRS should effectively buffer the TLD authors from direct contact with the product owner. Building the program model is thus reduced to the mechanical task of *finding the best programmatic representation* for the problem domain entities and their required skills. This is typically reduced to deciding on software architecture in general and design patterns and performance optimizations in particular.

2.2. The authors of the STD (Software Test Document) draw on the non -functional requirement list, found in the SRS.

2.3. *Warning!* Although the SRS *seems* to provide information of value to PM (Project Management – i.e. functional requirement priorities) and STD (Software Test Document – i.e. use case references), this information is not useful at neither SRS production time, nor the SRS level of abstraction. The authors of these documents must wait for the TLD to process the information to useful precision.



Document Validation

1. Cross-reference

- 1.1. The SRS must refer to all *major items* (e.g. *use cases, flow charts*) in its declared base document(s).
 - 1.1.1. The classification of major and minor, in this respect, is subjective (to the SRS authors).
 - 1.1.2. Nevertheless, the SRS does not have to refer to all the items inside the above.
- 1.2. All *required entities* in the SRS must be mapped to the TLD as software entities, (e.g. “classes” - but not on a one-to-one basis).
- 1.3. All *functional requirements* in the SRS must be mapped to the TLD (as either software skills or entity definitions).
- 1.4. The SRS is not required to do anything with the list of non -functional requirements (e.g. cross-reference them with other requirements) – just collect them carefully.

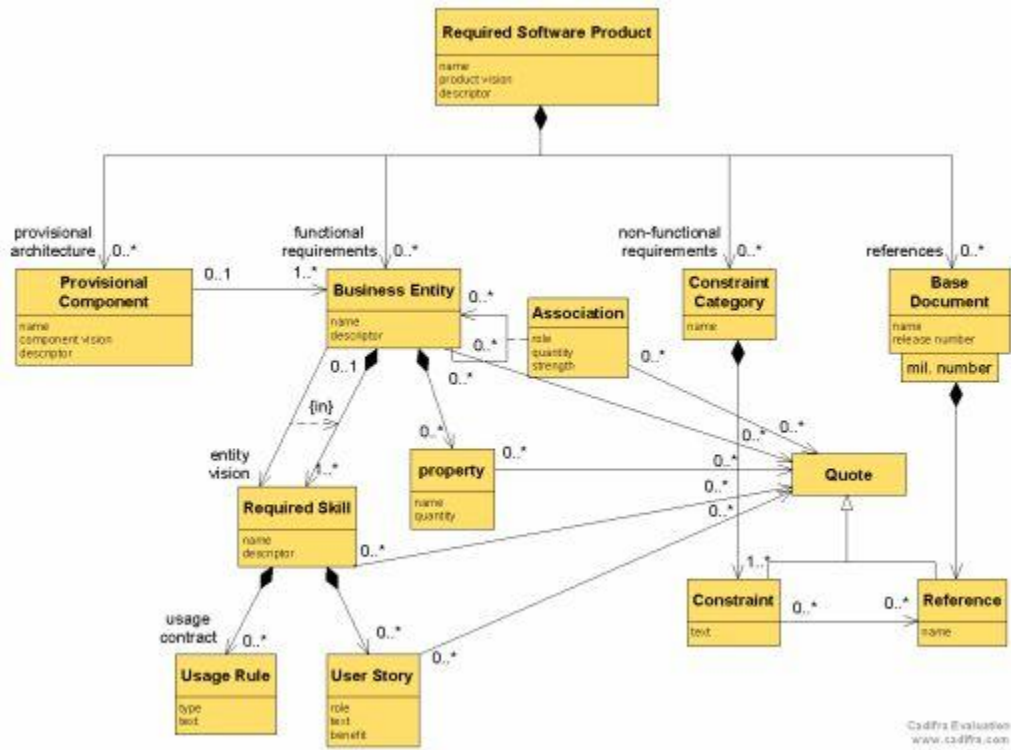
2. Approval

- 2.1. The SRS is inspected by the chief software architect, who highlights erroneous points and their risk factor: inconsistencies, missing information, obscurities, logical errors, etc.
- 2.2. The inspected SRS is approved (or rejected) by Project Management, considering the risk factors.

Document structure

1. **Use case references.**
 - 1.1. This chapter consists of a *list of references* to base document use cases.
 - 1.1.1. (The term is used here generically. Referenced items may also be flowcharts, etc.).
 - 1.1.2. A use case is uniquely identified by SRS name (or placeholder) and detailed by document name and version, hierarchy number and original name.
 - 1.1.2.1. The SRS name may be the same as the original name. The objective is to protect the SRS from minor base document numbering/naming changes.
2. **Deliverables.** Unstructured list of tangible products to deliver (e.g. code, user manual, etc.).
3. **Functional requirements** (*see item*).
 - 3.1. **Statement of product vision** (*see item*).
 - 3.2. List of **required entities** (*see item*).
 - 3.2.1. All functional requirements in the SRS must be organized under *entities*, one-to-many.
 - 3.2.2. An entity is identified by a unique name, features entity vision (a required skill) and (optionally) refers to the base document.
 - 3.2.3. An entity is defined by a list of **required skills** (*see item*) that detail its vision.
 - 3.2.3.1. A required skill is identified by a unique name, (optionally) has a description and (optionally) refers to the base document.
 - 3.2.3.1.1. The required skill name follows the required skill **form** (*see item*).
 - 3.2.3.1.2. A required skill description consists of free-form descriptor and/or **usage contract** (*see item*).
 - 3.2.3.1.3. A required skill may specify constraints (*see item*), where not obvious.
 - 3.2.3.1.4. A required skill may be strengthened by **user stories** (*see item*).
 - 3.2.4. An entity is (optionally) required to manage a list of **properties** (*see item*).
 - 3.2.4.1. A property is identified by name (unique in the entity) and (optionally) has a description and (optionally) value range.
 - 3.2.5. An entity is (optionally) required to manage a list of **associations** (*see item*).
 - 3.2.5.1. An association is identified by **target role** name (unique in the entity), association quantity (*see item*) and association strength (*see item*).
 - 3.2.4. An entity is (optionally) required to manage a list of **properties** (*see item*).
 - 3.2.4.1. A property is identified by name (unique in the entity) and (optionally) has a description and (optionally) value range.
 - 3.2.5. An entity is (optionally) required to manage a list of **associations** (*see item*).
 - 3.2.5.1. An association is identified by **target role** name (unique in the entity), association quantity (*see item*) and association strength (*see item*).
4. **Non-functional requirements.** Ordered list of **constraint categories** (*see item*), each consisting of an ordered list of **constraints** (*see item*).
5. **Provisional division to components.** List of proposed software components.
 - 5.1. *For each proposed component:*
 - 5.1.1. Component vision (a required skill).
 - 5.1.2. (Optionally) “Façade” entity.
 - 5.1.3. List of required entity references.
 - 5.1.4. Optionally (where possible and desirable), the very document hierarchy may consist of components, with entities festered below.

Class diagram:



Definitions

1. **Functional requirement:** A requirement of the software that suggests a *discrete action* by the software on the platform.
 - 1.1. Each functional requirement is traceable to exactly one code entry point.
 - 1.2. Functional requirements are expressed by required *skills*.
 - 1.2.1. Skills are grouped under entities.
 - 1.2.2. They are formulae to compress frequent skills: properties and associations.
2. **Non-functional requirement:** A constraint that may affect design decisions, but does not invite a discrete action. E.g. “Frame refreshment rate shall be 20 per second” does not suggest a discrete skill, but constrains others, such as the reliance upon “to display frame”.
 - 2.1. Non-functional requirements are made by skills (and entities). In addition, they are summarized and categorized in an appendix.
3. **Product vision:** A skill that defines the objective of the entire software product, so that all other skills may be verified against it. E.g. “to demonstrate telephone touch capability through an interesting game”.
4. **Required Entity:** A unique noun on which information is managed and with which work is done.
 - 4.1. Entities feature a vision, a list of *required skills* (detailing the vision), (optionally) a list of *required properties* and (optionally) a list of *required associations*.
 - 4.2. Entities may be grouped under software components.
5. **Entity vision:** A skill that defines the objective of an entity, effectively making all its other skills of the entity into its details. E.g. MVC Editor: “to edit document visually in multiple Views”, Model: “to store data for editing”, View: “to display Model data for editing from a particular View”, Edit Control: “to alert a View to edit requests”, Controller: “to detect and dispatch edit requests”.
 - 5.1. A vision skill is not required to have a discrete code mapping.
6. **Required skill:** A uniquely-named capability of the software to perform a discrete job with the platform *to satisfy the user* (directly or indirectly).
 - 6.1. Skills follow the required skill form.
 - 6.2. Where *polymorphism* is required, skills that implement a generic skill are non-unique and must be qualified to their entity. E.g. generic: “to compute Game Entity position”, implementation: “to (implement) compute Game Entity position (by Monster)”.
7. **User story.** Statement by a potential user of the product demonstrating skill necessity.
 - 7.1. User stories follow the “user story” form.
8. **Property.** A short cut for required access skills, e.g. “to get” “to set” “to iterate”.
 - 8.1. Functional *properties* differ from data-driven *attributes* (as in the entity/relationship model). Attributes specify member data (which belong in TLD); properties are required-skill shortcuts.
9. **Association.** A property that is defined by an entity (in this problem domain). E.g. The current Level (in World), the current World (in The Game), the list of Worlds (in The Game), the Chapters in a Book.
 - 9.1. the term “association” is borrowed from the Entity/Relationship Model of information. The SRS association *subset* is binary and directed.
 - 9.2. Association follows the association form.
10. **Role name.** The role played by instances of the target entity in the context of an association. The role name creates a subset of the entity, e.g., a Person sits in Jail in the role of “inmate”.
 - 10.1. The role name is optional. By default, any instance may fit.
 - 10.2. A many-role name refers to the collection (e.g. “levels” in World) or container (e.g. “request stack” in Server).
11. **Provisional software component.** (Also: module, package, assembly). The grouping of entities into a unit for the purposes of deployment, distribution and maintenance.
 - 11.1. The division to components is optional in SRS and does not constrain the TLD anyway. It usually represents product architecture, which is deep in TLD territory. E.g. applying the Model/View/Controller architecture (“MVC”) to a game is likely to result in division to at least (1) “Game Engine”, (2) “GUI”, (3) “Event dispatch” and (4) “Graphic Primitives”. This division, although reasonable, owes nothing to the *problem domain*!
12. **Component vision:** A required skill that uniquely defines the objective of a component, so that its entity visions become its details.

12.1. The component vision is not required to have a discrete code mapping.

12.2. **“Façade”** is an entity, to typically singleton, that donates its vision to its component, e.g. “The Game” singleton is façade for the “Game Engine” component.

12.2.1. Typically, a façade entity concentrates access to all skills in its component (by either forwarding or granting access).

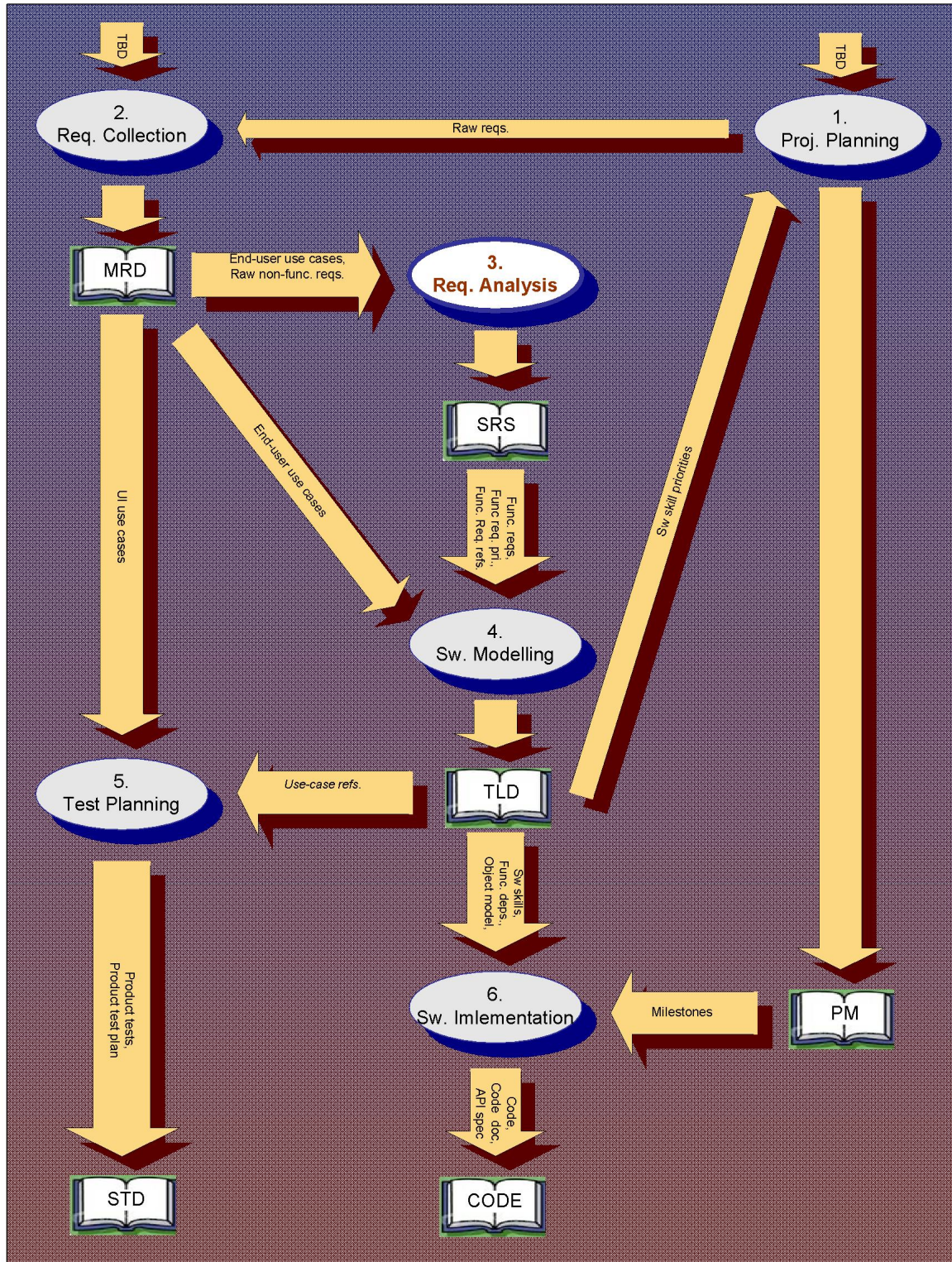
13. **Constraint.** A rule that constrains a skill (or an entity). A constraint consists of text and (optionally) validation and testing rules.

14. **Constraint category.** Non-functional requirements are concentrated in an appendix, headed by convenient categories, e.g. performance constraints, platform and resource constraints, localization constraints, standard compliance, security constraints.

Appendices

Appendix 1. Software Development Life Cycle (Dataflow)

The chart shows the document *shelf*, ordered by output/input dependencies.



- The graph describes the finished document shelf. Ovals stand for required documents and arrows for input/output dependencies – i.e. “dataflow”.
 - Dataflow may represent two facts:
 - Direct reference to one document from another (“see item...”).
 - The fact that the information in one document has been processed from the information in the other document.
 - The processing may vary from mere copying to reformatting to inspiration. The purpose is to point out that documents are not created in a void, but owe to each other mainly by *recycling* the same information with varying focus and refinement.
 - Ovals do not necessarily imply a continuous, contiguous process.
 - If this was so, than the dataflow would dictate a “waterfall model”, where the preparation of each document must follow the successful completion of all documents on which it depends.
 - The waterfall model is a strong source of inspiration, but is only feasible under the *ideal* conditions of complete control over the process and no risk factors.
 - Normally, compiling the document shelf is an iterative process, where the preparation of at least the pivotal documents is likely to be segmented and concurrent with other documents. The quanta of work owe to gradual refinement and concentration on discrete subjects.
 - In an iterative software life cycle, the main “waterfall” dataflow graph is complemented with unhierarchical (backward and sideways pointing) arcs, representing *feedback*. The typical example is Top-level Design leading to better understanding of the problem domain, which must be reflected in - i.e. opens - the SRS document.
 - The prospect of charting the dataflow of legitimate feedback use cases – if feasible at all - is beyond the scope of this document.
 - Of course, the iterative document shelf, where no single document is ever final, requires a rigorous change control system (at least, to maintain cross-document references).
 - Change control is beyond the scope of this document.
- The proposed dataflow deliberately *defers* some outputs from SRS to TLD, outputs which one could have already extract - albeit in crude form and risking the backlash of change - as early as the SRS. These arguable products are (1) use case references and (2) requirement priorities.

Appendix 2. Form

1. **The required skill Form.** “The software product shall be capable to [verb] [rest of sentence].
 - 1.1. A required skill may always be expressed in the *complete* form, as in “the system shall be capable to identify collision between Game Entities” or the short form, as in “to identify collision between Game Entities”.
 - 1.2. A skill is discrete, suggesting a single decision. E.g. “to *identify* collision between Game Entity and bomb and *blast* it” is an ill-formed skill (because it features two verbs).
 - 1.3. **Internal references.** References inside a skill, e.g. its entity, should be highlighted, e.g. by capitalization.
2. **Skill usage contract.** A *structured* skill descriptor, useful in TLD.
 - 2.1. The usage contract is defined in a separate document.
3. **User Story Form.** “As [role player], I [predicate expressing need] [value gained]”.
 - 3.1. The role player may be either an *organizational title* or a person name (which may be real or invented).
 - 3.2. The *value* clause is optional.
4. **The Required Property Form.**
 - 4.1. **Quantity.** Required properties may have quantity of either one (the default) or many. Many - properties may be sequenced, qualified by index or qualified by key.
 - 4.1.1. **One-properties** *imply* the required skills “to get [property] from [entity]” and “to set [property] in [entity]”.
 - 4.1.2. **Many-properties** imply the required skill “to iterate in [property] in [entity]” and “to add to [property] in [entity]”.
 - 4.1.2.1. **Qualified** many-properties also imply the required skills “to set into [property] in [entity]” and “to get from [property] in [entity]”.
 - 4.2. Exceptions to the default are specified in comment form.
5. **The association form.**
 - 5.1. **Association quantity:** The number of instances that are accessible from each side. E.g. the “1:3” association from World to Level states that all World -related skills have exactly three Levels at their disposal, accessible from within the world (as opposed to obtained as arguments or available globally).
 - 5.1.1. Association quantity is expressed by a pair of ranges delimited by colon.
 - 5.1.1.1. Each range consists of either a pair of numbers delimited by hyphen (or double dot) or a single number.
 - 5.1.1.1.1. The numbers represent lower bound and upper bound, respectively.
 - 5.1.1.1.2. A single number represents lower bound equals upper bound.
 - 5.1.1.1.3. The number may be replaced, where appropriate, by the letter “N”, meaning “any number (but not zero)”.
 - 5.1.1.2. The default association quantity is “1:1”.
 - 5.1.2. **Association strength:** *Two possible values:*
 - 5.1.2.1. **“Contained”** (also, “contained by value”, “composes” - UML): The target role is controlled for its life time (“possessed”, is “part of”) by the source role. The target is not necessarily created by the source, but will disappear from the scene with it.
 - 5.1.2.2. **“Referenced”** (also, “contained by reference”, “aggregates” - UML): The target role is known to the source role, but is not controlled by it. Other objects may share this knowledge.
 - 5.1.3. The default association strength is “*contained by value*”.

Appendix 3. Methodical guidelines

1. General.

1.1. Redundancy (prevention of).

1.1.1. By all means, *do repeat* obvious terms that are *cornerstones* of the problem domain (e.g. the entities and very functional requirements that define the product).

1.1.2. *Do omit* any term that may be understood from other information, if it has no dependencies and cannot possibly be missed by the TLD authors (due to convention, domain knowledge or common sense). E.g. specify properties instead of obvious “to get” and “to set” skills.

1.2. Complexity. Stress facts about the problem domain that either *increase or decrease* design complexity. Omit information that is otherwise obvious if it has no effect on design complexity. (See complexity pointers below).

1.2.1. Bear in mind that one of the SRS objectives is to supply data for assessing the cost of designing and implementing the software. Software complexity is affected mainly (apart from project-management factors) by the *size* of the problem domain (the number of required entities, required skills and required associations) and its complexity (the nature of the associations and the depth of skill reliance).

2. Deliverables. SRS deliverables should not involve dates.

2.1. The function of distributing product functionality among *milestones* and scheduling them belongs in the PM document.

3. Required entities.

3.1. Instances.

3.1.1. Entities typically have instances. However, naming instances is not a worthwhile SRS function, unless used to suggest *associations*.

3.1.1.1. Much work is put, in the implementation phase, to building classes (entities).

Variables (instances) are created as much and were needed. The declaration of an instance involves no cost in work hours.

3.1.1.1.1. All cost that seems to be involved with declaring and debugging variables actually belongs in entities, associations, properties and skills.

3.1.2. An exception is made by *singleton* entities, which have a single instance and are globally available. However, (true) singleton entities are rare in the problem domain. They are more typical of the solution domain, where they implement resource management – Pool, Allocator, etc.).

3.2. **Naming.** Use Human names, with words delimited by spaces. Reserve programmatic notations (e.g. “Camel Notation” and underscores) to programming (and CASE tools)!

3.2.1. Required entity names must be *capitalized* (throughout).

3.2.2. The capitalization of required skill names must be retained when *referenced*, e.g. inside required skills.

3.3. The required entities in the SRS are tentative. They do not have to suggest programmatic entities (e.g. classes) on a one-to-one basis. Leave this understanding to TLD authors.

4. Required skills.

4.1. **Required of whom?** The skills are required *of the software*. Please do not mix user skills with software skills. E.g. “to push a button” is a user skill. “To respond to button push” (and possibly also “to recognize button pushing”) are required *software* skills.

4.1.1. *The User* rarely suggests a programmatic *entity*. The software automates *controls* that the user may manipulate, but not the consciousness that manipulates them (i.e. The User).

4.2. **Name uniqueness.** Required skills are uniquely named. It is not enough for a skill to be unique in the context of its entity. E.g. “to tell the state of the game” is correct. “To tell state” (required of “The Game”) – is incorrect.

4.2.1. The reason is that required skills are unique product-wide is that they must be available for review by the product owner (and whoever else it may concern). The client makes requirements *of the entire body of software/hardware*. The idea of collecting required skills under entities is a design technique (as well as programming language construct) which may or may not appeal to the client.

5. The required skill Form.

- 5.1.1. Required skills with multiple verbs (“to do this and to do that”) are possible, but suggest arguable requirements! E.g. “to identify collision with an enemy and blast it” may do well to capture the original requirement, but makes a poor case for programmatic reuse.
 - 5.1.2. Avoid dropping the “to”, using the “imperative statement” form. This inevitably leads to premature digression in programmatic detail (such as function names, arguments and return values). On the contrary, when the required skill name starts with “to”, it remains a requirement.
 - 5.1.3. Avoid using nouns instead of verbs (e.g. “collision detection” instead of “to detect collision”). The second is a requirement, while the first can be just about anything!).
 - 5.1.3.1. If what seems like a functional requirement does not yield to the required -skill form (e.g. “the score” – we do need a score in The Game!), then it may be an entity (“The Score”), a property (the score in The Game), an association (between The Game and a Counter object, if we have one) or possibly a non-functional requirement (or should not be in the SRS in the first place!).
 - 5.1.3.2. Nouns do suit *use-case names* better than verbs. E.g. “to detect collision” may be the major requirement extracted from the “Collision Detection” use case.
 - 5.1.4. Avoid generic and horizontally-aggregate skills, such as “to process...”, “to manage...”. (Unless they involve real action and are indeed *called that way* in the problem domain). E.g. “to manage the score” is not a required *skill* (because it does not invite a *discrete* piece of action), but rather a layer of plaster trying to cover a property. On the contrary, “to increment the score” and “to decrement the score” are required skills (though, so obvious as to be compressed in “the score” property).
 - 5.1.4.1. Although the introduction of generic skills seems *to reduce* the body of requirements, in effect, it is a source of obscurity, increasing the complexity of the SRS.
- 5.2. **Usage contract.**
- 5.2.1. Defer specifying exhaustive usage contract to TLD phase. At SRS phase, concentrate on phrasing meaningful required skill names. Specify usage contract (in particular and thorough skill description in general) only if not understood unambiguously from the name or the use-case context.
 - 5.2.2. **Transform** rules may or may not be implemented as **formal** arguments and return values. This information is seldom of importance in the SRS.
 - 5.2.3. **Transition** rules are of any value only if they create coupling, i.e. the **effect** of one required skill is the **start** of another.
 - 5.2.4. The objective of **invalidation** rules is to prevent the redundant specification of negative “required skills”. The capability “to cope with bad data” is not a discrete skill, but rather an exit condition from another, proper required skill.
- 5.3. **Matching skills to entity**
- 5.3.1. The *grouping of skills* under an entity is *tentative*. The TLD authors may implement a required skill as a *method* (or code block or statement) in any class, including one that implements another entity. E.g. the SRS required skill “to compute next Game Entity position” (by Game Entity, in the SRS), turns out to be implemented in the TLD by a skill of the same name, but in class “Physics Engine”. The prospect of going back to update the SRS is arguable in this case, depending on whether one sees “Physics Engine” as part of the problem or solution domains.
 - 5.3.1.1. *Warning!* The premature identification of required skills as programmatic *methods* is bound to lead to the premature specification of programmatic entities (e.g. Factory, Pool, Game Grid), design pattern artifacts (e.g. Visitor, Observer, Composite) and design idioms (e.g. Reference Count base/template class).
- 5.4. **Locating required skills.**
- 5.4.1. *Extracting the final requirements* from the base document is an art. It cannot be done mechanically due to the inherent difference in perspectives between client and technology expert. The one thing in common between the base document and SRS will be so many functional requirements. However, their *ordering, emphasis and phrasing* may have little in common. In addition, The SRS will be populated with many functional requirements which *are not* in the base document and vice versa.
 - 5.4.1.1. On the one hand, client-prepared requirement documents typically...

- 5.4.1.1.1. ...mix *levels* of abstraction.
- 5.4.1.1.2. ...mix *major* requirements with *minor* requirements that are implicit in them (according to programmatic logic).
- 5.4.1.1.3. ...exaggerate the importance of minor technical *details*.
- 5.4.1.1.4. ...tend to *suggest familiar* current-technology solutions, instead of just stating the problem. E.g. the existence of an (invisible) Game Grid is often a design artifact which the user may require in the next game, not necessarily justifiably. The SRS authors are free to ignore such requirements (as long as the result will be satisfactory).
- 5.4.1.2. On the other hand, requirement analysis is bound to come up with many required skills that *are not stated* in the base requirements, but are implicit in them (according to programmatic logic).
 - 5.4.1.2.1. *While some skills are required of the software to satisfy the user, as many skills are required of the software to satisfy itself* (to satisfy *the latter*, recursively).
 - 5.4.1.2.2. Eventually, *every software skill must be traced to a client requirement*, but not necessarily on an *immediate* basis!

6. Properties.

- 6.1. Properties are required-skill *generators*. The objective of requiring properties is to reduce the number of *obvious* required skills that must be written down.
 - 6.1.1. Properties do not reduce design complexity, only *size*.
- 6.2. Properties should not involve significant processing.
 - 6.2.1. Property-access that involves logic that must be detailed make explicit required skills. E.g. although “position” may be a property of “Game Entity”, “to compute Game Entity position” is unlikely to be a trivial “getter” that may be overlooked.
- 6.3. The question whether the property does or does not suggest an attribute (i.e. programmatic “field”) is not interesting at SRS time.
- 6.4. Specify programmatic data types only in case of ambiguity or unusual value range. E.g. there is no need to state that “the score” is an “integer” or that “user name” is a string (at least, not in SRS stage).
- 6.5. Avoid inventing data types (at SRS stage). Data types - *as opposed to entities* – are dumb value holders that only know “to get”, “to set” and “to convert into ...”.
 - 6.5.1. Data types – unlike properties – imply skills *generically*, i.e. for values that are below the requirements of the *problem* domain.
- 6.6. Do not use properties to specify attributes that implement a defined association, e.g. number of elements, the current element pointer etc. Let this be understood at TLD from the association descriptor, e.g. “cyclic list”.

7. Associations.

7.1. Complexity.

- 7.1.1. Associations are a major contribution to software complexity (and program bugs)! Henceforth, the importance of telling them from properties as early as the SRS.
- 7.1.2. **Quantity.**
 - 7.1.2.1. **Many-to-one** associations *increase* design complexity. They imply *sharing* and therefore require some sharing safety mechanism, e.g., reference -counting.
 - 7.1.2.2. **Many-to-many** associations are *major contributors* to design complexity.
- 7.1.3. **Strength.**
 - 7.1.3.1.1. The requirement for containment by value *reduces* design complexity.
 - 7.1.3.1.1.1. **Containment by value** suggests *centralized management* of functionality and thus, *introduces order* into the problem domain. In a containment hierarchy, required skills tend *to follow the natural dependency* from containing object to contained object (and seldom the other way around).
 - 7.1.3.1.2. The requirement for **containment by reference** *increases* design complexity. It implies the existence of a *shared* resource, calling for the detailed design of a *sharing policy* (whereas containment by value is implemented trivially).

7.2. Association networks.

- 7.2.1. Entity/Relationship diagrams (or *class diagrams*) are not required in the in the SRS.
However, fragments may be attached to illustrate a discrete point (e.g. the internal game hierarchy: “The Game has many Worlds (of which one is current) where each has three levels, of which one is current”).
 - 7.2.1.1. Avoid the temptation to show the “overall” (i.e. data-driven) picture. Data-driven design is inseparably coupled with *implementation detail* and design-pattern selection.
 - 7.2.1.2. Avoid specifying inheritance in the SRS.
- 7.3. **Source role quantity.** In object-oriented programming, all associations are binary and directed; i.e. only the target role access is to be implemented. Therefore, the source role quantity is for information (about the problem domain) only.
 - 7.3.1. This information is important, because it may not be reverse-engineered trivially from the resulting code.