Lecture 2

Requirements A condition or capability that must be achieved. Expressed in terms of environmental phenomena

Specification A description of the proposed system. Expressed in terms of interface phenomena.

The **environment** is the relevant subset of the world using posing a problem and the **system** is you solution to the problem. These two interact through the **interface**.

Domain Knowledge We would like to show that specifications imply requirements but this is often impossible so we can make **assumptions** about how the environment behaves, denoted as domain knowledge.

This results in the **fundamental law of requirements** which says that domain knowledge (Dom) and specifications (Spec) imply requirements (Req). We also want to make sure that Dom $\wedge$ Spec is satisfiable.

Lecture 3

Project Types

- Rabbit: small projects, short lifetime, few stakeholders close by
- Horse: medium project, medium lifetime, dozens of stakeholders, geographically distributed
- Elephant: large project, some outsourced development, need for certification

Scope the Problem Find the **purpose** of the project. Figure out some high level **goals** (measurable criteria). Look at the **scope** of the area that will be effected by the installation of the product and the **stakeholders** that are interested in it. Lastly consider the **constrictions** if the system (this includes other systems).

Context Diagram This is a model of the environment in which you need to do your work. This allows you to modularize phenomena into domains. Basically you break up the environmental phenomena that might effect your solution and connect them to the task they effect through an edge.

Stakeholders These are people that have a stake in the success of the project. The **owner/client** is the stakeholder that is paying for the project to be developed. A **customer** is a stakeholder that pays for the product after is is finished. A **user** is a stakeholder that is an expert on the existing system or a competitors products that can give us feedback. When getting feedback make sure that the user base is diverse. The **domain expert** are stakeholders that know the problem very well. The **software engineer** is the stakeholder that is an exert in the technology involved. **Inspectors** are experts on regulations related to the problem. **Market Researchers** are experts in the customers needs and trends. **Lawyers** are experts in the legal requirements. Stakeholders can also be experts on adjacent

systems that might be effected by your system. There can be stakeholders that don't want you project to succeed **negative stakeholders**.

Lecture 4

Use Cases We want to break down the work to be done into pieces to manage complexity.

The Work This includes what you are permitted to change, what you need to understand inorder to decide what to change, and anything that an be affected by the product. Consider all of these things when you when you decide the scope of the project.

We break the work up into vertical slices called **use cases**. These represent some end to end functionality. You have a user or actor interacting with an actor. Time can be one of the actors to show a time trigger used case. Actors can be subclassed using the triangle notation seen in uml. We can label edges with an include statement that shows a sub usecase to be use within this (usually some form of validation).

Lecture 5

Basically domain models are just UML diagrams. Just include class names, attributes and operations.

Lecture 6

Light weight modeling is fairly informal, like mind maps.

Scenarios These are one full exeuction path if a use case listing only the observable actions. Basically pretend to be a user doing something. Number steps as you go, allow logical expressions (like for and if) to have sob paths denoted (.). You can also list an **Alternatives**, sub use case that achieves the same goal through a different path (denoted A). **Exceptions** are unwanted but inevitable deviations (denoted E). A **negative scenario** is a scenario to be prohibited. A **misuse case** is a scenario that captures undesirable inputs from the user environment.

Use Case A use case is a collection of success and failure scenarios initiated by an external actor. These are different than scenarios.

Template:

- Business Event Name:
- Business Use Case Name and Number:
- Trigger:
- Preconditions: Sometimes certain conditions must exist before the use case is valid.
  - Interested Stakeholders:

- Active Stakeholders:
- Normal Case Steps:
- Step 1 . . .
- Step 2 . . .
- Step 3 . . .
- Alternatives:
- Exceptions:
- Outcome:

Activity Diagram These are essential flow charts of actions and interactions from the user. Actions are denoted with circles and the results are branches off diamonds, if there is a fork that is denoted with a long bar.

Process Model This is a decomposition of the work and data dependencies between functions denoted as actors rectangles flow data arrows to process circles and data store bars flowing into the process as well.

Lecture 7

Elicitation This is the process of getting information from your stakeholders.

- required functionality - what the software should do
- quality attributes - desired characteristics of the software
- design constraints - customer-specified limits on solution space
- environmental assumptions - assumed context of the software
- preferences

Documentation Different project types require different amounts of elicitation and documentation.

- Rabbit: requirements considered but probably not documented and mostly just stored in the heads of the stakeholders
- Horse: requirements need to be documented so that the many stakeholders can be on the same page
- Elephant: requirements are critical to get right and very well documented

Artifact based elicitation We want to learn as much as we can by studying artifacts(documentation, systems, and such) before asking stakeholders. **Documents** for system, environment, and domain analysis are used. **Norms** are known problems to improve on (build a better X). A **requirement taxonomy** is a classification of requirements to act as a checklist of details to be elicited.

Model Based elicitation We want to re-express requirements in a different language to raise new questions. **Models** are often used to reveal holes in our understanding, make them simple enough for stakeholders to modify. **Analysis patterns** are like design patterns (in oo programming) for modeling business patterns. **Mockups and prototypes** are good for baiting stakeholders into providing new details.

Stakeholder Based elicitation Acquire detailed information about the system that the stakeholders want and distill requirements from that. Analyzing current users through **questionnaires and interviews** is very helpful. Closed questions gather opinions and open questions gather suggestions. **Ethnographic analysis** is basically just watching a user do stuff and note what they try. Apprenticeship is to have the user teach you how to use the product. **Personas** are fake users to use when you don't have enough (think of stereotypes and how they will use the product).

Creativity Based elicitation We want to invent new crazy requirements to bring about innovation and get an advantage. **Systematic Thinking** is just thinking about the work to be done and not the future system. **Brainstorming** is another from of elicitation. **Creativity Workshops** create a risk free space for creating new ideas beyond the stakeholder requirements.

Lecture 8

Functional Requirement These are used to describe what the product has to do to support and enable the work that needs to be done (basically how we can build the goal).

Atomic Requirements These are derived from scenarios or shit you did previously. These should be written as a single sentence with a single verb (usually shall or must, or could, etc). You should also make atomic requirements for alternatives and exceptions. Once again you can include logical flow with ifs and whiles and such. Atomic requirements are accompanied with rationale explaining why the requirement exists.

Easy Approach to Requirements Syntax (EARS) This is just a template for structuring the syntax for atomic requirements

`<optional precondition><optional trigger> the <system name> shall <system response>`

An **ubiquitous requirements** is an invariant behavior (should always hold true, not precondition or trigger).

Variants:

- event driven: `WHEN <trigger> the <system> shall <response>`
- state driven: `WHILE <in state> the <system> shall <response>`
- options: `WHERE <some feature is included> the <system> shall <reponse>`
- unwanted events: `IF <trigger>, THEN the <system> shall <response>`

User Stories These provide a light-weight approach to managing requirements. Create a short statement of some new functionality but write it from the users perspective. Stories should fit on a index **card**. You should have a **conversation** with the product owner for requirements. Finally you should get **confirmations** to know the acceptance criteria for objectively determining if requirements are met. These are the three C's of user stories.

Lecture 9

Conflict can arrise when chosing requiremetns and such:

- data conflict: conflicting understandings of an issue
- interest conflict: stakeholders have different goals
- value conflict: stakeholders have different preferences

Lecture 10

We have to figure out how to prioritize our requirements (usually into critical, standard, and optimal).

Kano Model Here we ask the user their reaction for if a requirement was included and if it was not included. Using these two values you can sort your requirements.

- Basic: requirements that the customer takes for granted
- Performance: requirements that the customer specifically asked for
- Excitement: requirements that the customer does not request or expect
- Reverse: requirements that the customer hates.
- Indifferent: customer doesnt care
- Questionable: customer gave mixed responses

100-Dollar Test Basically we give stakeholders 100 points to assign to requirements.

Analytic Hierarchy Process This uses the stakeholders' pairwise comparison of requirements and produces a relative ranking. Make a grid and assign factors between requirements (ex. I like requirement A 1/2 as much as B and C times as much as A). Then you normalize so each column sums to 1. Then sum each row and normalize the sums. These values are the priority rateings for each requirement.

Lecture 11

Buisness Rule This is an assertion that defines or constrains some aspect of the work to be done (ex. a customer must be 18+). This is represented in object constraint language which complements the

UML. Basically this is just text on an UML stating constraints. Functions on collections are denoted with - ¿. Functions include, size, reject, select, exists, forall. These rules can be broken, they only state what should be.

Basic types

Booleans

Operation	Notation	Result Type
or	a or b	Boolean
and	a and b	Boolean
exclusive or	a xor b	Boolean
negation	not a	Boolean
equals	a = b	Boolean
not equals	a <> b	Boolean
implies	a implies b	Boolean
if then else	if a then b else b'	Type of b and b'

Strings

Operation	Notation	Result Type
concatenation	s.concat(t)	String
size	s.size	Integer
to lower case	s.toLower	String
to upper case	s.toUpper	String
substring	s.substring(int,int)	String
equals	s=t	Boolean
not equals	s<>t	Boolean

Integers and Reals

Operation	Notation	Result Type
equals	a = b	Boolean
not equals	a <> b	Boolean
less	a < b	Boolean
more	a > b	Boolean
less or equal	a <= b	Boolean
more or equal	a >= b	Boolean
plus	a + b	Integer or Real
minus	a - b	Integer or Real
multiplication	a * b	Integer or Real
division	a / b	Real
modulus	a.mod(b)	Integer
integer division	a.div(b)	Integer
absolute value	a.abs	Integer or Real
maximum value	a.max(b)	Integer or Real
minimum value	a.min(b)	Integer or Real
round	a.round	Integer
floor	a.floor	Integer

Arlow and Neustadt, *UML 2 and the Unified Process*

Expressions on collections

Set of T		
Operation	Notation	Result Type
equals	a = b	Boolean
not equals	a <> b	Boolean
size	a->size()	Integer
sum	a->sum()	Type T
count	a->count(t)	Integer
includes	a->includes(t)	Boolean
excludes	a->excludes(t)	Boolean
includes all	a->includesall(b)	Boolean
excludes all	a->excludesall(b)	Boolean
is empty	a->isEmpty()	Boolean
not empty	a->notEmpty()	Boolean
union	a->union(b)	Set of T
intersection	a->intersection(b)	Set of T
difference	a - b	Set of T
insert	a->including(t)	Set of T
remove	a->excluding(t)	Set of T

a, b : Set of Type T
t : object of Type T

Arlow and Neustadt, *UML 2 and the Unified Process*

Lecture 12

Process Model These represent a single function rather than a full use case.

Lecture 13

Quality Requirements A functional requirement tells you what the software must do, a quality requirement tells you how well it has to do that for it to be acceptable.

Motherhood Requirements These are reliable, user friendly, and maintainable requirements. Basically no one would ever ask for the opposite (no one will ask for unreliable, anti-user, or unmaintainable software).

Fit Criteria This quantifies the exted to which a quality requirement must be met. For example the system should be usuable is a functionality requirement, the users shall judge the system to be usable is a quality requirement, and 75% of users should judge the system as usuable is the fit criteria. We set values for an outstanding case, the target case, and the minimum based on stakeholder feedback.

Monte Carlo techniques These are used to estimate an unknown quality to evaluate fit criteria that we cannot test before releasing (for example uptime per year). An example of doing this is to plot a bunch of points in a rectangle, then measure the area of some strange shape by comparing the ratio of points in the shape to the ration of area of the shape.

Lecture 14

Risk A risk is an uncertain factor whose occurrence may result in some loss of satisfaction corresponding to some objective. We attempt to manage risk since elimination is impossible.

Defect Detection and Prevention Process developed by NASA

1. identify most critical requirements
2. identify potential risks
3. estimate the impact of each risk on those requirements
4. identify possible countermeasures
5. identify the most effective countermeasures

This results in optimized collection of mitigating actions to apply to the project. Store this data in a risk consequence table.

Tall Poles are the most critical risks having the most severe consequence.

The slides on this are shit, look it up later.

Lecture 15

We like state machines since they are easy to draw. There needs to be a designate start and exit state denoted with black dots. Since these states don't actually exit they are called pseudo states. State transitions are labeled with the input event that triggers the transition, some condition to guard the transition, and an action that happens on it. Denoted `event(args)[condition]/action`

Behavior Model First identify the input and output events. Then think of natural partitioning into states (activity state, idle states, and system modes). Finally consider the behavior of the system in each state.

Completeness For each state of the state machine we need to consider all events and make sure there is a transaction for them if needed. If there is a set of similar states we can cluster them and for a hierarchy of states. We also need to prioritize events that might occur at the same time. The history is a pseudo state that designates the hild state of H's parent in a hierarchy. H* is the deep history for the descendant state of h*'s parent