SE 463 Course Notes

Software Requirements Specification and Analysis

Michael Socha

4A Software Engineering
University of Waterloo
Spring 2018

Contents

1 Course Overview

1.1 Logistics

- **Professor:** Joanne Atlee

1.2 Overview of Topics

Many software project failures are not due to technical reasons, but rather due to poor project specifications. This course focuses on requirement-related topics including:

- Different types of requirements
- Learning about stakeholder needs
- Analyzing and refining preliminary requirements
- Expressing requirements in various representations
- Managing requirement changes

2 Business Model

2.1 Traditional Product Model

A traditional "waterfall" style of development relies heavily on the initial implementation meeting customer needs. This approach often leads to requirements-related problems, since the business and development team might not understand the customer needs or the customer needs might change, and a waterfall model offers little room to refine these requirements.

2.2 Lean Product Development Model

A lean product development model focuses on first determining whether a project addresses an important problem (e.g. problem fit), and whether the project provides a solution people want (market fit). Only after these are established is a product built, which is usually nimble at first and then scales up.

2.3 Lean Canvas

A lean canvas is a template to help determine the initial requirements for a project, especially one in the new product or consulting space. The sections of a lean canvas are described below:

2.3.1 Customer Segments

This is a list of potential customers and users. It helps to have granular segments, and focus should be placed on segments likely to become early adopters.

2.3.2 Problem Segments

This is a list of a few top problems that each customer segment wants solved.

2.3.3 Unique Value Proposition

This is a reason why the project is unique and valuable. The focus should also be on early adopters.

2.3.4 Solution

This is a list of solutions early adopters already apply to solve this problem. From this list, some key capabilities and features of the software project can be determined.

2.3.5 Channels

This is a list of methods to reach the project's target customers. This is important to consider early, since these customers are not just useful as end users, but also as subjects of testing and experiments during development.

2.3.6 Revenue Streams and Costs

Revenue and costs should be considered early in a project. If the project is going to operate on a subscription model, it is a good idea to charge something from the start (with the exception of a free trial period). "Freemium" business models, in which core services are free, tend to have low conversion rates to paying features.

2.3.7 Key Metrics

This is a list of metrics to be used for measuring success.

2.3.8 Unfair Advantage

This refers to some sort of competitive advantage that is hard to replicate. Examples include an existing reputation or customer base, proprietary data, or patented work.

3 Hypothesis Testing

The purpose of hypothesis testing is to validate or disprove guesses in a project's business model. Hypothesis testing is generally conducted by performing pass/fail experiments through market research or customer interviews. Hypothesis being disproved can lead to learning, after which the hypothesis can be pivoted based on new understanding.

3.1 Hypothesis

A hypothesis is a tentative guess about a phenomenon of interest that is testable and falsifiable. The phenomena of interest (e.g. characteristics of customer problem) are considered to be dependent variables, while independent variables affect the phenomenon of interest. A good hypothesis must be testable (i.e. possible to observe effects of independent variables on dependent variables) and falsifiable.

3.1.1 Falsifiable Hypothesis

A falsifiable hypothesis is a statement that can easily be proven wrong. Specific and testable statements are more falsifiable than vague ones. Falsifiable hypothesis often have somewhat of a universally quantification aspect (e.g. all, most, etc), rather than merely existential quantification.

3.2 Testing Risks

Uncertainty simply means that there are multiple possibilities. Risk tends to involve uncertainty, with some of the possibilities leading to failure. The three largest risks to a project are:

- **Customer Risk** e.g. Is there a viable customer segment, and what are their risks?
- **Product Risk** e.g. How do customers rank the top three problems?
- **Market Risk** e.g. How do customers solve these problems today?

3.3 Problem Interview

Interviews to test hypotheses generally follow the plan below:

- Welcome
- Collect demographics (i.e. test customer segment)
- Tell a story (i.e. set a problem context)

- Problem ranking
- Explore customer's worldview
- Wrapping up (i.e. hook and ask)
- Document results

3.4 Processing Results

It is important to home in on early adopters. As more information is gathered, it may help to refine the problem or pivot the hypothesis. It helps to continue gathering results until they are fairly consistent. Good indications that enough testing has been done are that the demographics of an early adopter can be identified, there is a must-have (i.e. critical problem to work on first) problem, and there is an understanding of how customers currently solve this problem.

4 Stakeholders

A stakeholder is considered to be anyone who has a stake in a project's ultimate success or failure. Stakeholders an include the development team, people in the operational work area, the containing business, and forces from the outside world.

4.1 Owners/Clients/Champions

The owner/client is the group paying for the software to be developed. They are usually considered to be the ultimate/champion stakeholder, since they often have the final say in a project. Examples include a client in a consultancy project, or the company that developers are working for in an internal project.

4.2 Customers

A customer buys a project after its completion. The customer may be the same person as the owner or user.

4.3 Users

Users tends to be experts on the work the system is performing, and experts on any existing systems or competing products. Users typically have specific needs a product should satisfy.

4.3.1 User Classes

To meet various user needs, it helps to categorize users by their differences, including:

- Access privilege and security levels
- Tasks regularly performed
- Features used
- Frequency of use
- Application domain expertise, technical expertise
- Platform of use
- Preferred language
- Disfavored users, which are users who should not have access for security, legal or safety reasons (e.g. kids on some social media platforms)

4.4 Personas

Personas are resemblances of actual users that can be constructed when real users are hard to interview (e.g. not numerous, too numerous). Personas should imitate the key details of important user classes. Enough details should be provided to make the persona seem realistic. Effectively built personas can:

- Guard against building a product just from the developer's perspective
- Guard against adapting to "elastic" users, and instead focus on keeners and early adopters

4.5 Domain Expert

Domain experts understand the problem domain very well, are familiar with typical users, their expectations, and potential deployment environments. Note that domain experts need not know much about software engineering.

4.6 Software Engineer

A software engineering is an expert on the project's development technologies. These stakeholders can represent the development teams they run, and are responsible for overseeing the progress of the technical and some economic aspects of the project. Software engineering can also educate customers how available technology can affect the requested functionality.

4.7 Other Stakeholders

Other potential stakeholders include:

- **Inspectors:** Experts on government and safety regulations
- **Market researchers:** Can serve as proxy for customer interviews
- **Lawyers:** Familiar with legal requirements and licensing
- **Experts on adjacent systems:** Can explain how adjacent systems can affect the project
- **Negative stakeholders:** Stakeholders that want a project to fail

5 User Requirements

5.1 Use Cases

A use case represents some sort of end-to-end functionality in a system, capturing both its triggering event and the complete system response. Time-triggered use cases feature time as their triggering event. Well-defined use cases should not overlap much with one another.

5.2 Actors

An actor is an entity that interacts with the described system, which may include users or other systems. A supporting actor provides some service to the described system.

5.2.1 Actor Generalization

Actors often share common use cases with one another. In these cases, actor generalization can be applied, which factors out common behavior as an abstract actor.

5.3 Modifying Use Cases

- <<include>> is used to add a sub use case that is used in multiple other use cases
- <<extend>> is used to extend or replace the end of an existing use case

Note that well-described systems tend to use these modifiers sparingly.

5.4 Use-Case Descriptions

Textual formats known as use-case descriptions are commonly used to represent use-cases. These descriptions vary in the amount of detail provided for each use case.

5.5 User Stories

User stories are an alternative to use-case descriptions for describing a system. Instead of being told from the system's perspective, user stories provide a description of something a user wants to be able to do (i.e. user requirements). The parts of a successfully managed user story are:

- **Card:** The initial description, often phrased as "As a <role>, I want <something> so that <benefit>"
- **Conversation:** Discussion with product owner to determine requirement details

- **Confirmation:** Criteria for determining whether an implementation meets requirements (conditions of satisfaction are described from the system's perspective)

5.5.1 Benefits of User Stories

The benefits of user stories include:

- Easy for stakeholders to understand
- User-focused style encourages discussion more than ordinary written documentation
- Encouraging iterative development when stories are sized accordingly

5.6 Changing Requirements

System requirements often change during the development process. Below are key topics to consider when managing requirement changes.

5.6.1 Requirements Baseline

A requirements baseline is a set of formally reviewed and agreed core requirements for a system. Major requirement changes should pass through this review process, and it may be helpful to limit the rate of change to these requirements.

5.6.2 Unique Value Proposition (UVP)

New requirements should not weaken a project's UVP.

5.6.3 Project Scope

When project scope is not managed well, there is a tendency for "scope creep" to occur. Scope creep can be mitigated by keeping a prioritized list of requirements, and only adding a few major ones to each release.

6 Workflow Models

A use case can be decomposed into an ordered sequence of tasks known as a workflow. Each task in a workflow specifies the data it requires and the data it produces.

6.1 Scenarios

A scenario is one full execution path through a use case, listing only observable actions between a system and external actors. A use case can be considered to be a collection of success and failure scenarios initiated by an external actor to achieve a particular goal. Below is an example scenario for a banking system:

1. User requests to withdraw funds, specifies amount.
2. Bank authenticates user.
3. Bank checks account has sufficient funds.
4. Bank dispenses cash and receipt.
5. Bank records the transaction.
6. User takes case and receipt, leaves.

A general template for a scenario includes:

- **Business event name**
- **Business use case name/reference**
- **Trigger**
- **Preconditions**
- **Interested stakeholders**
- **Active stakeholders**
- **Normal case steps**
- **Alternatives** - Sub use cases that capture the main goal of this use case through a different sequence of actions
- **Exceptions** - Sub use cases that capture an unwanted deviation
- **Postconditions**

6.1.1 More Complex Actions

Scenario actions can include more complex control flow, such as branching and looping. However, this should be used sparingly, and is often a good indicator that the scenario is

becoming too detailed or all-encompassing.

6.2 Workflow Diagrams

6.2.1 Activity Diagram

Activity diagrams are UML graphical representations of a system's flow control between activities and actions.

6.2.2 Dataflow Diagram (DFD)

DFDs are graphical representations of the functional decomposition of an information system. Each step in a described workflow states what information it requires along with its source, and what information is produced along with its destination.

7 MVP (Minimum Viable Product) Hypothesis Testing

It is important to run experiments to assess the risks of a solution. Areas of risk to assess include customer risk (e.g. who the early adopters are, how they will be reached), and product risk (e.g. whether the solution and UVP align with the customers' main problem, and what the minimum set of features required to launch is).

7.1 Solution Interview

7.1.1 Interview Flow

An interview designed to test the risks of a MVP solution can follow the general flow below:

1. **Welcome:** Set the stage for how the interview works.
2. **Collect demographics:** Try to determine how the interviewee approaches the problem at hand (e.g. for a job search platform, learn about their job search habits).
3. **Tell a story:** Illustrate the top problems the MVP is trying to solve through a brief story.
4. **Demo (Test Solution):** For each problem, go through the workflow and determine whether it addresses the interviewee's main problems. This may appear to be a pitch, but its goal is learning more so than marketing. Attention should be paid to the interviewee's attitude and reactions. Unexpected reactions provide great learning opportunities and should be examined.
5. **Wrapping up:** Ask for permission to follow up, as well as for referrals for other prospective interviewees.

7.1.2 Interview Followup

Based on the results, the earlier pivot hypothesis may need to be pivoted again, after which features may be added/removed/amended. Exit criteria for this series of interviews includes:

- Can identify demographics of early adopters
- Can identify minimum features needed to solve problem
- MVP is significant improvement over existing solutions

7.2 MVP Advice

Before an MVP is built, a UVP should be determined that provides significant value. The MVP should deliver on the UVP, and focus on addressing the customers' number one problem. The MVP should not focus at first on optimizing the solution, but rather on accelerating the learning process to facilitate continual improvement.

8 Requirements Elicitation

Requirements elicitation is the process of collecting requirements for a system. These requirements can be gathered from many types of sources, including stakeholders, existing documentation, similar or previous systems, and the environment of a system.

8.1 Artifact-based Elicitation

The key idea behind artifact-based elicitation is to gather requirements from existing artifacts (e.g. documentation, existing systems) instead of relying on input from stakeholders. Elicitation techniques that are considered artifact-based include document studies, analyzing similar companies, norm studies, domain analysis, and requirements taxonomy.

8.1.1 Documents

Classes of documents that can be used for analysis include:

- **System documentation:** Can include design documents, bug reports, change requests, user manuals, work procedures, usage statistics, marketing data, and performance figures.
- **Environment documentation:** Can include organizational charts, business plans, policy manuals, financial reports, meeting overviews.
- **Domain analysis:** Can include textbooks, surveys, and standards and regulations documentations.

8.1.2 Norms

Requirements can be gathered by forming a comparison to a different existing system (i.e. statements of the form "Build a better X").

8.1.3 Requirements Taxonomies

Requirements taxonomies are requirements classifications that can act as a list of requirements to be elicited. A sample taxonomy for information systems is an acronym known as PIECES (Performance, Information and data, Economy, Control, Efficiency, and Services). Taxonomies can be classified as being domain-specific (e.g. PIECES is for information systems) or domain-independent.

8.2 Stakeholder-based Elicitation

The key idea behind stakeholder-based elicitation is to gather detailed requirements information specific to particular problems or stakeholders. Elicitation techniques considered to be stakeholder-based include stakeholder analysis, questionnaires, interviews, observations, task demos, and personas.

8.2.1 Interviews

Interviews can be useful for gathering information specific to particular stakeholders, such as their specific problems or potential areas of conflict. To gather more information, it helps to ask open questions and allowing the stakeholder sufficient time to formulate a response. Examples of open questions include "Why would you use this product?", or "What problems do you expect this product to solve for you?".

Interviews in groups can help in sharing more ideas, taking the spotlight off specific individuals, and pre-emptively identifying conflicts among stakeholders. Common mistakes during interviews include not interviewing the right people, asking directed questions early on, letting one person dominate a group discussion, or allowing the interview results to over-specify requirements.

8.2.2 Questionnaires

Questionnaires can be useful in gathering insights from a wide group of people, and can consist of open question (to gather suggestions) or closed questions (to gather opinions). A major limitation of questionnaires is that questions need to be fairly concrete, so although gathering large amounts of data can be effective, new insights are hard to obtain.

8.2.3 Ethnographic Analysis

Ethnographic analysis is a study of user behavior through observation, and can be useful in determining the human/social factors in a system that may be hard to gather from interviews alone. Ethnographic analysis can be useful in identifying the critical features of a system, though tends to focus on existing rather than new solutions.

8.2.4 Apprenticeship

An apprentice can gather requirements by spending time with an expert in some field. Information can be gathered through observation, asking questions, and performing some tasks under the expert's supervision.

8.2.5 Personas

Personas have been previously described; they can be useful if it is not feasible to interview actual users.

8.3 Model-based Elicitation

The key idea behind model-based elicitation is to express requirements languages useful for a particular purpose, which can help raise new insights and concerns. Elicitation techniques considered to be model-based include analysis patterns, mockups and prototyping, and pilot experimentation.

8.3.1 Requirements Models

A model is a simplified representation of a system that captures key details. Models can be used in requirements analysis to guide elicitation, help uncover problems, and provide a measure of progress. Re-expressing requirements through formal models can often reveal problems such as omissions and contradictions.

8.3.2 Mockups and Prototypes

Mockups and prototypes provide the essence of a proposed solution, and are often used to encourage further feedback and requirements elicitation from stakeholders.

8.4 Creativity-based Elicitation

The key idea behind creativity-based elicitation is to come up with new requirements that bring about a competitive advantage through innovation. Elicitation techniques considered to be creativity-based include systemic thinking, brainstorming, creativity workshops, and constraint relaxation.

8.4.1 Systemic Thinking

The goal of systemic thinking is to enlarge the scope of a problem under study from just a system to the outside world at large. The goal of systemic thinking is to gain an understanding of *what* a topic of interest is, with attention paid to its observable behaviors and information flows rather than implementation details.

8.4.2 Brainstorming

Brainstorming is a group activity designed to generate a large number of new and creative ideas. Brainstorming sessions start with an idea generation phase, where participants are encouraged to produce many new ideas, and judgement is suspended. Later, possibly with a new set of participants, the generated ideas are analyzed, and some can be expanded on even further.

8.4.3 Creativity Workshops

Creativity workshops are designed to be spaces for creating new ideas that go above and beyond original stakeholder requirements. Techniques for generating new ideas include brainstorming, constraint relaxation, and analogical reasoning.

9 Domain Models

A domain model is a model of the operating environment of some work under study. Domain models describe real-world entities and their interrelationships, and also capture information about the subject matter and environmental phenomena.

9.1 Unified Modeling Language (UML)

UML is a general-purpose modeling language for software systems. UML provides specifications for various types of structural diagrams (e.g. class diagrams, object diagrams) and behavioral diagrams (e.g. use case diagrams, activity diagrams, state machine diagrams).

UML class diagrams are a good choice for representing domain models. When forming a UML class diagram based on requirements, it is typically best to leave out details; information based on requirements includes class names, key attribute names and association/roles names, while most other information is based on design.

9.2 Conceptual Classes

Conceptual classes provide a high-level overview of what a class does. Conceptual class category examples include business transactions (e.g. FlightReservation), events (e.g. FlightDeparture) and descriptions of things (e.g. FlightDescription).

9.3 Finding Domain Entities

The main ways of finding domain entities include:

- **Refining existing domain models.** Either refining models for previous systems or general domain models.
- **Using a conceptual class list.** Can be generated by brainstorming what kinds of entities and entity classes are relevant to a system.
- **Identifying noun phrases.** Nouns in system descriptions often correspond to entities.

9.4 Attributes

An attribute is a sub-part of an object describing the characteristics or properties of object instances. When forming class diagrams based on requirements, attributes can initially be modeled as just a name, with implementation details added later.

9.5 Associations

Associations represent relevant relationships between multiple conceptual classes. Physical (e.g. "part of") relationships are typically good candidates, while transient (e.g. "communicates with") relationships tend to be weaker. Where appropriate, associations can be named, typically with short verbs.

9.5.1 Roles and Multiplicities

A role is one end of a relationship, describing how an entity is viewed within the context of the relationship. Roles can also state multiplicities of association.

9.5.2 Association Classes

Should an association contain attributes of its own, it can be represented with an association class. The state of an association class should only be known by the instances of classes forming the relationship, and each of these objects should have at most one instance of the association class.

9.5.3 Association Qualifiers

Association qualifiers are an association attribute that can be used at one end of an association to distinguish among a set of objects at the other end of the association. An example is a filename serving as an association attribute for a set of files stored in a directory.

9.6 Composition/Aggregation

Composition is a stronger type of association than aggregation. Composition implies that an object cannot exist without its container object (i.e. there is a strong relation to at most one container), while aggregated objects may be weakly related to several containers.

9.7 Inheritance/Generalization

Inheritance is useful for representing "is a kind of" relationships (e.g. an Engineer is a kind of Employee). Properties such as attributes and association of parents can be inherited by children, which can implement abstract features, add new functionality, or override existing functionality. Non-leaf nodes may be abstract classes, which are indicated by italicized names.

9.8 Common Modeling Conundrums

9.8.1 Tightly Coupled Classes

It may be tempting to combine classes with multiple one-to-one composition associations into a single class. However, if the class associations are dynamic, the classes should be kept distinct.

9.8.2 Similar but Distinct Classes

Multiple distinct concepts should not be mixed in a single class. This is particularly important when the concepts have similar names, since this can make them tempting to combine but also produces a confusing model.

9.8.3 Representing Object Identity

Objects should have an identity such that they can be distinguished from other objects with the same values. As such, it is typically a poor idea to base an object's identity on its properties (e.g. identify a house on a street by its street name).

9.8.4 Temporal Objects

In some situations, it is appropriate for a model to store a history of values for an object rather than just the current value.

9.8.5 Class-Scope Attributes

Class-scope attributes have a changeable value that is shared by all object instances of a class. Class-scope attributes can lead to trouble if different instances require that a different value be stored.

10 Prototyping

A prototype is a partial implementation designed to help stakeholders learn more about a problem or its solution.

10.1 Prototyping Categories

Prototypes can be divided into different categories, including:

- **Presentation prototypes**, which are used as a proof of concept to explain certain features/designs and then thrown away.
- **Exploratory prototypes**, which are used to learn more about the problem (e.g. elicit needs, clarify goals), and are also thrown away.
- **Experimental/Breadboard prototypes**, which are used to explore technical feasibility, typically with little to no user/customer involvement.
- **Evolutionary/Operational prototypes**, which is an early deliverable in a continuous development process that gets improved on later.

10.2 Prototyping Methodologies

10.2.1 Throwaway Prototyping

Throwaway prototypes are used to learn more about a problem, after which they are discarded and not used in a final product. The development of these systems is designed to be done rapidly, with a horizontal strategy often applied that only builds up a single layer at a time (e.g. only build UI for UI-related experimentation). Throwaway prototyping can be useful for gathering information early on (low risk). However, it may result in a lack of formal documentation, unrealistic customer expectations, and in some situations with major time constraints can end up getting developed into a final product (which typically leads to poor design).

10.2.2 Evolutionary Prototyping

Evolutionary prototyping involves learning about a solution to a problem. Emphasis is placed on the "risky" parts of a solution first and the system is developed incrementally, typically with a vertical strategy that builds up all layers feature-by-feature. Evolutionary prototypes tend to be good at evolving to requirements changes, and can be reverted to their previous increment should some change introduce major problems. However, evolutionary prototypes run the risk of poor early architectural choices, poorly structured systems, and often lack high-level direction and control.

10.3 Sketching and Wireframing

Sketches of a UI focus on content, and include the scope of what should be included on a display and how it should be organized. Wireframes are a visual representation of a UI that focus on content hierarchy and flow rather than presentation details (e.g. color, fonts, images, etc.). Sketching and wireframing both allow for the rapid generation of UI designs, which can be iterated on and refined further.

11 Quality Requirements

Functional requirements describe what a system is supposed to do (e.g. what services are provided, I/O behavior). Quality requirements describe some additional constraints for the system.

Common examples of quality requirements include:

- **Performance** (e.g. execution speed, response time, throughput)
- **Reliability** (e.g. fault tolerance, mean time to failure (MTTF))
- **Robustness** (e.g. tolerance to invalid inputs, performance under stress)
- **Adaptability** (e.g. ease of adding new functionality, reusability in other environments)
- **Security** (e.g. controlled access to data, protection against theft)
- **Usability** (e.g. ease of learning, user productivity)
- **Scalability** (performance as workload increases in terms of data size, number of users, etc.)
- **Efficiency** (with respect to utilization of resources)
- **Accuracy/Precision** (e.g. precision of computations, tolerance for computation errors)

11.1 "Motherhood" Requirements

The term motherhood requirements refers to system requirements that are all-encompassing and do not differentiate well between different systems (e.g. "reliable", "user-friendly", "maintainable"). Virtually every system must meet such requirements to some degree; what differentiates systems is the degree to which these attributes are required, and their relative importance.

11.2 Fit Criteria

Fit criteria quantify the extent to which a quality requirement must be met. Examples include:

- System must be down no more than 5 minutes a year.
- 90% of users should be able to create an account within 2 minutes.

11.2.1 Richer Fit Criteria

Richer fit criteria are a way of organizing fit criteria that includes a target metric, a minimum metric, and a high-achieving "outstanding" metric. An example is that a system has a minimum response time of 1s, a target response time of 0.5s, and an outstanding response time of 0.1s.

11.3 Measurements

Effectively applying fit criteria can allow for measurement of quality requirements. However, not all fit criteria can be measured on a system prototype or new systems in general (e.g. criteria that concern large amounts of users or long amounts of time). In these cases, possible approaches to measurement include:

- Measuring attributes of prototype.
- Estimating quality attributes.
- Delivering a system and performing more accurate quality measurements later.

11.3.1 Monte Carlo Techniques

Monte Carlo techniques are a class of computational techniques that apply random sampling to generate estimates. Such techniques can be used in estimating various aspects of a system's requirements. A sample application is deliberately planting N bugs in a system, and then measuring how many are reported by developers or testers. The resulting bugs to reported bugs ratio can be used to give a rough estimate of the total number of unreported bugs in the system.

11.4 Prioritizing Quality Requirements

Different quality requirements can come into conflict with one another. Common areas of conflict include maintainability vs robustness, performance vs security, and performance vs reuse. Conflicts may also occur between quality and functional requirements (e.g. performance vs certain features). In such cases, requirements need to be prioritized. This is expanded on further in the next section.

12 Prioritizing Requirements

A system may have more requirements than can actually be implemented (e.g. too many requirements or requirements that conflict with one another). In such cases, requirements must be balanced with a project's limited resources in mind. This may be an issue of determining what features to add to an MVP or to a periodic release.

12.1 Prioritizing Criteria

A requirement's value is its potential contribution to customer satisfaction. Requirements can be prioritized by their value to cost ratios.

12.2 Grouping Requirements

A common prioritization technique is grouping requirements into 3-4 groups based on their urgency. An example of a set of groups may be "Critical", "Standard", and "Optional".

12.3 Ranking Requirements

It is possible to stack rank all requirements, though this gives little info on the relative differences between ranked requirements. This motivates the development of different ranking techniques.

12.3.1 100-Dollar Test

In this ranking system, stakeholders are given 100 "dollars" (i.e. points) to distribute among a project's requirements.

12.3.2 Kano Model

Kano models serve to analyze requirements based on customer perception. Kano charts have lines for features over two axis, one for customer satisfaction and one for execution. The shape of the lines can provide information on the type of feature, with the main types being:

- **Basic:** Poor execution and low satisfaction to good execution and neutral satisfaction indicates functionality taken for granted.
- **Performance:** Poor execution and low satisfaction to good execution and high satisfaction indicates a core requirement the customer expects.
- **Excitement:** Poor execution and neutral satisfaction to good execution and high satisfaction indicates functionality that is useful but not expected.

- **Indifferent:** Poor execution and neutral satisfaction to good execution and neutral satisfaction indicates a requirement the customer does not care about.

Kano surveys can be run to collect data to construct Kano models. The purpose of such surveys is to gauge customer reaction if a particular feature is included/excluded from a system. Generally speaking, the importance of features is Basic > Performance > Excitement > Indifferent.

12.3.3 Analytical Hierarchy Process (AHP)

AHP is a requirement prioritization technique based on stakeholders' pairwise comparisons of requirements. These comparisons are used to produce a relative ranking of all requirements. This technique is based on the idea that absolute values of requirements are hard to judge, and small comparisons are usually easier to manage. AHP analysis consists of the following steps:

1. Form a square matrix that stores a score between all pairs of requirements. If one comparison of requirements $R1$ and $R2$ is assigned a score of A, then the reciprocal comparison $R2$ and $R1$ is assigned a score of $\frac{1}{a}$. An example is below.

	Req1	Req2	Req3	Req4
Req1	1	$\frac{1}{3}$	2	4
Req2	3	1	5	3
Req3	$\frac{1}{2}$	$\frac{1}{5}$	1	$\frac{1}{3}$
Req4	$\frac{1}{4}$	$\frac{1}{3}$	3	1

2. Normalize the columns of the matrix.

	Req1	Req2	Req3	Req4
Req1	0.21	0.18	0.18	0.48
Req2	0.63	0.54	0.45	0.36
Req3	0.11	0.11	0.09	0.04
Req4	0.05	0.18	0.27	0.12

3. Sum up reach row.

Sum
1.05
1.98
0.34
0.62

4. Report the normalized, relative values of each requirement.

Values
26%
50%
9%
16%

To check for consistency, the following steps can be applied:

1. Multiply the comparison matrix by the priority vector. Using the above matrix and vector, this should yield:
$$\begin{bmatrix} 1.22 \\ 2.18 \\ 0.37 \\ 0.64 \end{bmatrix}$$

2. Divide each element by the corresponding element in the priority matrix.
$$\begin{bmatrix} 4.66 \\ 4.40 \\ 4.29 \\ 4.13 \end{bmatrix}$$

3. Compute the principle eigenvalue.
$$\frac{4.66 + 4.40 + 4.29 + 4.13}{4} = 4.37$$

4. Calculate the consistency index.
$$CI = \frac{4.37 - n}{n - 1} = \frac{4.37 - 4}{4 - 1} = 0.12$$

5. Compare against consistency index of random matrix (this value can be found in a lookup table). Ideally, this consistency index should be below 0.10.
$$CR = \frac{0.12}{0.90} = 0.14$$

12.4 Challenges

Common challenges faced when prioritizing requirements include:

- Many requirements deemed essential
- Large number of requirements to prioritize
- Conflicting priorities
- Changing priorities

- Effective collaboration between developers and stakeholders
- Subjective prioritization

12.5 Benefits

Benefits commonly gained from requirements prioritization include:

- Focusing on important requirements first, which improves customer satisfaction.
- Determining how to allocate limited project resources.
- Encouraging stakeholders to have a broader view of project requirements (i.e. consider other stakeholder requirements, not just their own).

13 Risks

A risk is an uncertain factor whose occurrence may result in some loss of satisfaction for some objective.

13.1 Risk Exposure

Risk exposure is a measure of the degree of risk, and is typically defined as:

$$RE = Probability_{occurrence} \cdot Cost_{consequences}$$

The higher the risk exposure, the more it makes sense to reduce risk or mitigate consequences.

13.2 Risk Consequence Table

A risk consequence table has a list of objectives (e.g. creating a product users like, completing the product on time) on one axis and a list of failure modes (e.g. requirements, project management, technical or dependency failures) on the other axis. Each cell in the table is assigned a value from 0 to 1 that estimates the impact of losing a particular objective for a particular failure mode. Rows and columns can be summed up to determine the overall likelihood of losing a particular objective and the criticality of each failure mode. Each objective and failure mode can also be assigned a weighting.

13.3 Risk Management

Risk management attempts to manage the degree to which projects are exposed to risks of quality, delay or failure. Sample risk management techniques include assessment (e.g. identification, analysis, prioritization), avoidance, and control (e.g. management planning, resolution, monitoring).

13.4 Risk Countermeasures Table

A risk countermeasures table is structured similarly to a risk consequence table. However, instead of listing objectives, countermeasures are listed instead. Each cell is assigned a value from 0 to 1 that estimates the reduction of a particular risk using that particular countermeasure.

A set of countermeasures should be selected with join effectiveness and cost in mind.

13.5 Residual Risk

Residual risk is a kind of risk that is expected to remain even after all risk countermeasures are taken. Some residual risk is acceptable for a project; the purpose of countermeasures is to mitigate risk where possible.

14 Specifications

14.1 Terminology

- An **environment** is the part of the outside world that is relevant to some system.
- An **interface** refers to phenomena shared between the system and the environment.
- A **requirement** is a condition or capability to be achieved in the environment.
- A **system** is a proposed solution for achieving requirements. Interactions with the outside environment are performed through an interface.
- A **specification** is a description of a proposed system.

14.2 Requirements vs Specifications

Requirements are statements of desired properties. They describe changes to an environment without referring to a system. These descriptions are often high-level and get elaborated on as development progresses.

Specifications are descriptions of how a proposed system should behave to satisfy requirements. These are typically expressed in terms of a system's interface, and tend to be more concrete and detailed than requirements.

14.3 Deriving Requirements from Specifications

It is often useful for a set of specifications (Spec) to logically imply to set of requirements (Req). However, this is often not possible without additional information or assumptions on how the environment behaves. These domain assumptions (Dom) combined with specifications being able to determine requirements can be considered to be a fundamental law of requirements.

$$Dom, Spec \implies Req$$

14.4 Deriving Specifications

Specifications can also be derived with the help of a set of requirements and domain assumptions. The following is considered for each requirement:

- Determine any specs that for how the system should control the environment to meet this requirement. This may require adding interface phenomena so that the system can perform the necessary interactions with the environment.

- Determine any domain assumptions necessary to link environmentally-controlled phenomena to system-controlled phenomena.

Once these specifications are determined, the following should hold:

$$Dom, Spec \implies Req$$

$$Dom \wedge Spec \text{ is satisfiable}$$

15 Object Constraint Language (OCL)

A business rule is an assertion that defines or constrains some aspect of work. Examples include:

- Rental agreements must be no longer than four weeks.
- Frequent customers receive a 10% discount.

Object Constraint Language (OCL) can be used to express business rules over UML models. OCL allows for relating classes even when they have no direct UML association, and for expressing queries over objects and collections of objects. Types of OCL constraints and expressions include:

- Invariant properties about objects, links, and attribute values
- Initial variable or attribute values
- Pre/post conditions of functions
- Guard conditions and assignment expressions in state machine models

15.1 Constraint Expressions

OCL allows for expressions over:

- Attributes
- Navigations derived from associations and aggregations
- Rolename on far end of association
- Class name on far end of association
- Literal values

15.2 Operations on Collections

While model-defined names and OCL operations are prefaced with dots, OCL operations on collections are prefaced with arrows (->). For example, a sample OCL expression that counts the number of vehicles in a rental agreement is self.RentalAgreement.Vehicle->size()

15.3 Filtering Operations

Filtering operations extract specific elements from an existing collection based on the value of some expression. The select operation returns all values that satisfy an expression, while reject returns all values that falsify that expression. A sample filtering operation that selects all red cars in a company is self.owns->select(color="red")

15.4 Collect Operation

A collect operation iterates over a collection, computes a value for each element, and returns that value in a new collection. A sample use of a collect operation that returns the prices of rental agreements is self.RentalAgreement->collect(price)

15.5 Quantification

15.5.1 Exists

The exists operation is used to assert that at least one element in a collection satisfies some expression. An example of an exists operation that asserts every rental agreement has at least one black car is self.RentalAgreement.Vehicle->exists(color="black")

15.5.2 Forall

The forall operation is used to assert that all elements in a collection satisfy some expression. An example of an forall operation that asserts all rental cars are white is self.owns->forall(color="white")

15.6 OCL Tools

OCL is supported by various types of tools, including:

- Parsers and type checkers
- Evaluators that check OCL expressions against UML class models
- Debuggers that step through an OCL expression and evaluate each subsection
- Code generators that translate OCL expressions into run-time assertions

16 Conflict Resolution

Conflict refers to disagreements among a project's stakeholders.

16.1 Types of Conflicts

16.1.1 Data Conflicts

Data conflicts concern multiple conflicting understandings of an issue. Data conflicts may arise as a result of misunderstanding, or as a result of contradictory requirements. An example would be a requirement that states that a student transcript should include attendance, but some other set of requirements taking up the space where attendance would be placed.

16.1.2 Interest Conflicts

Interest conflicts occur when stakeholders have different goals or interests. An example would be a province demanding that student attendance be tracked for accounting purposes, but students insisting that attendance not be recorded to protect their privacy.

16.1.3 Value Conflicts

Value conflicts occur when stakeholders express different preferences or belief systems (i.e. different perceptions of "good" and "bad"). An example would be a university wanting to stack rank students in courses, but students thinking that distinctions between student performance are often too fine to matter.

16.2 Conflict Resolution Strategies

Various group and individual strategies can be applied to resolving conflicts. When applying these strategies, it is important to consider the relationship stakeholders have with one another, as well as any goals they may share.

17 Behavior Modeling

17.1 UML State Machines

State machines can be used to model behavior based on the history of received inputs. UML state machines model the flow of control between states in a system.

17.1.1 States

States partition a system into distinct modes of operation. An example would be a turnstile having the states of open, closed, and rotating. A single state should have:

- Equivalence of input traces
- Equivalence of future behavior
- Distinct set of input events
- Distinct reactions to input events
- Input processing done internally

States can contain:

- Entry actions, which occur every time the state is entered
- Exit actions, which occur every time the state is exited
- Internal actions

These actions are uninterruptible. On the other hand, activities are interruptible, and are usually expected to take a substantial amount of time.

Each state machine has an initial state. The designator of this initial state may is a pseudo-state, since no time is actually spent in the state. Some state machines may also have a final state, which is a real (non-pseudo) state.

17.1.2 Transitions

Transitions represent observable execution steps. A transition label may consist of the following parts:

- **event(args):** Input event that triggers transition
- **[condition]:** Guard condition; transition does not occur unless this expression is true
- **/action:** Simple, non-interruptible action performed

A transition with no event or condition is enabled when the source state is idle.

17.1.3 Inputs

Inputs represent events that change their environment. Common types of inputs include:

- **Time events**, which occur at a specific time (e.g. today at 8pm) or after a specified passage of time (e.g. 10 seconds after event A)
- **Change events**, which occur when some condition becomes true (e.g. temperature is at least 100 degrees)

17.1.4 Actions

Actions are atomic responses to an event. Sample actions include an output message or a change to some interface phenomena.

17.1.5 Concurrent Regions

Some systems contain several "subsystems" with orthogonal behavior. These systems can be modeled as concurrent state machines, where multiple regions (i.e. substate machines) operating concurrently form a state machine for the entire system. The execution of a concurrent state machine is considered complete once all regions have entered a final state. Alternatively, a termination state may be used to immediately end system execution, even when other regions have not completed.

Concurrent regions can interact with one another in the following ways:

- Reacting to the same input.
- Interacting with the same interface phenomenon.
- One region generating an event that another region handles as input.
- A transition's guard condition containing the state of a different region (not part of the formal UML spec, but can be useful).

17.1.6 Other UML State Diagram Concepts

A few other UML state diagram details include:

- **Hierarchical States:** States with common behaviors or exiting transitions can be grouped together, and can treated externally as a single state.
- **Determinism and priority:** Deal with handling multiple possible transitions (e.g. conditions for two different change events becoming true at one).
- **History:** Pseudo-states known as history states (depicted as H) represent the most recent active direct substate of their containing state. Special types of history states

known as deep history states (depicted as H*) represent the most recent active leaf-substate of their containing state.

17.1.7 Good Style

Some recommendations for state machines include:

- States should model modes of operation, and transitions should model flow of control.
- Fewer transitions are preferable (hierarchy can help with this).
- Concurrent regions should be applied if they can simplify the model.
- Common behavior should not be repeated (hierarchy and entry/exit actions can help with this).

17.1.8 Creating Behavior Models

Below is a general guideline for creating behavior models:

1. Identify input and output events (can be useful to reference domain model).
2. Partition the system into states. State types include:
    - **Activity states**, where a system performs some operation
    - **Idle states**, where a system waits for input
    - **System modes**, where different states are used to distinguish between different event reactions
3. Model the behavior for each input at each state. For each state, the system's response to an event can be any of the following:
    - Event can occur and is handled (transition exists).
    - Event can occur, but results in error (transition exists to report error).
    - Event can occur, but is ignored (no transition).
    - Event cannot occur (no transition).

17.1.9 Validation

Some important validations to carry out on UML state machines include:

- Avoiding inconsistencies (e.g. multiple transitions are not leaving a state under the same conditions)
- Ensuring completeness (e.g. valid input always associated with some transition)

- Walkthrough tests (e.g. state machine flows match use-case scenarios)

17.2 Uses of State Machine Models

State machine models can be used to create mockups, which include rough screen contents and navigation between screens. State machines can also serve as navigation diagrams, where transitions are triggered by user input (e.g. mouse clicks) and the states correspond to "screenshots" of the UI.

17.3 Sequence Diagrams

Sequence diagrams illustrate the interactions between objects over time. While state diagrams describe all allowable scenarios, sequence diagrams only describe a single one. However, sequence diagrams can be useful in validating specific scenarios in state diagrams, and also tend to be more useful for communicating with customers.

18 Temporal Logic

18.1 Overview

Descriptive specification notations can describe constrains on the behavior of a system. An example is OCL, which expresses constraints on object models. Temporal logic describes constraints on variables (e.g. environmental variables, I/O phenomena, events, states) that can vary over time. Temporal constraints can be effective when describing behavior that spans multiple steps.

Examples of temporal logic expressed in natural language include:

- Dialing a phone number always results in either the call being connected or a busy-tone.
- If a car approaches the intersection, the light in its direction will eventually be green.

18.2 States

A sequence of states represents the execution of a system. A state formula is a predicate logic property that is evaluated for a particular execution state. A state formula with a predicate logic expression p being applied to a state s can be expressed as $s \vDash p$.

18.3 Timed Logic

18.3.1 Explicit Time

Each variable is considered to be a function of time. The execution states can be enumerated (i.e. 0, 1, 2, ...), after which each variable can be treated as a function of one of these states (e.g. $x(0) = 1$ means that the variable x is 1 in the initial state). These expressions can also be used to relate variables across different states (e.g. $x(0) >= y(1)$).

Constraints where the precise timing of events is specified as known as real-time constraints.

18.3.2 Implicit Time

Often, only the temporal ordering of events needs to be described, rather than the precise timing of each event. In this case, special connectives (i.e. logic operations) can be used to relate variables at different execution stages. Linear Temporal Logic (LTL) is designed for expressing temporal order of events and variables values while leaving time implicit.

The temporal connectives of LTL include:

- **Henceforth ($\Box f$)**: Asserts f is true in current and all future states.
- **Eventually ($\Diamond f$)**: Asserts f is true in current or some future state.

- **Next ($\bigcirc f$)**: Asserts f is true in next future state.
- **Until (fUg)**: Asserts g is eventually true, and f is true until g is true.
- **Unless (fWg)**: Asserts f is true until g is true, but with no guarantee that g will become true (often used to describe environmental properties out of system control instead of system properties).