CHAPTER 01

Software Requirements

ACRONYMS

ATDD	Acceptance Test Driven Development
BDD	Behavior Driven Development
CIA	Confidentiality, Integrity, and Availability
FSM	Functional Size Measurement
INCOSE	International Council on Systems Engineering
JAD	Joint Application Development
JRP	Joint Requirements Planning
SME	Subject Matter Expert
SysML	Systems Modeling Language
TDD	Test Driven Development
UML	Unified Modeling Language

INTRODUCTION

Software requirements should be viewed from two perspectives. The first is as an expression of the needs and constraints on a software product or project that contribute to the solution of a real-world problem. The second is that of the activities necessary to develop and maintain the requirements for a software product and for the project that constructs it. Both perspectives are presented in this knowledge area (KA).

If a team does a poor job of determining the requirements, the project, the product or both are likely to suffer from added costs, delays, cancellations and defects. One reason is that each software product requirement generally leads to many design decisions. Each design decision generally leads to many code-level decisions. Each decision can involve several test decisions, as well. In other words, determining the requirements correctly is high-stakes work. If not detected and repaired early, missing, misinterpreted and incorrect requirements can induce exponentially cascading rework to correct them.

Real-world software projects tend to suffer from two primary requirements-related problems:

1. incompleteness: stakeholder requirements, and necessary detail, exist that are not revealed and communicated to the software engineers;
2. ambiguity: requirements are communicated in a way that is open to multiple interpretations, with only one possible interpretation being correct.

Beyond the obvious short-term role requirements play in initial software construction, they also play a less recognized but still important role in long-term maintenance. Upon receiving software without any supporting documentation, a software engineer has several means to determine what that code does, such as execute it, step through it with a debugger, hand-execute it, statically analyze it, and so on. The challenge is determining what that code is *intended to do*. What is generally referred to as a *bug* — but is better called a *defect* — is simply an observable difference between what the software is intended to do and what it does. The role of requirements documentation throughout the service life of the software is to capture and communicate intent for software engineers who maintain the code but might not have been its original authors.

The Software Requirements KA concerns developing software requirements and managing those requirements over the software's service life. This KA provides an understanding that software requirements:

- are not necessarily a discrete front-end activity of the software development life cycle but rather a process initiated at a project's beginning that often continues to be refined throughout the software's entire service life;
- need to be tailored to the organization and project context.

The term *requirements engineering* is often used to denote the systematic handling of requirements. For consistency, the term *engineering* will not be used in this KA other than for software engineering per se.

The Software Requirements KA is most closely related to the Software Architecture, Software Design, Software Construction, Software Testing, and Software Maintenance KAs, as well as to the models topic in the Software Engineering Models and Methods KA, in that there can be high value in specifying requirements in model form.

This KA is also related to the Software Life Cycles topic in the Software Engineering Process KA, in that this KA's focus is on *what* and *how* requirements work can and should be done, whereas the project's life cycle determines *when* that work is done. For example, in a waterfall life cycle, all requirements work is essentially done in a discrete *Requirements phase* and is expected to be substantially complete before any architecture, design and construction work occurs in subsequent phases. Under some iterative life cycles, initial, high-level requirements work is done during an *Inception phase*, and further detailing is done during one or more *Elaboration phases*. In an Agile life cycle, requirements work is done incrementally, just in time, as each additional element of functionality is constructed.

The *whats* and *hows* of software requirements work on a project should be determined by the nature of the software constructed, not by the life cycle under which it is constructed. Insofar as requirements documentation captures and communicates the software's intent, downstream maintainers should not be able to discern the life cycle used in earlier development from the form of those requirements alone.

This KA is also related, but somewhat less so, to the Software Configuration Management, Software Engineering Management and Software Quality KAs. Software CM approaches can be applied to trace and manage requirements; software quality looks at how well formed the requirements are, and engineering management can use the status of requirements to evaluate the completion of the project.

BREAKDOWN OF TOPICS FOR SOFTWARE REQUIREMENTS

The topic breakdown for the Software Requirements KA is shown in Figure 1.1.

1. Software Requirements Fundamentals

1.1. Definition of a Software Requirement

[1*, c1pp5-6] [2*, c4p102]

Formally, a *software requirement* has been defined as [28]:

- a condition or capability needed by a user to solve a problem or achieve an objective;
- a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed document;
- a documented representation or capability as in (1) or (2) above.

This formal definition is extended in this KA to include expressions of a software project's needs and constraints.

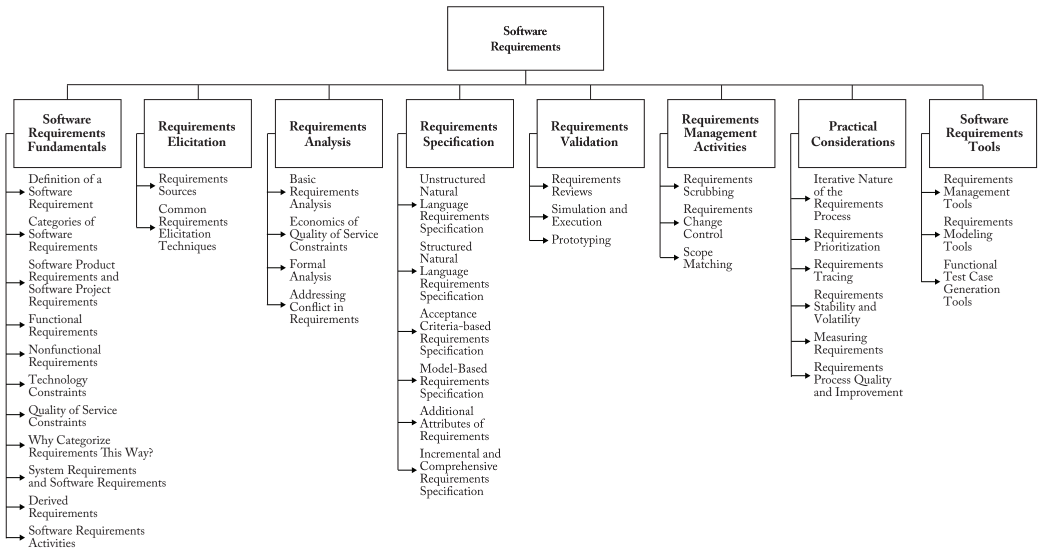
Software Requirements
Software Requirements Fundamentals
Definition of a Software Requirement
Categories of Software Requirements
Software Product Requirements and Software Project Requirements
Functional Requirements
Nonfunctional Requirements
Technology Constraints
Quality of Service Constraints
Why Categorize Requirements This Way?
System Requirements and Software Requirements
Derived Requirements
Software Requirements Activities
Requirements Elicitation
Requirements Sources
Common Requirements Elicitation Techniques
Requirements Analysis
Basic Requirements Analysis
Economics of Quality of Service Constraints
Formal Analysis
Addressing Conflict in Requirements
Requirements Specification
Unstructured Natural Language Requirements Specification
Structured Natural Language Requirements Specification
Acceptance Criteria-based Requirements Specification
Model-Based Requirements Specification
Additional Attributes of Requirements
Incremental and Comprehensive Requirements Specification
Requirements Validation
Requirements Reviews
Simulation and Execution
Prototyping
Requirements Management Activities
Requirements Scrubbing
Requirements Change Control
Scope Matching
Practical Considerations
Iterative Nature of the Requirements Process
Requirements Prioritization
Requirements Tracing
Requirements Stability and Volatility
Measuring Requirements
Requirements Process Quality and Improvement
Software Requirements Tools
Requirements Management Tools
Requirements Modeling Tools
Functional Test Case Generation Tools

Figure 1.1. Breakdown of Topics for the Software Requirements KA

At its most basic, a software requirement is a property that must be exhibited to solve a real-world problem. It might aim to automate all or part of a task supporting an organization's business policies and processes, correct existing software's shortcomings, or control a device — just a few of the many problems for which software solutions are possible.

Business policies and processes, as well as device functions, are often very complex. By extension, software requirements are often a complex combination of requirements from various stakeholders at different organizational levels who are involved or connected with some aspect of the environment in which the software will operate.

Clients, customers and users usually impose requirements. However, other third parties, like regulatory authorities and, in some cases, the software organization or the project itself, might also impose requirements. (See also [5, c1] [6, c1] [9, c4].)

1.2. Categories of Software Requirements

[1*, c1pp7-12] [2*, s4.1]

Figure 1.2 shows the categories of software requirements defined in this KA and the relationships among those categories. (See also [5, c1] [6, c1] [9, c4].) Each category is further described below.

1.3. Software Product Requirements and Software Project Requirements

[1*, c1pp14-15]

Software product requirements specify the software's expected form, fit or function. *Software project requirements* — also called *process requirements* or, sometimes *business requirements*— constrain the project that constructs the software. Project requirements often constrain cost, schedule and/or staffing but can also constrain other aspects of a software project, such as testing environments, data migration, user training, and maintenance. Software project requirements can be captured in a project charter or other high-level project initiation document. They are most relevant to how the project is managed (see the Software Engineering Management KA) or what life cycle process should be used (see the Software Engineering Process KA). This KA does not discuss software project requirements further.

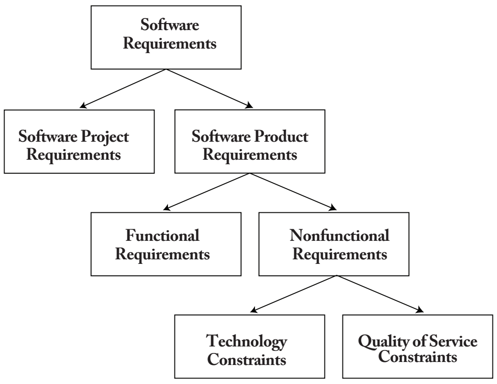
Software Requirements
Software Project Requirements
Software Product Requirements
Functional Requirements
Nonfunctional Requirements
Technology Constraints
Quality of Service Constraints

Figure 1.2. Categories of Software Requirements

1.4. *Functional Requirements*

[1*, c1p9] [2*, s4.1.1]

Functional requirements specify observable behaviors that the software is to provide — policies to be enforced and processes to be carried out. Example policies in banking software might be "an account shall always have at least one customer as its owner," and "the balance of an account shall never be negative." Example processes could specify the meanings of depositing money into an account, withdrawing money from an account and transferring money from one account to another.

Even highly technical (nonbusiness-oriented) software, such as software that implements the transmission control protocol/internet protocol (TCP/IP) network communications protocol, has policies and processes: "a Port shall be able to exist with zero, one, or many associated Connections, but a Connection shall exist on exactly one associated Port," "acceptable states of a Connection shall be 'listen,' 'syn sent,' 'established,' 'closing,' . . . ," and "if the time-to-live of a Segment reaches zero, that Segment shall be deleted." (See [5, c1] [6, c10] [9, c4].)

1.5. *Nonfunctional Requirements*

[1*, c1pp10-11] [2*, s4.1.2]

Nonfunctional requirements in some way constrain the technologies to be used in the implementation: What computing platform(s)? What database engine(s)? How accurate do results need to be? How quickly must results be presented? How many records of a certain type need to be stored? Some nonfunctional requirements might relate to the operation of the software. (See the Operation and Maintenance KA.) (See also [5, c1] [6, c11] [9, c4].)

The nonfunctional requirements can be further divided into technology constraints and quality of service constraints. They have essential relationships among themselves, which affect them positively or negatively and require that, whenever a nonfunctional requirement is modified, the impact it may cause on others should be considered.

1.6. *Technology Constraints*

These requirements mandate — or prohibit — use of specific, named automation technologies or defined infrastructures. Examples are requirements to use specific computing platforms (e.g., Windows™, MacOS™, Android OS™, iOS™), programming languages (e.g., Java, C++, C#, Python), compatibility with specific web browsers (e.g., Chrome™, Safari™, Edge™), given database engines (e.g., Oracle™, SQL Server™, MySQL™), and general technologies (e.g., reduced instruction set computer (RISC), Relational Database). A requirement prohibiting use of pointers would be another example. (See also [9, c4].)

1.7. *Quality of Service Constraints*

These requirements do not constrain the use of specific, named technologies. Instead, these specify acceptable performance levels an automated solution must exhibit. Examples are response time, throughput, accuracy, reliability and scalability. ISO/IEC 25010: "System and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models" [27] contains a large list of the kinds of quality characteristics that can be relevant for software. (See also [9, c4].) Safety and security are also a particularly important topic where requirements tend to be overlooked. (See the Security KA for details on the kinds of specific security requirements that should be considered.) (See also [2*, c13].)

1.8. Why Categorize Requirements This Way?

Categorizing requirements this way is useful for the following reasons:

- requirements in one category tend to come from different sources than other categories;
- elicitation techniques often vary by source;
- analysis techniques vary by category;
- specification techniques vary by category;
- validation authorities vary by category;
- the different categories affect the resulting software in different ways.

In addition, organizing the requirements in these categories is beneficial in the following ways:

- complexity can be better managed because different areas can be addressed separately; software engineers can deal with policy and process complexities without worrying about automation technology issues at the same time (and vice versa). One large problem becomes two smaller ones. This is classic *divide and conquer* complexity management;
- distinct areas of expertise can be isolated; stakeholders, not software engineers, are the experts in the policies and processes to be automated. Software engineers, not stakeholders, are the technology experts. When a business expert is given interspersed functional and nonfunctional requirements for review or validation, they might give up because they don't understand — or even care about — the technology issues. The relevant requirements reviewer can focus on just the subset of requirements relevant to them.

The *Perfect Technology Filter* originally described in [18, c1-4] but also explained in [8] and [9, c4] helps separate functional from nonfunctional requirements. Simply put, functional requirements are those that would still need to be stated even if a computer with infinite speed, unlimited memory, zero cost, no failures, etc., existed on which to construct the software. All other software product requirements are constraints on automation technologies and are therefore nonfunctional.

Large systems often span more than one subject matter area, or domain. As explained in [9, c6], recursive design shows how nonfunctional requirements in a parent domain can become, or can induce, functional requirements in a child domain. For example, a nonfunctional requirement about user security in a parent banking domain can become or can induce functional requirements in a child security domain. Similarly, cross-cutting nonfunctional requirements about auditing and transaction management in a parent banking domain can become or induce functional requirements in a child auditing domain and a child transaction domain. Decomposing large systems into a set of related domains significantly reduces complexity.

1.9. System Requirements and Software Requirements

The International Council on Systems Engineering (INCOSE) defines a *system* as "an interacting combination of elements to accomplish a defined objective. These include hardware, software, firmware, people, information, techniques, facilities, services, and other support elements" [24].

In some cases, it is either useful or mandatory to distinguish system requirements from software requirements. System requirements apply to larger systems — for example, an autonomous vehicle. Software requirements apply only to an element of software in that larger system. Some software requirements may be derived from system requirements. (See also [5, c1].) In other cases, the software is itself the system of interest, and hardware and

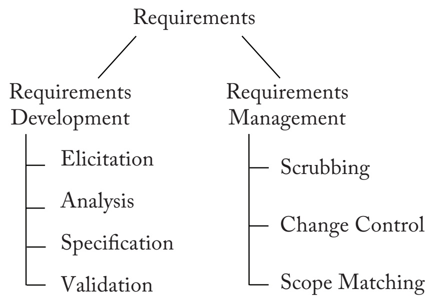

Figure 1.3. Software Requirements Activities

support system are regarded as the platform or infrastructure, so that the system requirements are mostly software requirements.

1.10. Derived Requirements

In practice, *requirements* can be context-sensitive and can depend on perspective. An external stakeholder can impose a scope requirement, and this would be a requirement for the entire project — even if that project involves hundreds of software engineers. An architect's decision to use a pipes-and-filters architecture style would not be a requirement from the perspective of the overall project stakeholders, but a design decision. But that same decision, when seen from the perspective of a sub-team responsible for constructing a particular filter, would be considered a requirement.

The aerospace industry has long used the term *derived requirement* to mean a requirement that was not made by a stakeholder external to the overall project but that was imposed inside the larger development team. The architect's pipes-and-filters decision fits this definition. That choice would be seen as a design decision from the point of view of external stakeholders, but as a requirement for the sub-teams responsible for developing each filter. (See also [9, c4].)

1.11. Software Requirements Activities

[1*, c1pp15-18] [2*, s4.2]

Figure 1.3 shows the requirements development and management activities.

Requirements development, as a whole, can be thought of as "reaching an agreement on what software is to be constructed." In contrast, requirements management can be considered "maintaining that agreement over time." Each activity is presented in this KA. Requirements development activities are presented as separate topics, with requirements management presented as a single topic. (See also [5, c1] [6, 2].)

2. Requirements Elicitation

[1*, c6-7] [2*, s4.3]

The goal of requirements elicitation is to surface candidate requirements. It is also called *requirements capture*, *requirements discovery* or *requirements acquisition*. As stated earlier, one problem in requirements work on real-world software projects is incompleteness. This can be the result of inadequate elicitation. Although there is no guarantee that a set of requirements is complete, well-executed elicitation helps minimize incompleteness. (See also [5, c2-3] [6, c3-7].)

2.1. Requirements Sources

[1*, c6] [2*, s4.3]

Requirements come — can be elicited — from many different sources. All potential requirements sources should be identified and evaluated. A *stakeholder* can be defined as any person, group or organization that:

- is actively involved in the project;
- is affected by the project's outcome;
- can influence the project's outcome.

Typical stakeholders for software projects include but are not limited to the following:

- clients — those who pay for the software to be constructed (e.g., organizational management);
- customers — those who decide whether a software product will be put into service;
- users — those who interact directly or indirectly with the software; users can

often be further broken down into distinct user classes that vary in frequency of use, tasks performed, skill and knowledge level, privilege level, and so on;

- subject matter experts (SMEs);
- operations staff;
- first-level product support staff;
- relevant professional bodies;
- regulatory agencies;
- special interest groups;
- people who can be negatively affected if the project is successful;
- developers.

Stakeholder classes are groups of stakeholders that have similar perspectives and needs. Working on a software project in terms of stakeholder classes rather than with individual stakeholders can produce important, additional insight.

Many projects benefit from performing a stakeholder analysis to identify as many important stakeholder classes as possible. This reduces the possibility that the requirements are biased toward better-represented stakeholders and away from less well-represented stakeholders. The stakeholder analysis can also inform negotiation and conflict resolution when requirements from one stakeholder class conflict with requirements from another. (See also [5, c3] [6, c3].)

Requirements are not limited to only coming from people. Other, non-person requirements sources can include:

- documentation such as requirements for previous versions, mission statements, concept of operations;
- other systems;
- larger business context including organizational policies and processes;
- computing environment.

2.2. Common Requirements Elicitation Techniques [1*, c7] [2*, s4.3]

A wide variety of techniques can be used to elicit requirements from stakeholders. Some techniques work better with certain stakeholder classes than others. Common stakeholder elicitation techniques include the following:

- interviews;
- meetings, possibly including brainstorming;
- joint application development (JAD) [13], joint requirements planning (JRP) [14] and other facilitated workshops;
- protocol analysis;
- focus groups;
- questionnaires and market surveys;
- exploratory prototyping, including low-fidelity and high-fidelity user interface prototyping [1*, c15];
- user story mapping.

Elicitation can be difficult, and the software engineer needs to know that (for example) users might have difficulty describing their tasks, leave important information unstated or be unwilling or unable to cooperate. Elicitation is not a passive activity. Even if cooperative and articulate stakeholders are available, the software engineer must work hard to elicit the right information. Many product requirements are tacit or can be found only in information that has yet to be collected.

Requirements can also be elicited from sources other than stakeholders. Such sources and techniques include the following:

- previous versions of the system;
- defect tracking database for previous versions of the system;
- systems that interface with the system under development;
- competitive benchmarking;
- literature search;
- quality function deployment (QFD)'s House of Quality [15];
- observation, where the software engineer studies the work and the environment where the work is being done;
- apprenticing, where the software engineer learns by doing the work;
- usage scenario descriptions;

- decomposition (e.g., capabilities into epics into features into stories);
- task analysis [16];
- design thinking (empathize, define, ideate, prototype, test) [17];
- ISO/IEC 25010: "System and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models" [27];
- security requirements, as discussed in the Security KA;
- applicable standards and regulations.

(See also [5, c3] [6, c4-7].)

3. Requirements Analysis [1*, c8-9]

Requirements are unlikely to be elicited in their final form. Further investigation is usually needed to reveal the full, true requirements suggested by the originally elicited information. Requirements analysis helps software developers understand the meaning and implications of candidate requirements, both individually and in the context of the overall set of requirements.

3.1. Basic Requirements Analysis [1*, c8-9]

The following list of desirable properties of requirements can guide basic requirements analysis. The software engineer seeks to establish any of these properties that do not hold yet. Each requirement should:

- be unambiguous (interpretable in only one way);
- be testable (quantified), meaning that compliance or noncompliance can be clearly demonstrated;
- be binding, meaning that clients are willing to pay for it and unwilling not to have it;
- atomic, represent a single decision
- represent true, actual stakeholder needs;
- use stakeholder vocabulary;
- be acceptable to all stakeholders.

The overall collection of requirements should be:

- complete —The requirements adequately address boundary conditions, exception conditions and security needs;
- concise — No extraneous content in the requirements
- internally consistent — No requirement conflicts with any other;
- externally consistent — No requirement conflicts with any source material;
- feasible — A viable, cost-effective solution can be created within cost, schedule, staffing, and other constraints.

In some cases, an elicited statement represents a solution to be implemented rather than the true problem to be solved. This risks implementing a suboptimal solution. The *5-whys* technique (e.g., [3*, c4]) involves repeatedly asking, "Why is this the requirement?" to converge on the true problem. Repetition stops when the answer is, "If that isn't done, then the stakeholder's problem has not been solved." Often, the true problem is reached in two or three cycles, but the technique is called *5-whys* to incentivize engineers to push it as far as possible.

3.2. Economics of Quality of Service Constraints [3*]

Quality of service constraints can be particularly challenging. This is generally because engineers do not consider them from an economic perspective [9, c4]. Figure 1.4 illustrates the economic perspective of a typical quality of service constraint, such as capacity, throughput and reliability, where value increases with performance level. This curve is mirrored vertically for quality of service constraints whose value decreases as performance level increases (response time and mean time to repair would be examples).

Over the relevant range of performance levels, the stakeholders have a corresponding value if the system performs at that level. The value curve has two important points:

1. Perfection point — This is the most favorable level of performance, beyond which there is no additional benefit. Even if the system can perform better than the perfection point, the customer cannot use that capacity. For example, a social media system that supports more members than the world population would have this excess capacity.
2. Fail point — This is the least favorable level of performance, beyond which there is no further reduction in benefit. For example, the social media system might need to support at least a minimum market share to be viable as a platform.

A quantified requirement point, even if stated explicitly, is usually arbitrary. It is often based on what a client feels justified requesting, given what they are paying for the software. Even if the software engineers cannot construct a system that fully achieves the stated requirement point, the software typically still has value; it just has less value than the client expected. Further, the ability to exceed the requirement point can significantly increase value in some cases.

The cost to achieve a given performance level is usually a step function. First, for a given investment level, there is some maximum achievable performance level. Then, additional investment is needed, and that further investment enables performance up to a new, more favorable maximum. Figure 1.5 illustrates the most cost-effective performance level — the performance level with the maximum positive difference between the value at that performance level and the cost to achieve it.

(See the Software Engineering Economics KA or [3*] for more information on performing economic analyses such as this.)

The software engineer should pay particular attention to positive and negative relationships between quality of service constraints (e.g., Figure 14-1 in [1*, c14]). Some quality of service constraints are mutually supporting; improving one's performance level will automatically improve the other's performance

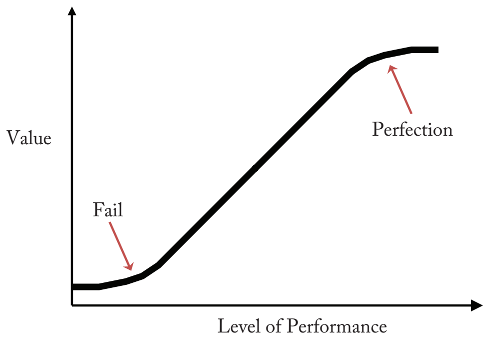

Figure 1.4. Value as a Function of Level of Performance

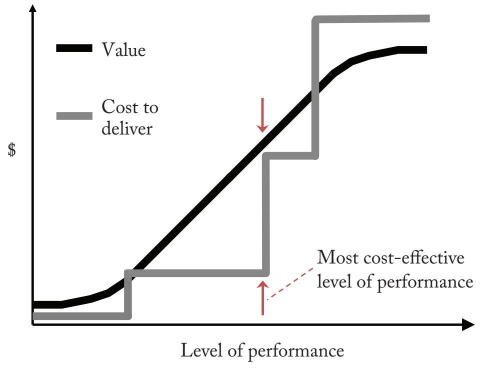

Figure 1.5. Most Cost-Effective Level of Performance

level. For example, the more modifiable code is, the more reliable it tends to be, as both modifiability and reliability are, to a degree, a consequence of how clean the code is. On the other hand, the higher the code's speed, the less modifiable it might be, because high speed is often achieved through optimizations that make the code more complex.

3.3. *Formal Analysis*

[2*, s12.3.2-12.3.3]

Formal analysis has shown benefits in some application domains, particularly high-integrity systems (e.g., [5, c6]). The formal expression of requirements depends on the use of a specification language with formally defined semantics. Formality has two benefits. First, formal requirements are precise and concise, which (in principle) will reduce the possibility for misinterpretation. Second, formal requirements can be reasoned over, permitting desired properties of the specified software to be proved. This permits static validation that the software specified by the requirements does have the properties (e.g., absence of deadlock) that the customer, users and software engineer expect it to have.

This topic is related to Formal Methods in the Software Engineering Models and Methods KA.

3.4. Addressing Conflict in Requirements

When a project has more — and more diverse — stakeholders, conflicts among the requirements are more likely. One particularly important aspect of requirements analysis is identifying and managing such conflicts (e.g., [6, c17]). Once conflicting requirements have been identified, the engineer may consider two different approaches to managing that conflict (and possibly other approaches as well) and determine the most appropriate course of action.

One approach is to negotiate a resolution among the conflicting stakeholders. In most cases, it is unwise for the software engineer to make a unilateral decision, so it becomes necessary to consult with the stakeholders to reach a consensus resolution. It is often also important, for contractual reasons, that such decisions be traceable back to the customer. A specific example is *project scope management* — namely, balancing what's desired in the stated software product requirements with what can be accomplished given the project requirements of cost, schedule, staffing and other project-level constraints. There are many useful sources for information on negotiation and conflict resolution [25].

Another approach is to apply *product family development* (e.g., [20]). This involves separating requirements into two categories. The first category contains the *invariant requirements*. These are requirements that all stakeholders agree on. The second category contains the *variant requirements*, where conflict exists. The software engineer can focus on understanding the range of variations needed to satisfy all stakeholders. The software can be designed using *design to invariants* to accommodate the invariant requirements and *design for change* to incorporate customization points to configure an instance of the system to best fit relevant stakeholders. In a simple example, some users of a weather application require temperatures displayed in degrees Celsius while others require degrees Fahrenheit.

4. Requirements Specification
[1*, c10-14, c20-26] [2*, s4.4, c5]

Requirements specification concerns recording the requirements so they can be both remembered and communicated. Requirements specification might be the most contentious topic in this KA. Debate centers on questions such as:

- should requirements be written down at all?
- if requirements are written down, what form should they take?
- if requirements are written down, should they also be maintained over time?

There are no standard answers to these questions; the answer to each can depend on factors such as the following:

- the software engineer's familiarity with the business domain;
- precedent for this kind of software;
- degree of risk (e.g., probability, severity) of incorrect requirements;
- staff turnover anticipated during the service life of the software;
- geographic distribution of the development team members;
- stakeholder involvement over the course of the project;
- whether the use of a third-party service, packaged solution or open source library is anticipated;
- whether any design or construction will be outsourced;

- the degree of requirements-based testing expected;
- effort needed to use a candidate specification technique;
- accuracy needed from the requirements-based estimates;
- extent of requirements tracing necessary, if any;
- contractual impositions of requirements specification content and format.

As stated in this KA's introduction, the *whats* and *hows* of software requirements work on a project should be determined by the nature of the software constructed, not by the life cycle under which it is constructed. Downstream maintainers should not be able to discern the life cycle used in earlier development from the form of those requirements alone. The chosen life cycle's effect should be limited to the completeness of the requirements at any point in the project. Under a waterfall life cycle, the requirements are expected to be completely specified at the end of the Requirements phase. Under an Agile life cycle, the requirements are expected to change, grow, or be eliminated continuously and not be complete until the project's end.

Some organizations have a culture of documenting requirements; some do not. Dynamic startup projects are often driven by a strong product vision and limited resources; their teams might view requirements documentation as unnecessary overhead. But as these products evolve and mature, software engineers often recognize that they need to recover the requirements that motivated product features in order to assess the impact of proposed changes. Hence, requirements documentation and change management become important to long-term success. A project's approach to requirements in general, and to requirements specification in particular, may evolve over the service life of that software.

The most basic recommendation for requirements documentation is to base decisions on an *audience analysis*. Who are the different consumers who will need information from a requirements specification? What information will they need? How can that information be packaged and presented so that each consumer can get the information they need with the least effort?

There is a degree of overlap and dependency between requirements analysis and specification. Use of certain requirements specification techniques — particularly model-based requirements specifications — permit and encourage requirements analysis that can go beyond what has already been presented.

Documented software requirements should be subject to the same configuration management practices as the other deliverables of the software life cycle processes. (See the Configuration Management KA for a detailed discussion.) In addition, when practical, the individual requirements are also subject to configuration management and traceability, which is generally supported by a requirements management tool. (See Topic 8, Software Requirements Tools.)

There are several general categories of requirements specification techniques, each of which is discussed below. The requirements specification for a given project may also use various techniques. ISO/IEC/IEEE 29148 [26], as well as [1*, c10-14], [5, c4], [6, c16], and many others offer templates for requirements documentation.

4.1. *Unstructured Natural Language Requirements Specification*

[1*, c11] [2*, s4.4.1]

Natural language requirements specifications express requirements in common, ordinary language. Natural language requirements specifications can be unstructured or structured.

A typical unstructured natural language requirements specification is a collection of statements in natural language, such as, "The system shall . . . ." For example, business rules are statements that define or constrain some aspect of the structure or the behavior of the business to be automated. "A student cannot register in next semester's courses if there remain any unpaid tuition fees" is an example of a business rule that serves as a requirement

Use case #66	Use case name: Reserve flight(s)
Triggering event(s)	Customer requests reservation(s) on flight(s)
Parameters	Passenger, itinerary, fare class, payment method(s)
Requires	Legal itinerary, fare class restrictions met
Guarantees	Seat(s) reserved for passenger on itinerary flight(s)
Normal course	Non-FF passenger, all domestic itinerary, Economy fare class, credit/debit card
Alternative course(s)	Is FF passenger: [None, Silver, Gold, Platinum, Elite] Itinerary: [all international, mixed domestic + international] Fare class: [Basic economy, Premium Economy, Business, First] Payment method: [Voucher, FF miles]
Exceptions	C/D card declined, voucher doesn't exist, voucher expired, FF account doesn't exist, insufficient miles in FF account

Figure 1.6. Example of Structured Natural Language Specification for a Single Use Case

for a university's course-registration software. Some projects can publish a user manual as a satisfactory requirements specification, although there are limits to how effective this can be. (See also [5, c4] [26].)

4.2. *Structured Natural Language Requirements Specification* [1*, c8] [2*, s4.4.2]

Structured natural language requirements specifications impose constraints on how the requirements are expressed; the goal is to increase precision and conciseness.

The simplest example might be the actor-action format. The actor is the entity responsible for carrying out the action, and action is what needs to happen. A triggering event might precede the actor, and the action might be followed by an optional condition or qualification. The statement "When an order is shipped, the system shall create an Invoice unless the Order Terms are 'Prepaid'" uses actor-action format. The triggering event is "When an order is shipped." The actor is "the system." The action is "create an Invoice." The condition/qualification is "except the Order Terms are 'Prepaid'."

Another example is a use case specification template, as shown in Figure 1.6. (See [11] for guidelines on writing good use case specifications.)

The user story format, "As a <user> I want <capability> so that <benefit>" as well as decision tables are other examples. (See also [5, c4] [6, c12, c16] [7, c2-5].)

4.3. *Acceptance Criteria-Based Requirements Specification*

This general approach includes two specific variants: acceptance test driven development (ATDD) and behavior driven development (BDD).

ATDD [2*, s3.2.3, s8.2] is a part of the larger test driven development (TDD) approach. (See the Software Testing KA.). The main idea of TDD is that test cases precede construction. Therefore, no new production code is written and no existing code is modified unless at least one test case fails, either at the unit test level or at the acceptance test level. The ATDD process has three steps:

1. A unit of functionality (e.g., a user story) is selected for implementation.
2. One or more software engineers, one or more business domain experts, and possibly one or more QA/test professionals meet — before any production design or

construction work is done — to agree on a set of test cases that must pass to show that the unit of functionality has been correctly implemented.

3. At least one of those acceptance test cases must fail on the existing software. The existence of at least one failing test case gives the software engineer(s) permission to create or modify production code to pass all of the agreed-upon test cases. This step might require several iterations. The code may also be refactored during this step.

When all acceptance test cases have passed, and presumably all unit and integration test cases as well, then the unit of functionality is deemed to have been completely and correctly implemented. The ATDD process returns to step 1, where a new unit of functionality is selected, and the cycle repeats.

ATDD might seem to be a testing technique rather than a requirements specification technique. On the other hand, a test case has the general form of "When given input that looks like X, we expect the software to produce results that look like Y." The key is the underlined phrase, "we expect the software to produce." If we simply modify that phrase to say, "the software shall produce," as in "When given input that looks like X, the software shall produce results that look like Y," what first looked like a test case now looks like a requirement. Technically, one acceptance test case can encompass more than one single requirement, but the general idea holds that the ATDD test cases are essentially precise, unambiguous statements of requirements.

The BDD approach [19] is slightly more structured, and business domain experts typically prefer it over ATDD because it is less technical in appearance. In BDD, the unit of functionality is described as a user story, in a form such as this: "As a <user> I want <capability> so that <benefit>." This leads to the identification and specification of a set of "scenarios" in this form: "Given <some context> [and <possibly more context>], when <stimulus> then <outcome> [and <possibly more outcomes>]."

If the story is "As a bank customer, I want to withdraw cash from the automated teller machine (ATM) so that I can get money without going to the bank," one scenario could be that "the account has a sufficient balance." This scenario could be detailed as "Given the account balance is $500, and the customer's bank card is valid, and the automated teller machine contains enough money in its cash box, when the Account Holder requests $100, then the ATM should dispense $100 and the account balance should be $400, and the customer's bank card should be returned."

Another scenario could be that "the account has an insufficient balance" and could be detailed as "Given the account balance is $50, and the customer's bank card is valid, and the automated teller machine contains enough money in its cash box, when the Account Holder requests $100, then the ATM should not dispense any money, and the ATM should say there is an insufficient balance, the balance should remain at $50, and the customer's bank card should be returned."

The goal of BDD is to have a comprehensive set of scenarios for each unit of functionality. In the withdrawing cash situation, additional scenarios for "The Bank Customer's bank card has been disabled" and "The ATM does not contain enough money in its cash box" would be necessary.

The acceptance test cases are obvious from the BDD scenarios.

Acceptance criteria-based requirements specification directly addresses the requirements ambiguity problem. Natural languages are inherently ambiguous, but test case language is not. In acceptance-based criteria requirements specification, the requirements are written using test case language, which is very precise. On the other hand, this does not inherently solve the incompleteness problem. However, combining ATDD or BDD with appropriate functional test coverage criteria, such as Domain Testing, Boundary Value Analysis and Pairwise Testing (see the Software Testing KA), can reduce the likelihood of requirements incompleteness. (See also [9, c1, c12].)

4.4. Model-Based Requirements Specification

[1*, c12] [2*, c5] [4*]

Another approach to avoiding the inherent ambiguity of natural languages is to use modeling languages such as selected elements of the unified modeling language™ (UML) or systems modeling language™ (SysML). Much like the blueprints used in building construction, these modeling languages can be used in a computing technology-free manner to precisely and concisely specify functional requirements [9, c1-2]. This topic is closely related to the Software Engineering Models and Methods KA. Requirements models fall into two general categories:

1. Structural models for specifying policies to be enforced: These are logical class models as described in, for example, [9, c8]. They are also called conceptual data models, logical data models and entity-relationship diagrams.
2. Behavioral models for specifying processes to be carried out: These models include use case modeling as described in [9, c7], interaction diagrams as described in [9, c9] and state modeling as described in [9, c10]. Other examples are UML activity diagrams and data-flow modeling, as described in [1*, c12-13], [8], [10] and [18].

Model-based requirements specifications vary in the degree of model formality. Consider the following:

1. Agile modeling (see, for example, [10]) is the least formal. Agile models can be little more than rough sketches whose goal is to communicate important information rather than demonstrate proper use of modeling notations. In this type of modeling, the effect of the communication is considered more important than the form of the communication.
2. Semiformal modeling, for example [9, c6-12], provides a definition of the modeling language semantics ([9, Appendix L]), but that definition has not been formally proved to be complete and consistent.
3. Formal modeling, for example, Z, the Vienna development method (VDM), specification and description language (SDL) and [5, c7] have very precisely defined semantics that allow specifications to be mechanically analyzed for the presence or absence of specific properties to help avoid critical reasoning errors. The term *correctness by construction* has been used for development in this context. (See the Formal Methods section in the Software Engineering Models and Methods KA.)

Generally, the more formal a requirements model is, the less ambiguous it is, so software engineers are less likely to misinterpret the requirements. More formal requirements models can also be:

- more concise and compact;
- easier to translate into code, possibly mechanically;
- used as a basis for deriving acceptance test cases.

One important message in [4*] is that while formal modeling languages are stronger than semiformal and Agile modeling, formal notations can burden both the model creator and human readers. Wing's compromise is to use formally defined underpinnings (e.g., in Z) for surface syntaxes that are easier to read and write (e.g., UML statecharts).

4.5. Additional Attributes of Requirements

[1*, c27pp462-463]

Over and above the basic requirements statements already described, documenting additional attributes for some or all requirements can be useful. This supplemental detail can help software engineers better interpret and manage the requirements [6, c16]. Possible additional attributes include the following:

- tag to support requirements tracing;
- description (additional details about the requirement);
- rationale (why the requirement is important);
- source (role or name of the stakeholder who imposed this requirement);
- use case or relevant triggering event;
- type (classification or category of the requirement — e.g., functional, quality of service);
- dependencies;
- conflicts;
- acceptance criteria;
- priority (see Requirements Prioritization later in this KA);
- stability (see Requirements Stability and Volatility later in this KA);
- whether the requirement is common or a variant for product family development (e.g., [20]);
- supporting materials;
- the requirement's change history.

Gilb's Planguage (short for Planning Language) [7] recommends attributes such as scale, meter, minimum, target, outstanding, past, trend and record.

4.6. Incremental and Comprehensive Requirements Specification

Projects that explicitly document requirements take one of two approaches. One can be called *incremental specification*. In this approach, a version of the requirements specification contains only the differences — additions, modifications and deletions — from the previous version. An advantage of this approach is that it can produce a smaller volume of written specifications.

The other approach can be called *comprehensive specification*. In this approach, each version's requirements specification contains all requirements, not just changes from the previous version. An advantage of this approach is that a reader can understand all requirements in a single document instead of having to keep track of cumulative additions, modifications and deletions across a series of specifications.

Some organizations combine these two approaches, producing intermediate releases (e.g., x.1, x.2 and x.3) that are specified incrementally and major releases (e.g., 1.0, 2.0 and 3.0) that are specified comprehensively. The reader never needs to go any further back than the requirements specifications for the last major release to obtain the complete set of specifications.

5. Requirements Validation
[1*, c17] [2*, s4.5]

Requirements validation concerns gaining confidence that the requirements represent the stakeholders' true needs as they are currently understood (and possibly documented). Key questions include the following:

- do these represent all requirements relevant at this time?
- are any stated requirements not representative of stakeholder needs?
- are these requirements appropriately stated?
- are the requirements understandable, consistent and complete?
- does the requirements documentation conform to relevant standards?

Three methods for requirements validation tend to be used: requirements reviews, simulation and execution, and prototyping. (See also [5, c5] [6, c17] [9, c12].)

5.1. Requirements Reviews
[1*, c17pp332-342] [2*, c4p130]

The most common way to validate is by reviewing or inspecting a requirements document. One or more reviewers are asked to look for errors, omissions, invalid assumptions, lack of clarity and deviation from accepted practice. Review from multiple perspectives is preferred:

- clients, customers and users check that their wants and needs are completely and accurately represented;
- other software engineers with expertise in requirements specification check that the document is clear and conforms to applicable standards;
- software engineers who will do architecture, design or construction of the software that satisfies these requirements check that the document is sufficient to support their work.

Providing checklists, quality criteria or a "definition of done" to the reviewers can guide them to focus on specific aspects of the requirements specification. (See Reviews and Audits in the Software Quality KA.)

5.2. Simulation and Execution

Nontechnical stakeholders might not want to spend time reviewing a specification in detail. Some specifications can be subjected to simulation or actual execution in place of or in addition to human review. To the extent that the requirements are formally specified (e.g., in a model-based specification), software engineers can hand interpret that specification and "execute" the specification. Given a sufficient set of demonstration scenarios, stakeholders can be convinced that the specification defines their policies and processes completely and accurately. (See [9, c12].)

5.3. Prototyping
[1*, c17p342] [2*, c4p130]

If the requirements specification is not in a form that allows direct simulation or execution, an alternative is to have a software engineer build a prototype that concretely demonstrates some important dimension of an implementation. This demonstrates the software engineer's interpretation of those requirements.

Prototypes can help expose software engineers' assumptions and, where needed, give useful feedback on why they are wrong. For example, a user interface's dynamic behavior might be better understood through an animated prototype than through textual description or graphical models. However, a danger of prototyping is that cosmetic issues or quality problems with the prototype can distract the reviewers' attention from the core underlying functionality. Prototypes can also be costly to develop. However, if a prototype helps engineers avoid the waste caused by trying to satisfy erroneous requirements, its cost can be more easily justified.

6. Requirements Management Activities
[1*, c27-28] [2*, s4.6]

Requirements development, as a whole, can be thought of as "reaching an agreement on what software is to be constructed." (See Figure 1.3.) In contrast, requirements management can be thought of as "maintaining that agreement over time." This topic examines requirements management. (See also [5, c9].)

6.1. Requirements Scrubbing

The goal of requirements scrubbing [22, c14, c32] is to find the smallest set of simply stated requirements that will meet stakeholder needs. Doing so will reduce the size and complexity of the solution, thus minimizing the effort, cost and schedule to deliver it. Requirements scrubbing involves eliminating requirements that:

- are out of scope;
- would not yield an adequate return on investment;
- are not that important.

Another important part of the process is to simplify unnecessarily complicated requirements.
In waterfall and other plan-based life cycles, requirements scrubbing can be coordinated with requirements reviews for validation; scrubbing should occur just before the

validation review. In Agile life cycles, scrubbing happens implicitly in iteration planning; only the highest-priority requirements are brought into a sprint (iteration).

6.2. Requirements Change Control [1*, c28] [2*, s4.6]

Change control is central to managing requirements. This topic is closely linked to the Software Configuration Management KA. (Refer to that chapter for more information.)

Projects using waterfall or other plan-based life cycles should have an explicit requirements change control process that includes:

- a means to request changes to previously agreed-upon requirements;
- an optional impact analysis stage to more thoroughly examine benefits and costs of a requested change;
- a responsible person or group who decides to accept, reject, or defer each requested change;
- a means to notify all affected stakeholders of that decision;
- a means to track accepted changes to closure.

All stakeholders must understand and agree that accepting a change means accepting its impact on schedule, resources and/or commensurate change in scope elsewhere in the project. Ideally the change in scope should be objectively quantifiable, i.e., in terms of functional size units.

In contrast, requirements change management happens implicitly in Agile life cycles. In these life cycles, any request to change previously agreed-upon requirements becomes just another item on the product backlog. A request will only become "accepted" when it is prioritized highly enough to make it into an iteration (a sprint). (See also [5, c9] [22, c17].)

6.3. Scope Matching

Scope matching [22, c14] involves ensuring that the scope of requirements to architect, design and construct does not exceed any cost, schedule or staffing constraints on the project. When requirements scope exceeds the cost, schedule or staffing constraints, then either that scope must be reduced (presumably by removing a sufficient number of the lowest-priority requirements), capacity must be increased (by extending the schedule or increasing the budget and/or staffing), or some appropriate combination thereof must be negotiated. Where possible, scope matching should be quantitative instead of qualitative, i.e., in terms of functional size units.

In waterfall and other plan-based life cycles, scope matching can be coordinated with requirements validation; the scope matching should occur just before the validation review. In Agile life cycles, as long as some variant of *velocity-based sprint planning* is done, then the only work allowed into a sprint/iteration will be the work that can reasonably be expected to be completed during that sprint/iteration.

7. Practical Considerations

7.1. Iterative Nature of the Requirements Process [2*, s4.2]

Requirements for typical software not only have wide breadth; they must also have significant depth. The tension created by simultaneous breadth-wise and depth-wise requirements in real-world projects often prompts teams to perform requirements activities iteratively. At some points, elicitation and analysis favor expanding the breadth of requirements knowledge, while at other points, expanding the depth is called for. In practice, it is highly unlikely that all requirements work can be done in a single pass through the subject matter. (See also [6, c2, c9].)

7.2. Requirements Prioritization [1*, c16]

Prioritizing requirements is useful throughout a software project because it helps focus software engineers on delivering the most valuable functionality soonest. It also helps support intelligent trade-off decisions involving conflict resolution and scope matching. Prioritized requirements also help in maintenance beyond the initial development project itself. Defects raised against higher-priority requirements should probably be repaired before defects raised against lower-priority ones.

A variety of prioritization schemes are available. Answering a few key questions can help engineers choose the best approach. The first question is "What factors are relevant in determining the priority of one requirement over another?" The following factors might be relevant to a project:

- value; desirability; client, customer and user satisfaction;
- undesirability; client, customer and user dissatisfaction (Kano model, below);
- cost to deliver;
- cost to maintain over the software's service life;
- technical risk of implementation;
- risk that users will not use it even if implemented.

The Kano model, which underlies [6, c17], shows that considering only value, desirability or satisfaction can lead to erroneous priorities. A better understanding of priorities comes from considering how unhappy stakeholders would be if that requirement were not satisfied. For example, consider a project to develop an email client. Two candidate requirements might relate to:

1. Having an effective spam filter
2. Handling attachments on emails

Prioritization must weigh both the satisfaction users will experience from having certain features and the dissatisfaction they will experience if they lack certain features. For example, users are more likely to be happy with an effective spam filter than with the ability to handle attachments, so the spam filter would be given a higher priority based on the satisfaction criterion. On the other hand, the inability to handle attachments would make many users extremely unhappy — much more so than not having an effective spam filter. When considering happiness, or satisfaction, from implementing features combined with unhappiness (or dissatisfaction) from not implementing certain features, developers would generally give handling attachments a higher priority than the effective spam filter.

The second key question is "How can we convert the set of relevant factors into an expression of priority?" The formula

$$Priority = \frac{(Value * (1\text{-}Risk))}{Cost}$$

is just one example of an *objective function* to do so. The choice of measurement schemes for the relevant factors can impose constraints on the objective function. (See Measurement Theory in Computing Foundations).

Once the priority of the requirements has been determined, those priorities must be specified in a way that can be communicated to all stakeholders. Several ways to do this are possible, including the following:

- enumerated scale (e.g., must have, should have, nice to have);
- numerical scale (e.g., 1 . . . 10);
- Lists that sort the requirements in decreasing priority order.

Effective requirement prioritization focuses on finding groups of requirements with similar priorities rather than creating overly rigorous measurement scales or debating small differences.

7.3. Requirements Tracing [1*, c29]

Requirements tracing can serve two potentially useful purposes. One is to serve as an accounting exercise that documents consistency between pairs of related project work products. An important question might be "For each identified software requirement, are there identified design elements intended to satisfy it?" If no identified design elements can be found, then either that requirement is not satisfied in that design or the design is correct and one or more stated requirements can be deleted. Similarly, "For each identified design element, are there identified requirements that cause it to exist?" If no identified requirements can be found, then either that design element is unnecessary or the stated requirements are incomplete.

The other purpose is to assist in impact analysis of a proposed requirement change. If a particular system requirement were to change, for example, that system requirement could be traced to its linked software requirements. Not all linked software requirements would need to change. But each software requirement that would be affected could be traced to its linked design elements. Again, not all linked design elements would need to change. But each design element affected could be traced to the linked code. The affected software requirements, design elements and code units could also be traced to their linked test cases for further impact analysis. This helps establish a "footprint" for the volume of work needed to incorporate that change to the system requirement.

Software requirements can be traced back to source documentation such as system requirements, standards documents and other relevant specifications. Software requirements can also be traced forward to design elements and requirements-based test cases. Finally, software requirements can also be traced forward to sections in a user manual describing the implemented functionality. (See also [23].)

7.4. *Requirements Stability and Volatility* [2*, s4.6]

Some requirements are very stable; they will probably never change over the software's service life. Some requirements are less stable; they might change over the service life but might not change during the development project. For example, in a banking application, requirements for functions to calculate and credit interest to customers' accounts are likely to be more stable than requirements to support different tax-free accounts. The former reflects a banking domain's fundamental feature (that accounts can earn interest). At the same time, the latter may be rendered obsolete by a change in government legislation. Finally, some requirements can be very unstable; they can change during the project — possibly more than once. It is useful to assess the likelihood that a requirement will change in a given time. Identifying potentially volatile requirements helps the software engineer establish a design more tolerant of change, (e.g., [20]). (See also [9, c4].)

7.5. *Measuring Requirements* [1*, c19]

As a practical matter, it may be useful to have some concept of the *volume* of the requirements for a particular software product. This number is useful in evaluating the *size* of a new development project or the size of a change in requirements and in estimating the cost of development or maintenance tasks (e.g., [9, c23]), or simply for use as the denominator in other measurements. Functional size measurement (FSM) is a technique for evaluating the size of a body of functional requirements. Story points can also be considered a measure of requirements size.

Additional information on size measurement and standards can be found in the Software Engineering Process KA.

Many quality indicators have been developed that can be used to relate the quality of software requirements specification to other project variables such as cost, acceptance, performance, schedule and reproducibility. Quality indicators for individual software requirements and a requirements specification document as a whole can be derived from the desirable properties discussed in Section 3.1, Basic Requirements Analysis, earlier in this KA.

7.6. *Requirements Process Quality and Improvement* [1*, c31]

This topic concerns assessing the quality and improvement of the requirements process. Its purpose is to emphasize the key role of the requirements process in a software product's cost and timeliness and in customer satisfaction. Furthermore, it helps align the requirements process with quality standards and process improvement models for software and systems. Process quality and improvement are closely related to both the Software Quality KA and Software Engineering Process KA, comprising the following:

- requirements process coverage by process improvement standards and models;
- requirements process measures and benchmarking;
- improvement planning and implementation;
- security/CIA (confidentiality, integrity, and availability) improvement/planning and implementation.

8. Software Requirements Tools [1*, c30]

Tools that help software engineers deal with software requirements fall broadly into three categories: requirements management tools, requirements modeling tools and functional test case generation tools, as discussed below.

8.1. Requirements Management Tools

[1*, c30pp506-510]

Requirements management tools support various activities, including storing requirements attributes, tracing, document generation and change control. Indeed, tracing and change control might only be practical when supported by a tool. Because requirements management is fundamental to good requirements practice, many organizations have invested in tools. However, many more manage their requirements in more ad hoc and generally less satisfactory ways (e.g., spreadsheets). (See also [5, c8].)

8.2. Requirements Modeling Tools

[1*, c30p506] [2*, s12.3.3]

At a minimum, a requirements modeling tools support visually creating, modifying and publishing model-based requirements specifications. Some tools extend that by also providing static analysis (e.g., syntax correctness, completeness and consistency). Formal analysis requires tool support to be practicable for anything other than trivial systems, and tools generally fall into two categories: theorem provers or model checkers. In neither case can proof be fully automated, and the competence in formal reasoning needed to use the tools restricts the wider formal analysis. Some tools also dynamically execute a specification (simulation).

8.3. Functional Test Case Generation Tools

The more formally defined a requirements specification language is, the more likely it is that functional test cases can be at least partially derived mechanically. For example, converting BDD scenarios into test cases is not difficult. Another example involves state models. Positive test cases can be derived for each defined transition in that kind of model. Negative test cases can be derived from the state and event combinations that do not appear. (See Section 8.2, Testing Tools in the Testing KA, for more information.) A process for deriving test cases from UML requirements models can be found in [9, c12].

In the most general case, such tools can only generate test case inputs. Determining an expected result is not always possible, additional business domain expertise might be necessary.