Requirements Analysis
aka
Requirements Engineering

Defining the WHAT
Requirements

- Specify functionality
  - model objects and resources
  - model behavior
- Specify data interfaces
  - type, quantity, frequency, reliability
  - providers, receivers
  - operational profile (expected scenarios)
  - stress profile (worst case scenarios)

Requirements

- Specify interfaces
  - Control interfaces (APIs)
  - User interfaces - functionality and style
  - Hardware interfaces
- Specify error handling
- Identify potential modifications
Requirements

- Identify necessary constraints
  - performance, security, reliability
- Identify areas of risk
  - alternatives to be explored
- Specify validation plans
- Specify documentation to be provided

Analysis Principles

- Document reason for specific requirements
- Prioritize requirements
  - High, medium, low
- Ignore implementation details
  - Need to know feasible solutions can be developed
  - If feasibility is a concern, then propose alternatives to be explored
- Be prepared to change
Perspective and Early Binding of Constraints

Connect like letters without crossing lines or leaving box.

Early Focus on Constraints

- A-A line seems to be only solution
  - But, is it really?
  - Need to examine domain and constraints more?
- What transforms or shifts would yield easier problem?
Focus Change

These choices still leave a path between A's

---

The Requirements Process

- A **requirement** is an expression of desired behavior
- A requirement deals with
  - objects or entities
  - the state they can be in
  - functions that are performed to change states or object characteristics
- Requirements focus on the customer needs, not on the solution or implementation
  - designate **what behavior**, without saying how that behavior will be realized

---

The Requirements Process
Process for Capturing Requirements

- Performed by the req. analyst or system analyst
- The final outcome is a Software Requirements Specification (SRS) document

Requirements Elicitation

- Customers do not always understand what their needs and problems are
- It is important to discuss the requirements with everyone who has a stake in the system
- Come up with agreement on what the requirements are
  - If we can not agree on what the requirements are, then the project is doomed to fail
Requirements Elicitation

Stakeholders

- **Clients**: pay for the software to be developed
- **Customers**: buy the software after it is developed
- **Users**: use the system
- **Domain experts**: familiar with the problem that the software must automate
- **Market Researchers**: conduct surveys to determine future trends and potential customers
- **Lawyers or auditors**: familiar with government, safety, or legal requirements
- **Software engineers** or other technology experts

Means of Eliciting Requirements

- Interviewing stakeholders
- Reviewing available documentations
- Observing the current system (if one exists)
- Apprenticing with users to learn about user's task in more details
- Interviewing user or stakeholders in groups
- Using domain specific strategies, such as Joint Application Design, or PIECES
- Brainstorming with current and potential users
### 4.3 Types of Requirements

- **Functional requirement**: describes required behavior in terms of required activities
- **Quality requirement** or **nonfunctional requirement**: describes some quality characteristic that the software must possess
- **Design constraint**: a design decision such as choice of platform or interface components
- **Process constraint**: a restriction on the techniques or resources that can be used to build the system

---

### Sidebar: Making Requirements Testable

- Fit criteria form objective standards for judging whether a proposed solution satisfies the requirements
  - It is easy to set fit criteria for quantifiable requirements
  - It is hard for subjective quality requirements
- Three ways to help make requirements testable
  - Specify a quantitative description for each adverb and adjective
  - Replace pronouns with specific names of entities
  - Make sure that every noun is defined in exactly one place in the requirements documents
Different stakeholder has different set of requirements
  – potential conflicting ideas

Need to prioritize requirements

Prioritization might separate requirements into three categories
  – essential: absolutely must be met
  – desirable: highly desirable but not necessary
  – optional: possible but could be eliminated

---

Requirements definition: a complete listing of everything the customer wants to achieve
  – Describing the entities in the environment where the system will be installed

Requirements specification: restates the requirements as a specification of how the proposed system shall behave
Types of Requirements
Two Kinds of Requirements Documents (continued)

- Requirements defined anywhere within the environment's domain, including the system's interface
- Specification restricted only to the intersection between environment and system domain

Characteristics of Requirements

- Correct
- Consistent
- Unambiguous
- Complete
- Feasible
- Relevant
- Testable
- Traceable
Non-Functional Requirements (NFRs)

- Definitions
  - Quality criteria; metrics
  - Example NFRs
- Product-oriented Software Qualities
  - Making quality criteria specific
  - Catalogues of NFRs
  - Example: Reliability
- Process-oriented Software Qualities
  - Softgoal analysis for design tradeoffs

What are Non-functional Requirements?

- Functional vs. Non-Functional
  - Functional requirements describe what the system should do
    - functions that can be captured in use cases
    - behaviours that can be analyzed by drawing sequence diagrams, statecharts, etc.
    - ... and probably trace to individual chunks of a program
  - Non-functional requirements are global constraints on a software system
    - e.g. development costs, operational costs, performance, reliability, maintainability, portability, robustness etc.
    - Often known as software qualities, or just the “ilities”
    - Usually cannot be implemented in a single module of a program
NFRs

• The challenge of NFRs
  – Hard to model
  – Usually stated informally, and so are:
    • often contradictory,
    • difficult to enforce during development
    • difficult to evaluate for the customer prior to delivery
  – Hard to make them measurable requirements
    • We’d like to state them in a way that we can measure how well they’ve been met

Example NFRs

• Interface requirements
  – how will the new system interface with its environment?
    • User interfaces and “user-friendliness”
    • Interfaces with other systems

• Performance requirements
  – time/space bounds
    • workloads, response time, throughput and available storage space
    • e.g. “the system must handle 1,000 transactions per second”
  – reliability
    • the availability of components
    • integrity of information maintained and supplied to the system
    • e.g. “system must have less than 1hr downtime per three months”
  – security
    • e.g. permissible information flows, or who can do what
  – survivability
    • e.g. system will need to survive fire, natural catastrophes, etc

• Operating requirements
  – physical constraints (size, weight),
  – personnel availability & skill level
  – accessibility for maintenance
  – environmental conditions
  – etc

• Lifecycle requirements
  – “Future-proofing”
    • Maintainability
    • Enhanceability
    • Portability
    • expected market or product lifespan
  – limits on development
    • e.g. development time limitations,
    • resource availability
    • methodological standards
    • etc.

• Economic requirements
  – e.g. restrictions on immediate and/or long-term costs.
### Reviewing a requirements document

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is it ambiguous?</td>
<td>Carefully define terms and use these terms</td>
</tr>
<tr>
<td>Is it consistent?</td>
<td></td>
</tr>
<tr>
<td>Is it complete?</td>
<td>Vague requirements</td>
</tr>
<tr>
<td>Is it verifiable?</td>
<td></td>
</tr>
<tr>
<td>Is it realistic?</td>
<td></td>
</tr>
<tr>
<td>Does it plan for change?</td>
<td></td>
</tr>
<tr>
<td>Does it not overly constrain the problem?</td>
<td>Have alternatives been considered and explored?</td>
</tr>
<tr>
<td>Is it clearly presented?</td>
<td>Precise, concise, clear</td>
</tr>
<tr>
<td></td>
<td>diagram complex objects and behaviors</td>
</tr>
<tr>
<td>Is it what the customer wants?</td>
<td></td>
</tr>
</tbody>
</table>

### Why is requirements analysis difficult?

- **Communication:** misunderstandings between the customer and the analyst
  - Analyst doesn’t understand the domain
  - Customer doesn’t understand alternatives and trade-offs
- **Problem complexity**
  - Inconsistencies in problem statement
  - Omissions/incompleteness in problem statement
  - Inappropriate detail in problem statement
Escalator System Requirements

Two Signs on Escalator:

- Shoes Must Be Worn
- Dogs Must Be Carried

**Consistent Conclusions:**

- You must have on shoes, and you must be carrying a dog.
- If you have a dog, you have to carry it, so you have to wear all the shoes you are carrying.
- If you don’t have a dog, you don’t need to carry it, so you don’t have to wear shoes unless you are carrying some.

Why is requirements analysis difficult?

- Need to accommodate change
  - Hard to predict change
  - Hard to plan for change
  - Hard to foresee the impact of change
First Law of Software Engineering

“No matter where you are in the system lifecycle, the system will change, and the desire to change it will persist throughout the lifecycle.”

Reasons for changing requirements

- Poor communication
- Inaccurate requirements analysis
- Failure to consider alternatives
- New users
- New customer goals
- New customer environment
- New technology
- Competition
- Software is seen as malleable

Changes made after requirements are approved increase cost and schedule
## Requirements Products

- **Specification document**
  - Agreement between customer and developer
  - Validation criteria for software
  - Problem statement in domain language
    - *external* behavior
    - Constraints on system
- **Preliminary users manual**
- **Prototype** *(do not deliver the prototype!)*
  - If user interaction is important
  - If resources are available
- **Review by customer and developer**
  - Iteration is almost always required

## Modeling Notations

- It is important to have standard notations for Modeling, documenting, and communicating decisions
- Modeling helps us to understand requirements thoroughly
  - Holes in the models reveal unknown or ambiguous behavior
  - Multiple, conflicting outputs to the same input reveal inconsistencies in the requirements
DETOUR TO UML MODELING

Approaching a Problem

Where do we start?
How do we proceed?
Where Do We Start?

- Start with the requirements
  - Capture your goals and possible constraints
  - Environmental assumptions
- Use-case analysis to better understand your requirements
  - Find actors and a first round of use-cases
- Start conceptual modeling
  - Conceptual class diagram
  - Interaction diagrams to clarify use-cases
  - Activity diagrams to understand major processing

How Do We Continue?

- Refine use-cases
  - Possibly some “real” use-cases
    - Using interface mockups
- Refine (or restructure) your class diagram
  - Based on your hardware architecture
    - For instance, client server
- Refine and expand your dynamic model
  - Until you are comfortable that you understand the required behavior
- Identify most operations and attributes
How Do We Wrap Up?

- Refine the class diagram based on platform and language properties
  - Navigability, public, private, etc
  - Class libraries
- Identify most operations
  - Not the trivial get, set, etc.
- Write a contract for each operation
- Define a collection of invariants for each class
- Implement

Putting It Together

- **Principles**
  - Rigor and Formality
  - Separation of Concerns
  - Modularity
  - Abstraction
  - Anticipation of Change
  - Generality
  - Incrementality
- **Notion of Process**
Waterfall Approach

- Gather Requirements
- Generate Design
- Write Code
- Requirements Documents
- Design Documents
- Implementation
- Test

Domain language
Models, technical language

Frequent Industrial Approach

- Gather Requirements
- Write Code
- Implementation
- Test
- Requirements Documents
- Design Documents
- Generate Design
- Prototypes

Domain language
Models, technical language
Process Overview

- Inception
- Elaboration
- Construction
  - Many iterations
- Transition

Overview: Steps to Follow

- Map out environment as-is
- Map out environment as required
- Decide on systems boundaries / goals
- List actions with types
- Define terms
- Construct model
- Challenge model
- Modify as required
Analysis: Steps to follow

• Obtain a problem statement
• Develop use cases (depict scenarios of use)
• Build an object model and data dictionary
• Develop a dynamic model
  – state and sequence diagrams
• Verify, iterate, and refine the models
• Produce analysis document

Use Cases

• High-level overview of system use
• Identify scenarios of usage
• Identify actors of the system:
  – External entities (e.g., users, systems, etc.)
• Identify system activities
• Draw connections between actors and activities
• Identify dependencies between activities (i.e., extends, uses)
Analysis: Domain Model

• Organization of system into classes connected by associations
  – Shows the static structure
  – Organizes and decomposes system into more manageable subsystems
  – Describes real world classes and relationships

Analysis: Domain Model

• Object model precedes the dynamic model because
  – static structure is usually better defined
  – less dependent on details
  – more stable as the system evolves
Analysis: Domain Model

- Information comes from
  - The problem statement and use cases
  - Expert knowledge of the application domain
    - Interviews with customer
    - Consultation with experts
    - Outside research performed by analyst
  - General knowledge of the real world

Client View of Domain

- Clients can’t be expected to have rigorous or formal view of domain
- Hence, can’t be expected to be completely aware of domain-problem relationship
- Some knowledge is *explicit*
  - Easier to get at
- Some knowledge is *implicit* ("everybody knows...")
  - *Many* constraints are implicit
  - Hard to get at
Object Model: Steps to follow

- Identify classes and associations
  - nouns and verbs in a problem description
- Create data dictionary entry for each
- Add attributes
- Combine and organize classes using inheritance

Analysis: Dynamic model

- Shows the time dependent behavior of the system and the objects in it
- Expressed in terms of
  - states of objects and activities in states
  - events and actions
- State diagram summarizes permissible event sequences for objects with important dynamic behavior
Dynamic Model: Steps to follow

- Use cases provide scenarios of typical interaction sequences
- Identify events between objects (Sequence Diagram)
- Prepare an event trace for each scenario
- Build state diagrams
- **Match events between objects to verify consistency**

Analysis: Iteration

- Analysis model will require multiple passes to complete
- Look for inconsistencies and revise
- Look for omissions/vagueness and revise
- Validate the final model with the customer
  - Pose scenarios from various perspectives
    - Look for consistency in responses
Object Model: main embedded system objects or classes

- Controller object
  - might be made up of several controllers
  - is the brains of the system.
  - Takes input from the sensors and gives instructions to the actuators.

- Sensor object
  - environmental objects that gives information to controller.
  - Can be passive (thermometer) or active (button).

- Actuator object
  - Mechanical device to realize an effect

Meeting Purposes

- Disseminate information (including stating a problem)
- Gathering opinions
- Confirming consensus
- Social requirements
  - team building
  - approval
## Meeting Requirements

- **Agenda**
- **Leader**
- **Action list**
  - With assignments so we know who is doing what.
  - Timelines so we know when it’s to get done.
- **Summary**
  - Something happened or there would not have been a meeting. Record it briefly.

## Project Issue List

- Every issue goes on the list
  - Date and brief description
- Make assignment to get it resolved
- Put resolution on list.
  - “Close” issue.
- 1st version usually generated on 1st read of problem statement.
  - And then, back to the customer...
Interviewing

- Have a list of things you want to know.
- Listen.
- Listen.
- Ask open-ended questions.
- Don’t express (show, say) opinions on answers. Just record, and think.
- Listen.
- Ask questions more than one way.

Close-ended questions

Q: When a vehicle cuts in front of the car, you have to slow down quickly and not hit it, right?
A: Yes.

You learned absolutely nothing.
Open-ended questions

Q: Tell me what happens when a car cuts in front of you.

A: Well, if the lead car is too close, the driver has to intervene or else a crash results. I guess we need a warning light in this case. If the car is moving faster, you don't have to do anything. He's pulling away. I guess the only time brakes are used is when the closing speed is too high for the distance and yet within the capabilities of the system to slow down. But I guess if a collision is imminent, we should max out the braking.

Ah ha! new requirement!

And a clarification

Now, we learned something...

Responses

Q: Tell me what should happen if a car cuts in front of our car too close to avoid a collision?

A: I guess since there is nothing the system can do. Turn off the controller and hope the driver brakes in time.

Much better

Not good

Q: We have quite a bit of braking power in the system. What would happen if we used it here?

A: Well, I guess it could avoid a collision and at least get the car slowed down but the attorneys tell me we don't want the system active when a collision occurs.

Q: What? Are you nuts? We should at least try to stop. Shouldn't we?

A: Perhaps...

You are done at this point, and still unresolved.

Ah ha! Non-technical constraint