How to not fail SE 463

Jim Zhang

2018 Spring

1 Main course content

1.1 Introduction

Most software projects fail, not because some phony design or code's blessing, but because, they did not make something users want.

Requirements are hard when they are vague, ambiguous, changing, incomplete, infeasible, made with insufficient user input, or plain wrong.

1.1.1 Types of systems

There's an icon corresponding to each one that is apparently plastered on every slide it applies to.

- **Greenfield** vs. **brownfield**
- **Customer-driven** vs. **market-driven**
- **Web/mobile** vs. **enterprise** vs. **infrastructure** vs. **safety critical**
- **In-house** vs. **outsourced**

1.2 The lean business model

Constantly iterate in all stages of development.

1.2.1 Lean canvas model

For the purposes of this course, ignore "cost structure" and "revenue streams". Fill out in order: Outer, middle, NW, SE, SW, NE. Practice filling one of these out.

1.3 Hypothesis testing

Form hypotheses that are **testable** as well as **clearly falsifiable**.

1.3.1 Risks

Customer risk is "who has the pain?". **Product risk** is "what problem are you solving?". **Market risk** is "who is the competition?".

1.3.2 Problem interview

Consists of these stages:

- Welcome
- Collect demographics
- Tell a story
- Problem ranking
- Explore customer's worldview
- Wrap up
- Document results

1.4 Stakeholders

A stakeholder is anyone who will be affected by the project. Types of stakeholders:

- The **champion/owner** is the stakeholder paying for the software to be made.
- The **customer** buys the software after it is made.
- **Users** interact with the software.
- The **domain expert** is very familiar with the problem to be solved.
- The **software engineer** develops the product and makes sure it's feasible.
- **Negative stakeholders** want you to fail.
- **Disfavored users** should have have access to the system.

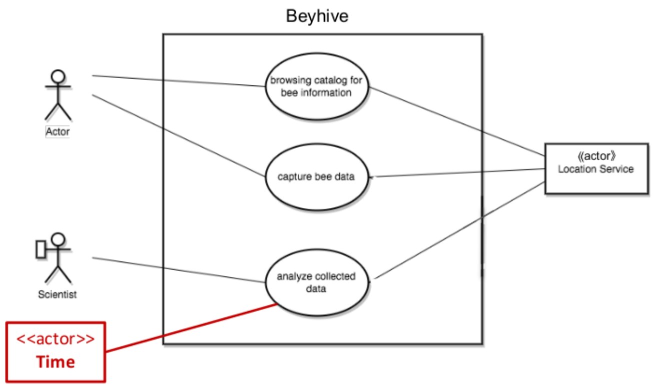
1.4.1 Personas

Fake users with heavily detailed stories and a moniker like "Ken the Keener", used to understand a user class. Can be data-driven.

1.5 User requirements

User requirements are things that users want to be able to do. A **use case** is a vertical slice of end-to-end functionality.

1.5.1 Use case diagram



Primary actors on the left, supporting actors on the right, and all your use cases in circles in a nice big box in the center. Can use actor generalization (UML inheritance). Can also «include» sub-use cases, or «extend» existing use cases, but do it sparingly.

1.5.2 User stories

Short stories written in the format "As a ROLE, I want THING so that BENEFIT". Three C's are Card (that format), Conversation (you should have one), and Confirmations (you should have objective acceptance criteria). This produces user requirements from which you can derive **conditions of satisfaction**, which are functional requirements.

1.5.3 Changing requirements

When user requirements try to change, keep in mind

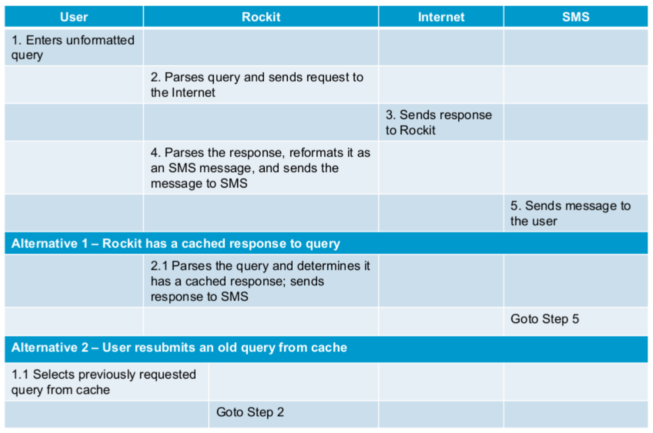
1. Your requirements baseline
2. Your unique value proposition
3. Project scope

And accordingly prioritize user requests.

1.6 Workflow models

A **workflow** is a series of tasks that accomplish some use case.

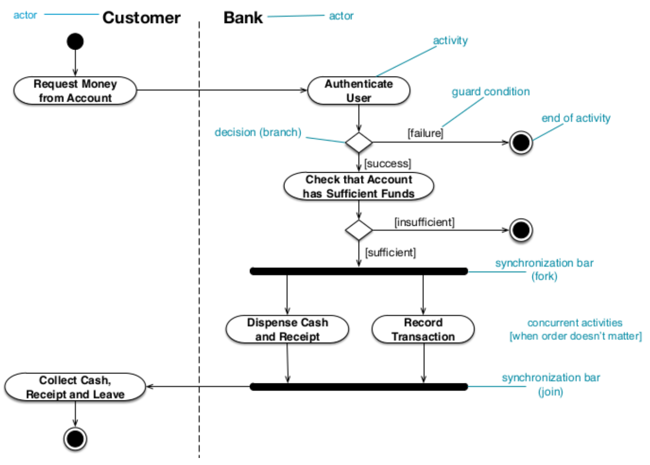
1.6.1 Scenario



User	Rockit	Internet	SMS
1. Enters unformatted query			
	2. Parses query and sends request to the Internet		
		3. Sends response to Rockit	
	4. Parses the response, reformats it as an SMS message, and sends the message to SMS		
			5. Sends message to the user
Alternative 1 – Rockit has a cached response to query			
	2.1 Parses the query and determines it has a cached response; sends response to SMS		
			Goto Step 5
Alternative 2 – User resubmits an old query from cache			
1.1 Selects previously requested query from cache			
	Goto Step 2		

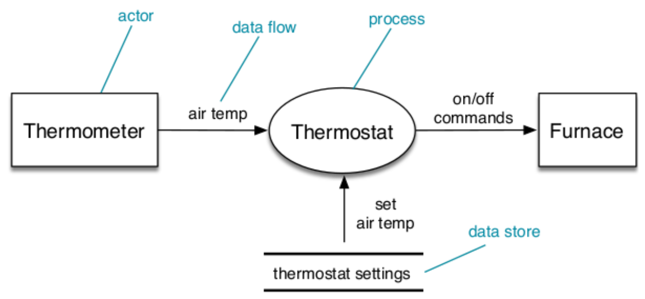
A **scenario** is a full execution path through a use case, listing only observable actions. They are drawn as a table where columns are the actors/systems and rows are time. Actions can have If, For, While, or Go To UC_n but those are signs that you're making the scenario too detailed. **Alternatives** are sub-use cases where the same thing is achieved but the sequence of actions is different. **Exceptions** are sub-use cases that are unwanted deviations. Both start off somewhere in the main use case flow.

1.6.2 Activity diagram


actor
Customer
Bank
actor
activity
Request Money from Account
Authenticate User
guard condition
end of activity
decision (branch)
[failure]
[success]
Check that Account has Sufficient Funds
[insufficient]
[sufficient]
synchronization bar (fork)
Dispense Cash and Receipt
Record Transaction
concurrent activities [when order doesn't matter]
Collect Cash, Receipt and Leave
synchronization bar (join)

An **activity diagram** describes control flow from one activity to another.

1.6.3 Dataflow diagram



A **dataflow diagram** shows functional and data dependencies between functions. Notably, data stores are represented by an over/under bar.

1.7 MVP hypothesis testing

To assess customer risk (who has the pain?) and product risk (how will you fix it?), conduct a **solution interview**:

1. **Welcome**. Describe the rest of the interview.
2. **Collect demographics** — particularly, those numbers relevant to your project.
3. **Tell a story** about the problem you're trying to solve and ask if it resonates with them.
4. **Demo** your solution to each problem.
5. **Wrap up** — ask if they want to see the product, and ask for leads.

1.8 Elicitation

Elicitation is getting requirements. There are four types of it:

- **Artifact-based elicitation**, where you study documentation and the domain without involving stakeholders.
- **Stakeholder-based elicitation**, where you interview, survey, or observe stakeholders. Examples: Interviews, observation, personas, apprenticeship.
- **Model-based elicitation**, where you express existing requirements in a different language. Examples: Modelling, mockups and prototypes, pilot experiments.
- **Creativity-based elicitation**, where you invent undreamt requirements. Examples: Systemic thinking, brainstorming, creativity workshop, constraint relaxation

A **requirements taxonomy** is a tree of requirements.

1.9 Domain models

Domain models are simplified UML class diagrams that model the environment. Classes should have attributes and associations, and multiplicities on all associations.

1.10 Prototyping

Four types of prototypes. We particularly care about the first two.

- **Presentation prototypes** are throwaway proofs of concept used for explaining design features.
- **Exploratory prototypes** are throwaway prototypes that determine problems.
- **Experimental prototypes** (breadboards) explore technical feasibility.
- **Evolutionary prototypes** improve upon an existing system.

Users tend to read pages in an **F-layout**.

1.10.1 SPRINT kickoff

Decide on a Facilitator and a Decider. No devices in the room. Five-day schedule:

- Monday: **Map** existing workflow and find problems with it.
- Tuesday: **Sketch** ideas on sticky notes alone.
- Wednesday: **Decide** on winning and turn them into a storyboard.
- Thursday: **Prototype** a facade using lots of human actors.
- Friday: **Test** your stuff on users (five is enough to reveal patterns).

1.11 Quality requirements

Quality requirements constrain a solution beyond functionality. They include performance, reliability, robustness, adaptability, security, usability, scalability, efficiency/capacity, and accuracy/precision. **Motherhood** requirements are those that are general and always desirable, like "reliable", "user-friendly", and "maintainable".

1.11.1 Fit criteria

A **fit criterion** quantifies the extent to which a quality requirement must be met; e.g. "no more than 5 minutes per year". **Richer fit criteria** additionally have a minimum and outstanding quantity.

1.12 Prioritizing requirements

You can prioritize requirements by attempting to quantify their value and cost and ordering them by value/-cost ratio. Typically you'll want to group the requirements into priority groups like Critical, Standard, and Optional. Here are some techniques.

1.12.1 100-dollar test

Give stakeholders 100 dollars to distribute among the requirements.

1.12.2 Kano model

In order of most to least important:

- Performance: Stuff customer asked for
- Basic: Stuff customer takes for granted
- Excitement: Stuff customer would be pleasantly surprised by
- Indifferent: Stuff customer doesn't care about

1.12.3 Kano survey

Ask customers how they would feel with/without a requirement, giving them the choices "I like it", "I expect it", "I'm neutral", "I can tolerate it", and "I dislike it". Use the Kano Evaluation Table to figure out where in the model the requirement fits.

1.12.4 Categorization plane

Assign weights of -2, -1, 0, 2, and 4 to customer responses. Results in the four quadrants of $[0, 4]^2$ corresponding to Indifferent, Basic, Performance, and Excitement.

1.12.5 Analytic hierarchy process

Get stakeholder to compare all pairs of requirements and assign values of $1, 3, 5, 7, 9$ depending on how much more one is preferred over the other. Put this in a matrix, then do AHP analysis by normalizing columns, taking row sums, and normalizing those — this is your priority vector, and tells you exactly how much goodness is in each requirement.

To check consistency, multiply the original matrix by the priority vector, pointwise divide by the priority vector, average the elements, calculate the consistency index, and compare that with the consistency index of the appropriately sized random matrix.

1.13 Risk management

Let X be a risk, $P(X)$ be the chance of it, and $C(X)$ be the cost of it. Then,

$$\mathit{Risk\ Exposure}(X) = P(X)C(X)$$

1.13.1 Risk consequence table

Risks and their likelihood along the top. Requirements and their weights along the left. Fill in centre with impacts (estimates of the effect of the risk on the requirement). This lets you compute

Loss of Objective for a requirement as the risk of the requirement $\times$ SUMPRODUCT(likelihoods, impacts for requirement) and Risk Criticality of a risk as the likelihood of the risk $\times$ SUMPRODUCT(weights, impacts for risk).

1.13.2 Risk countermeasures table

Risks and their criticalities along the top. Countermeasures along left. Fill in centre with effects (estimate of reduction on risk). This lets you compute the Overall Single Effect of Countermeasure of a countermeasure as SUMPRODUCT(criticalities, effect of countermeasure) and Combined Risk Reduction of a risk as $1 - \prod(1 - \text{effects for risk})$.

1.14 Specifications

Separate the world into the **environment**, the **interface**, and the **system**. A **requirement** is a desired change to the world. The system can only interact with the world through the interface; it cannot touch the environment directly. A **specification** describes how the system should behave in terms of interface phenomena. Ideally, Spec $\models$ Req, but in order to show that we usually need to make assumptions about how the environment behaves (Dom $\subseteq$ Env such that Dom, Spec $\models$ Req).

1.14.1 Domain models for specs and reqs

The system should interact only with other parts of the system and with «interface» phenomena.

1.15 Business rules and OCL

A **business rule** constraints something. We can use OCL, Object Constraint Language, to model these. You just gotta know it.

1.16 Conflict resolution

- **Data conflict**: Different understandies of an issue
- **Interest conflict**: Stakeholders have different goals
- **Value conflict**: Stakeholders have different preferences

1.17 Behavior modeling

Know FSMs, including hierarchy, (deep) history, concurrency, termination, state actions, and activities. Also know **navigation diagrams**, which are just state machines for screens of an app.

1.18 Temporal logic

It's predicate logic with some additions for modelling time. The connectives are:

1. $\Box$ (henceforth)
2. $\diamond$ (eventually)
3. $\circ$ (next state)
4. $\mathcal{U}$ (until)
5. $\mathcal{W}$ (unless)

Practice using them, and practice using them to describe FSMs.

1.19 Software estimation

Estimation error occurs from uncertainty, omitted activities, optimism, bias, and subjectivity. You can estimate in a couple ways.

1.19.1 By analogy

Compare a new project to an old project, and extrapolate from historical data.

1.19.2 Function point analysis

Estimate function points (arbitrary units of code), derive code size from that, and estimate resources from code size. Function points are counted by taking into account external inputs/outputs/queries, internal files, and external interfaces of low, medium, and high complexity, and putting it in a chart. Code size is done by standard SLOC/FP tables. PERT lets you compute expected case in a terrible way:

$$Expected = \frac{Best + 4 \times MostLikely + Worst}{6}$$

1.19.3 Confidence intervals

Once you have best, expected, and worst case effort estimates, you can estimate the standard deviation by taking $(worst - best)/P$ for each use case, where $P = 6$ if you're 99.7% confident in your estimate range and less if you're realistic. Then compute variance for each use case, add 'em up, and square root to get a standard deviation for your entire project. Then it's just a matter of assuming your project finish time is normally distributed and just consult a standard deviation table.

2 Guest lectures

2.1 Shyam Sheth

Users that are less committed to their current solution are easier to get.

2.2 Robyn Lutz

Requirements engineering was used to decide to use a molecular watchdog timer to monitor for events in a DNA nanosystem. In fact, requirements engineering is useful for lots of safety-critical applications.

2.3 James Corr

Made his consultancy a lot more successful by giving clients a big requirements doc and spending some time collecting requirements.