CS189A Capstone
Fall 2023

Lecture 4: Requirements Specification

PRD: Product Requirements Document

- The official statement of what is required of the system developers
- Includes a specification of both user and system requirements

- Defines WHAT the system should do, not HOW it should do it
  - Design comes later

- Agile and extreme SWE processes express requirements as
  - Use cases – how a system will act
  - Or as scenarios called user stories (describe result/benefit of it)

  - **Both** document how the system responds from an *external* perspective (when viewed from the **outside**) – like a black box…

So - we are only interested in describing externally visible behavior

PRDv1: Your **Living** Requirements Document: A Shared Google Doc (due next week)

- Authors, Team, Project Title
- Intro – including problem, innovation, science, core technical advance (2-3 pages)
  - Define project specifics, team goals/objectives, background, and assumptions
- System architecture overview
  - High level diagram (1 page)
  - User interaction and design (1 page)
- Requirements (functional and non-functional)
  - User stories or use cases (links): 10 for PRDv1 prioritized
  - Prototyping code, tests, metrics (5+ user stories): github commits/issues
- System models: contexts, sequences, behavioral/UML, state
- Appendices
  - Technologies employed

On-going Process

- Evolving (aka “living”) requirements document
  - Identify/learn (and teach each other) the technologies required
  - Write user stories in particular; update the requirements as you go:
    - Prioritize stories and mark **mandatory, important, or desirable**
    - Assign time estimates to stories; improve your estimation ability over time
    - Specify **acceptance test** for each story – should be in code
- Concurrently as part of Sprint
  - Break down stories into tasks (begin design/prototyping process)
    - Prioritize tasks
    - Assign timings to tasks
    - Specify what (code) test(s) are to be used as evidence of task completion/acceptance
    - Each member/developer chooses task, implements, and tests task
    - Another member does code review/test and accepts the pull request
      - Test is the one specified above (Acceptance)
    - When a Story is complete, some member performs story test/acceptance

Requirements Engineering

- Process of establishing
  the services that the customer requires from a system and
  the constraints under which it operates and is developed
  - May range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification
  - Precisely stated, unambiguous
- User requirements
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers
- System requirements
  - A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor

Software Requirements

- Brooks in “No Silver Bullet” paper says:

  The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements ... No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later.

- Developers of the early Ballistic Missile Defense System observed [Alford, IEEE TSE, 1977]

  In nearly every software project that fails to meet performance and cost goals, requirements inadequacies play a major and expensive role in project failure

- In mission-critical defense systems US government identifies requirements as a major problem source in two thirds of the systems examined [US General Accounting Office, 1992]
- Other studies on projects in aerospace industry and NASA also found requirements to be a critical software development problem

Requirements errors are costly to fix late!

Stage	Relative Repair Cost
Requirements	1-2
Design	5
Coding	10
Unit Test	20
System Test	50
Maintenance	200

Relative cost to repair a requirements error in different stages of software life-cycle (cost increases exponentially)

Requirements Validation Techniques

- Requirements reviews
  - Systematic manual analysis of the requirements.
  - Review/commit changes to repository as part of workflow
    - Multiple team members OK it before committing
    - All team members get notification when its updated
- Prototyping
  - Using an executable model of the system to check requirements.
- Test-case generation
  - Developing tests for requirements to check testability.
  - Your test cases / acceptance tests should be github commits

Software Requirements

- Software Requirements Specification:
  - Specification of a particular software product in a specific environment

- Basic goal of the software requirements specification is to specify what the software must do. To achieve this goal:
  - Understand precisely what is required of the software
  - Communicate the understanding of what is required to all the parties involved in the development
  - Provide a means for controlling the production to ensure that the final system satisfies the requirements (including managing the effects of changes)

What are included in software requirements

- **Functionality:** What is the software supposed to do?
  - Example: *The software product shall sort a set of integers in ascending order. The software product shall write the sorted set of integers to an output file in the ASCII format. In the output file each integer shall be separated by a blank space.*
- **External Interfaces:** How does the software interact with people, the system's hardware (there may be a hardware component within the system), other hardware and other software?
  - Example: *The software product shall read the set of integers from an ASCII file.*
    - We also have to specify the format of the input file and how the name of the file will be given.

What shall be in Software Requirements?

- **Performance:** What is the speed, availability, response time, recovery time of various software functions, etc.?
  - Example: *For the input files with less than 1000 integers the software product shall produce the output file within 2 seconds.*
- **Design constraints imposed on an implementation:** Are there any required standards, implementation language restrictions, resource limits, operating environment(s) etc.?
  - Example: *The software product shall run on PCs that run Linux operating system.*
    - We should also specify which version of Linux, what type of PC (constraints on processor, memory etc.)

Classification of Requirements

- Requirements can be classified as:
  - **Functional requirements:** Requirements defining the behavior of the system, fundamental process or transformation that the software performs on inputs to produce outputs
  - **Nonfunctional requirements:** Requirements and constraints on external interfaces, performance, dependability, maintainability, reusability, security, etc.
  - **Domain requirements:** Requirements that come from the application domain of the system and reflect characteristics of the domain (can be functional or nonfunctional)

Functional vs Non-functional Requirements

- Functional requirements (user + system requirements)
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations
  - May also state what the system should not do

- Domain requirements
  - Constraints on the system from the domain of operation
    - Operating environment (e.g. underwater, temp range, environmental conditions to be tolerated)

- Non-functional requirements
  - Constraints on services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  - Often apply to the system as a whole rather than individual features or services

Non-functional requirements

- These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular IDE, programming language or development method
- Non-functional requirements may be more critical than functional requirements and effect overall architecture (e.g. minimize communications). If not met, system may be useless
- A single non-functional requirement, such as a security requirement, may generate a number of related *functional* requirements that define system services that are required.
  - It may also generate requirements that restrict existing requirements

Metrics for Specifying Non-functional Requirements

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Who uses requirements?

- What are the uses of software requirement specification?
  - For customers it is a specification of the product that will be delivered, ***a contract***
  - For managers it can be used as a basis ***for scheduling and measuring progress***
  - For the software designers it provides a specification of ***what to design***
  - For coders it defines the range of ***acceptable implementations*** and the ***outputs that must be produced***
  - For quality assurance personnel it is used for ***validation, test planning, and verification***

Essential difficulties in Software Requirements

- **Comprehension:** People do not know exactly what they want. They may not have a precise and detailed understanding of what the output must be for every possible input, how long each operation should take, etc.
- **Communication:** Software requirements are difficult to communicate effectively. The fact that requirements specification has multiple purposes and audiences makes this problem even more severe
- **Control:** It is difficult to predict the cost of implementing different requirements. Frequent changes to requirements make it difficult to develop stable specifications
- **Inseparable concerns:** Requirements must simultaneously address concerns of developers and customers. There may be conflicting constraints which may require trade-offs, compromises.

Phases

- We can think of developing a software requirements specification document as a two-phase process
  - **Problem analysis:** Goal is to understand the purpose of the software, who will use it, what is the required functionality, constraints on acceptable solutions, possible trade-offs between conflicting constraints
  - **Requirements specification:** Goal is to create the Software Requirements Specification (SRS) document describing exactly what is to be built. SRS captures the results of the problem analysis

Problem Analysis

- Basic issues:
  - How to effectively elicit a complete set of requirements from the customer or other sources?
  - How to decompose the problem into intellectually manageable pieces?
  - How to organize the information so it can be understood?
  - How to communicate about the problem with all the parties involved?
  - How to resolve conflicting needs?
  - How to know when to stop?

Eliciting Requirements

To elicit the requirements

- Interviews with the customer
- Use questionnaires if there are multiple users
- Investigate the environment the product will be used
  - investigate the customer's business
- Scenarios: Walk through different scenarios of how the product will be used by the customer
  - understandable to the customer
  - can uncover additional requirements
- Rapid Prototyping: After an initial requirements analysis, build a prototype. Focus on aspects of the software that will be visible to the user such as input/output formats
  - Prototype should focus on the key functionality (for example, can ignore error checks)
  - Prototype is reviewed by the customer and/or user to refine the requirements for the software to be developed

Characteristics of a good Software Requirements

- **Correct:** Every requirement stated in SRS should be one that the software shall meet. Correctness can be checked by customer or a higher level specification (system specification)
- **Unambiguous:** Every stated requirement in SRS should have only one interpretation
  - Natural languages are inherently ambiguous, they should be used carefully
  - Use of formal languages can help, however they may be hard for the customer to understand
- **Complete:**
  - All significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interfaces should be included
  - Responses to all realizable classes of input data and situations should be included (responses to both valid and invalid input)

Characteristics of a good Software Requirements

- **Consistent:** No subset of specified requirements should conflict. Possible conflicts:
  - There may be logical or temporal conflicts between two specified actions
  - Different part of SRS may use different terms to refer to the same object
- **Verifiable:** A requirement is verifiable if there exists some cost-effective process with which a person or machine can check that the software product meets the requirement
  - Your claims should be ***measurable***
  - Avoid subjective phrases such as "*works well*" which are not possible to measure/verify
  - A verifiable requirement: *Output of the program shall be produced within 20 seconds of event X 60% of the time; and shall be produced within 30 seconds of event X 100 % of the time*

Characteristics of a good Software Requirements

- **Modifiable:** The style and structure of SRS should make it possible to change it easily, completely and consistently
  - No redundancy
  - Express each requirement separately (not intermixed)
- **Traceable:** SRS should facilitate referencing of each requirement in future development or enhancement documentation
  - Good indexing
    - **Do not forget the page numbers!**

IEEE Recommendation for Software Requirements Specification

https://ieeexplore.ieee.org/document/720574

We will not use this format in this course

For large and safety critical projects it might be necessary to use a more detailed requirements specification like the IEEE recommendation

IEEE format is more suitable for the waterfall model, in this class we are using agile software development

IEEE Recommendation for Software Requirements Specification (SRS)

Table of Contents

1. Introduction
   - 1.1 Purpose
     - Purpose of the SRS
     - Intended audience of the SRS
   - 1.2 Scope
     - List software products that will be produced
     - Summarize what software products will do
     - Describe the application of the software being specified, including relevant benefits, objectives and goals
   - 1.3 Definitions, acronyms, abbreviations
     - Definition of all terms, acronyms, abbreviations required to properly interpret SRS
   - 1.4 References
     - Provide a complete list of referenced documents
   - 1.5 Overview
     - Describe what is in the reminder of the document
     - Explain how SRS is organized

2. Overall description

2.1 Product perspective

- Identify the interface between the proposed software and existing systems, including a diagram of major system components.
- A block diagram showing major components of the larger system, interconnections, and external interfaces can be helpful.

2.2 Product functions

- Provide a summary of the major functions that the software will perform
- The functions should be organized in a way that makes the list of functions understandable to the customer or to anyone else reading the document
- Diagrams can be used to explain different functions and their relationships

2.3 User characteristics

- General characteristics of the intended user of the product, level of expertise/training required to use the product

2.4 Constraints

- List all the constraints that will limit the developers options, interfaces to other applications, programming language requirements, hardware limitations, etc.

2.5 Assumptions and dependencies

- List the factors that affect the requirements in the SRS (assumptions on which operating system is available etc.)

3. Specific requirements (these are the detailed requirements)
   - This section of the SRS should contain the software requirements to a level of detail sufficient to enable designers to design a system to satisfy those requirements, and testers to test that the system satisfies those requirements

   3.1 External interface requirements
   - This section should specify various interfaces in detail: system interfaces, user interfaces, hardware interfaces, software interfaces, communications interfaces, etc.
   - A detailed description of all inputs and outputs from the software system should be given:
     - Should include: source of input and destination of output; valid range, accuracy and/or tolerance; units of measure; timing; screen formats/organization; window formats/organization; data formats; command formats, etc.
   - Should complement but not repeat the information given in section 2

3.1.1 User interfaces

- Screen formats, page or window layouts, error messages, etc. Some sample screen dumps can be used here to explain the interface

3.1.2 Hardware interfaces

- Interface between hardware and software product, which devices are supported

3.1.3 Software interfaces

- Specify use of other software products and interfaces with other application systems

3.1.4 Communication interfaces

- Interfaces to communications such as local network protocols, etc.

3.2 Functional requirements

- Functional requirements should define all the fundamental actions that the system must take place in the software in accepting and processing the inputs and in processing and generating the outputs
- Should include: validity checks on input; exact sequence of operations; responses to abnormal situations; relationship of outputs to inputs
- It can be organized in various ways, such as with respect to user classes, features, stimulus or a combination of those.
- Use-case diagrams, scenarios, activity diagrams can be used here

3.3 Performance requirements
- Speed, availability, response time, recovery time of various software functions, etc.
- Performance requirements should be specified in measurable terms. For example: *"95% of the transactions shall be processed in less than 1 second."* rather than *"An operator shall not have to wait for the transaction to complete*
- There can be a separate section identifying the capacity constraints (for example amount of data that will be handled)

3.4 Design constraints
- Required standards, implementation language restrictions, resource limits, operating environment(s) etc.

3.5 Software system attributes
- Attributes such as security, portability, reliability

3.6 Domain requirements
- Explain the application domain and constraints on the application domain

4 Appendices
- Any other important material

Software Specification Problem

- In different phases of the software process we need ways to specify the deliverable for that phase
  - Need to specify the requirements
    - Which is what you are doing in product requirements document (PRD)
  - Need to specify the design
    - We need to document and communicate the design
  - Need to specify the implementation
    - Comments
    - Assertions

Specification Languages

- Main issue: When you write code you write it in a programming language
  - How do you *write* the requirements?
  - How do you *write* the design?
- Specification languages
  - Used to specify the requirements or the design
  - As we have seen parts of software requirements are necessarily in English (customer has to understand). To bring some structure to the requirements specification you can use semi-formal techniques such as use-case diagrams. Depending on the application you maybe able to use formal techniques too
  - For design you can use UML class diagrams, sequence diagrams, state diagrams, activity diagrams
  - Some specification languages (such as UML class diagrams are supported with code generation tools)

Specification

- Specifications can be
  - Informal
    - No formal syntax or semantics
      - for example in English
    - Informal specifications can be ambiguous and imprecise
  - Semiformal
    - Syntax is precise but does not have formal semantics
    - UML (Universal Modeling Language) class diagrams, sequence diagrams
  - Formal
    - Both syntax and semantics are formal
    - Z, Statecharts, SDL (Specification and Design Language), Message Sequence Charts (MSC), Petri nets, CSP, SCR, RSML

Ambiguities in Informal Specifications

- "The input can be typed or selected from the menu"
  - The input can be typed or selected from the menu or both
  - The input can be typed or selected from the menu but not both

- "The number of songs selected should be less than 10"
  - Is it strictly less than?
  - Or, is it less than or equal?

- "The user has to select the options A and B or C"
  - Is it "(A and B) or C"
  - Or, is it "A and (B or C)"

A success story for formal specifications: RSML and TCAS

- Requirements State Machine Language (RSML)
  - A formal specification language based on hierarchical state machines (statecharts)
- The developers of RSML applied it to the specification of Traffic Collision Avoidance System (TCAS) to demonstrate benefits of using RSML [Leveson et al. 1994]
  - TCAS: the specification of a software system which is required on all aircraft in USA carrying more than 30 passengers During the specification of TCAS in RSML ambiguities were discovered in the original English specification of TCAS
  - Eventually FAA decided to use the RSML versions of the TCAS specification

Another Example Formal Specification

- Formal specifications avoid ambiguity
  - However, they could be hard to understand
  - And it is not easy to write formal specifications

- Let's try to specify a sorting procedure formally (mathematically)
- I will just use basic Math concepts: functions, integers, arithmetic
  - Input: I : An array of size n of integers
    - How do we formally specify what an *array* is?
    - $I : \mathbf{Z} \rightarrow \mathbf{Z}$ (a function from integers to integers)
    - $I : 1 \ldots n \rightarrow \mathbf{Z}$
    - $n \geq 1$

Example: Sorting

– Output: $O : 1 \ldots n \rightarrow \mathbf{Z}$

- $\forall\ i\ ,\ O(i) \leq O(i{+}1)$
- $\forall\ i\ ,\ 1 \leq i \leq n \Rightarrow O(i) \leq O(i{+}1)$
- $\forall\ i\ ,\ 1 \leq i < n \Rightarrow O(i) \leq O(i{+}1)$
- $(\forall\ i\ ,\ 1 \leq i < n \Rightarrow O(i) \leq O(i{+}1))$

$\wedge\ (\forall\ i\ ,\ 1 \leq i \leq n \Rightarrow (\exists\ j\ ,\ 1 \leq j \leq n\ \wedge\ O(i) = I(j)))$

- $(\forall\ i\ ,\ 1 \leq i < n \Rightarrow O(i) \leq O(i{+}1))$

$\wedge\ (\forall\ i\ ,\ 1 \leq i \leq n \Rightarrow (\exists\ j\ ,\ 1 \leq j \leq n\ \wedge\ O(i) = I(j)))$

$\wedge\ (\forall\ i\ ,\ 1 \leq i \leq n \Rightarrow (\exists\ j\ ,\ 1 \leq j \leq n\ \wedge\ I(i) = O(j)))$

- $(\forall\ i\ ,\ 1 \leq i < n \Rightarrow O(i) \leq O(i{+}1))$

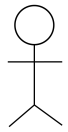
$\wedge\ (\exists\ f\ : 1 \ldots n \rightarrow 1 \ldots n\ ,$

$(\forall\ i\ ,j\ ,\ (1 \leq i \leq n\ \wedge\ 1 \leq j \leq n\ \wedge\ i \neq j) \Rightarrow f(i) \neq f(j))$

$\wedge\ (\forall\ i,\ 1 \leq i \leq n \Rightarrow O(i) = I(f(i))))$

Use cases

- Use cases document the behavior of the system from the users' point of view.
  - By user we mean anything external to the system
- An **actor** is a role played by an outside entity that interacts directly with the system
  - An actor can be a human, or a machine or program
  - Actors are shown as stick figures in use case diagrams



Customer

Use cases

- A **use case** describes the possible sequences of interactions among the system and one or more actors in response to some initial stimulus by one of the actors
  - Each way of using the system is called a use case
  - A use case is not a single scenario but rather a description of *a set of scenarios*
  - For example: *Creating an account*
  - Individual use cases are shown as named ovals in use case diagrams

- A use case involves a sequence of interactions between the initiator and the system, possibly involving other actors.
- In a use case, the system is considered as a black-box. We are only interested in externally visible behavior

Use cases

- To define a use case, group all transactions that are similar in nature
- A typical use case might include a main case, with alternatives taken in various combinations and including all possible exceptions that can arise in handling them
  - Use case for a bank: *Performing a Transaction at the Counter*
    - Subcases could include *Making Deposits*, *Making Withdrawals*, etc., together with exceptions such as *Overdrawn* or *Account Closed*
  - *Apply for a Loan* could be a separate use case since it is likely to involve very different interactions
- Description of a use case should include events exchanged between objects and the operations performed by the system that are visible to actors

Defining use cases

1. Identify the boundary of the application, identify the objects outside the boundary that interact with the system
2. Classify the objects by the roles they play, each role defines an actor
3. Each fundamentally different way an actor uses the system is a use case
4. Make up some specific scenarios for each use case (plug in parameters if necessary)
5. Determine the interaction sequences: identify the event that initiates the use case, determine if there are preconditions that must be true before the use case can begin, determine the conclusion
6. Write a prose description of the use case
7. Consider all the exceptions that can occur and how they affect the use case
8. Look for common fragments among different use cases and factor them out into base cases and additions

Documenting use cases: Online Shopping

Use case: Place Order **Actors:** Costumer

Precondition: A valid user has logged into the system

Flow of Events:

1. The use case begins when the customer selects Place Order
2. The customer enters his or her name and address
3. If the customer enters only the zip code, the system supplies the city and state
4. The customer enters product codes for products to be ordered
5. For each product code entered
   a) the system supplies a product description and price
   b) the system adds the price of the item to the total

   end loop
6. The customer enters credit card payment information
7. The customer selects Submit
8. The system verifies the information [Exception: Information Incorrect], saves the order as pending, and forwards payment information to the accounting system.
9. When payment is confirmed [Exception: Payment not Confirmed], the order is marked confirmed, an order ID is returned to the customer, and the use case terminates

Exceptions:

Payment not Confirmed: the system will prompt the customer to correct payment information or cancel. If the customer chooses to correct the information, go back to step 6 in the Basic Path. If the customer chooses to cancel, the use case terminates.

Information Incorrect: If any information is incorrect, the system prompts the customer to correct it.

Postcondition: If the order was not canceled, it is saved in the system and marked confirmed

Documenting use cases: Online Shopping

Flow of Events:

Basic Path:

1. The use case begins when the customer selects Place Order
2. The customer enters his or her name and address
3. While the customer enters product codes
   a) the system supplies a product description and price
   b) the system adds the price of the item to the total

end loop

6. The customer enters credit card payment information
7. The customer selects Submit
8. The system verifies the information, saves the order as pending, and forwards payment information to the accounting system. If any information is incorrect, the system prompts the customer to correct it.
9. When payment is confirmed [Exception: payment not confirmed], the order is marked confirmed, an order ID is returned to the customer, and the use case terminates

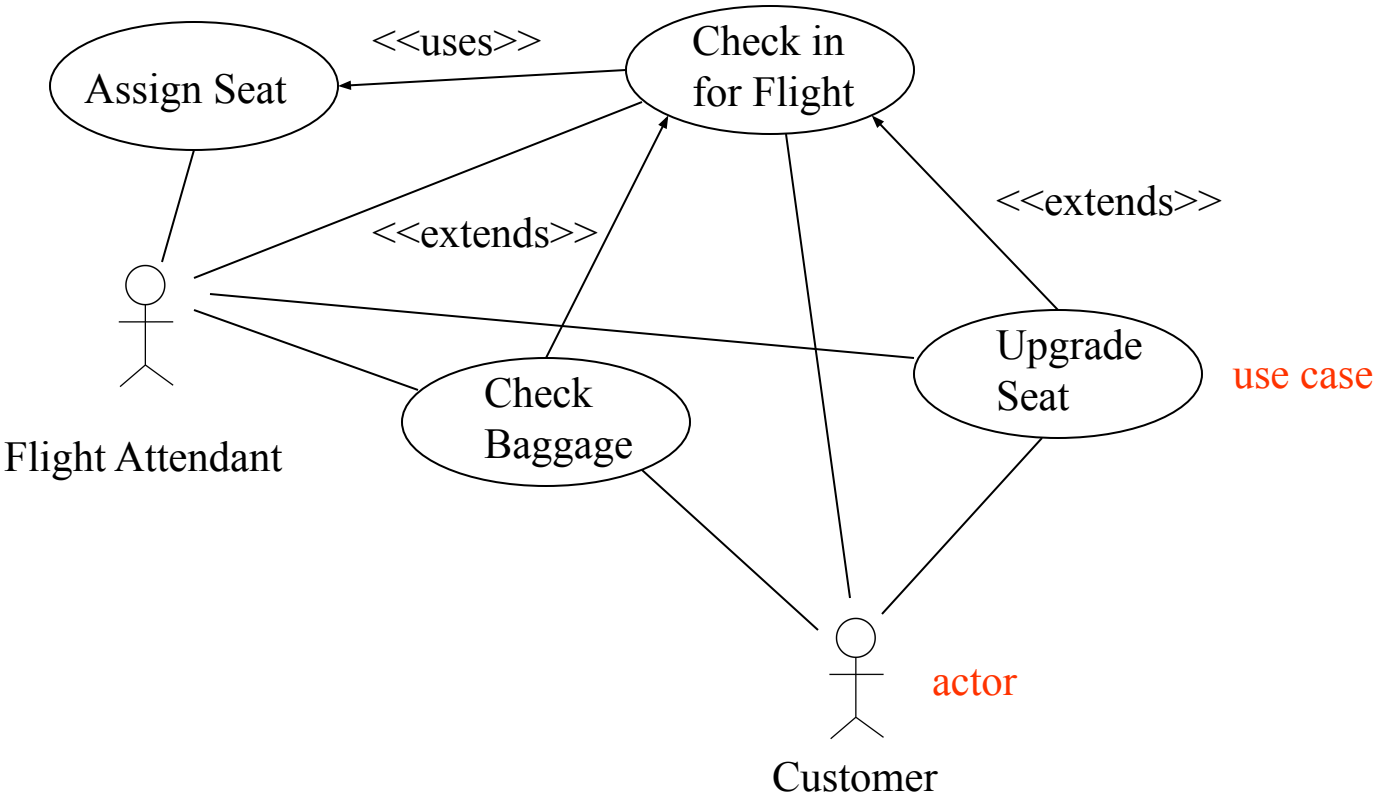
Alternative Paths:

- At any time before step 7, the customer can select Cancel. The order is not saved and the use case ends
- In step 2, if the customer enters only the zip code, the system supplies the city and state
- In step 6, if any information is incorrect, the system prompts the customer to correct the information
- In step 7, if payment is not confirmed, the system prompts the customer to correct payment information or cancel. If the customer chooses to correct the information, go back to step 4 in the Basic Path. If the customer chooses to cancel, the use case ends.

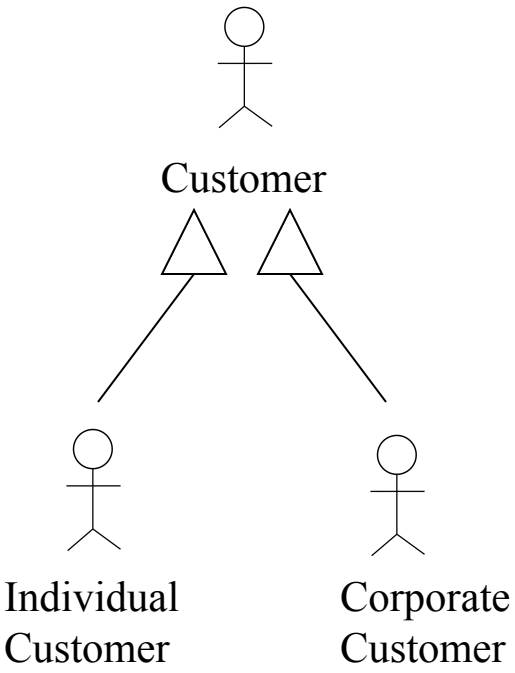
Combining use cases

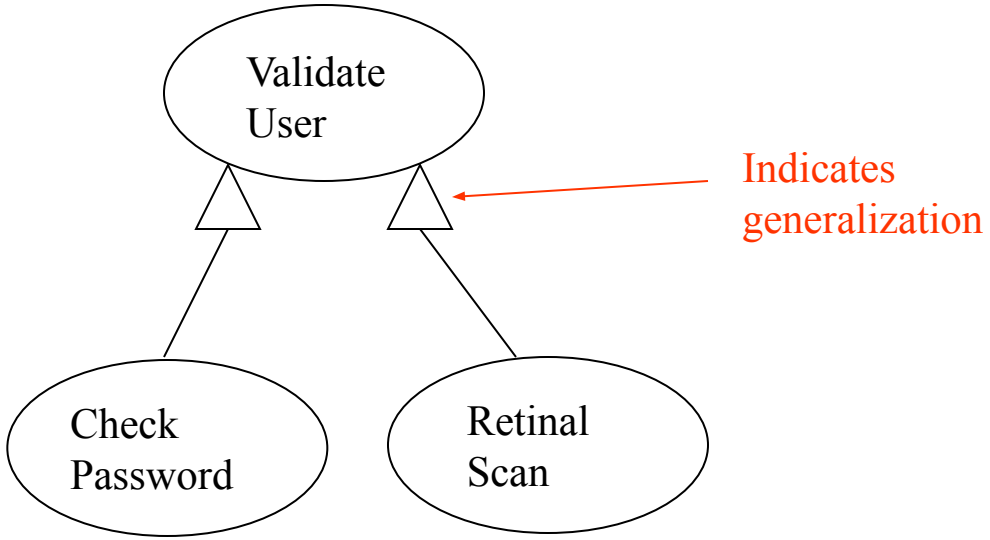
- A use case **extends** another use case when it embeds new behavior into a complete base case
  - *Check Baggage* **extends** the base case *Check in for Flight*
  - You do not have to check baggage to check in for flight.
- A use case **uses** another use case when it embeds a subsequence as a necessary part of a larger case (In some texts this relationship is called **includes** instead of **uses**)
  - **uses** relationship permits the same behavior to be embedded in many otherwise unrelated use cases
  - For example C*heck in for Flight* use case **uses** *Assign Seat* use case
- The difference is
  - In the **extends** the extended use case is a valid use case by itself
  - In the **uses** the use case which is using the other use case is not complete without it

Combining use cases

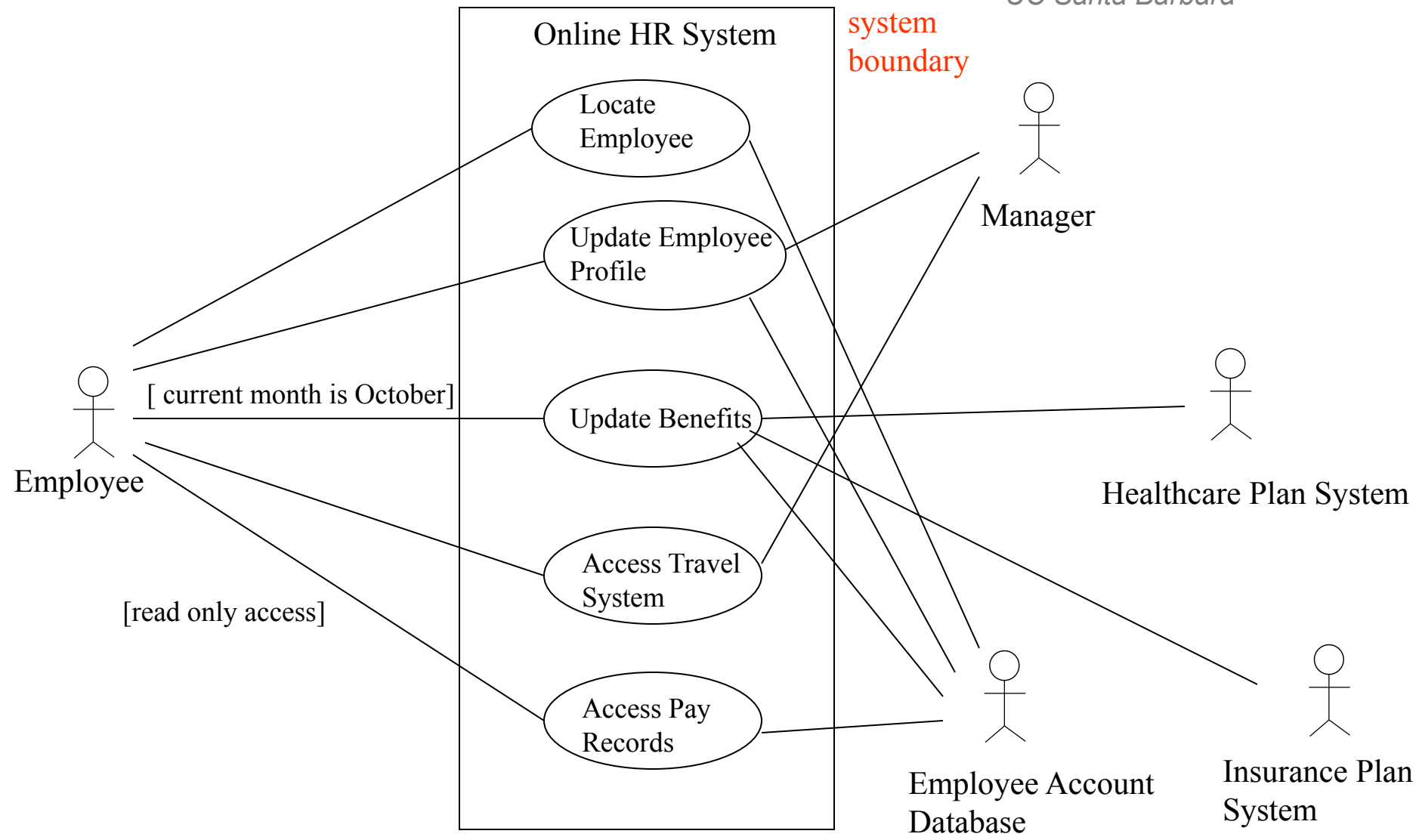


Generalization in use case diagrams





Online Human Resources (HR) System



Online HR System

Use case: Update Benefits

Actors: Employee, Employee Account Database, Healthcare Plan System, Insurance Plan System

Precondition: Employee has logged on to the system and selected "update benefits" option

Flow of Events:

Basic Path:

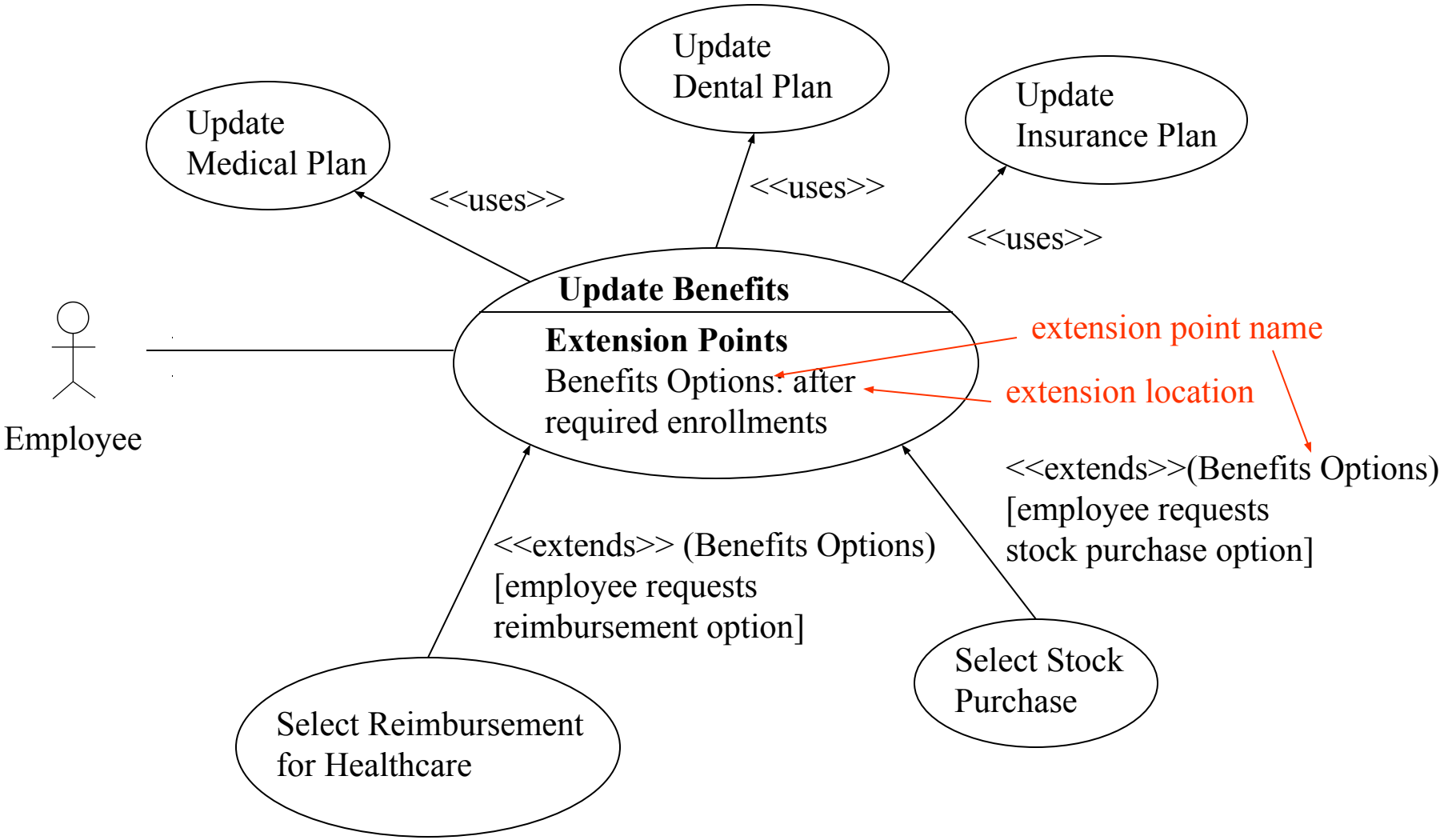
1. System retrieves employee account from Employee Account Database
2. System asks employee to select medical plan type; **uses** Update Medical Plan
3. System asks employee to select dental plan type; **uses** Update Dental Plan

...

Alternative Paths:

If health plan is not available in the Employee's area the employee is informed and asked to select another plan

Online HR System



References

- “Software requirements: A tutorial,” Stuart R. Faulk
- IEEE Recommended Practice for Software Requirements Specifications
- “Getting started: Using use cases to capture requirements”, James Rumbaugh, 1994
- “Using UML”, Perdita Stevens, Rob Pooley, 2000

User Stories

- Similar to Use Cases but not the same
  - User stories are centered on the result and the benefit of the thing you're describing, whereas use cases are more granular, and describe how your system will act.

- Use cases: actors – scope – goals – steps – success
  - Details of most important requirements worked out ahead of time to ensure that everyone is on the same page
  - Useful for groups of similar stories and describing overall system
    - Use cases decompose stories into actions in the system

- User stories: scope of a feature + acceptance criteria
  - Each feature is captured as a story; stories easily prioritized
  - A story is a place holder for discussion and planning poker in a sprint

  See **recommended reading** links for examples and suggestions

Writing Good User Stories

- It is typically difficult to get started writing good user stories

- Here are 4 steps to make it easier

1. As a [role], I can [feature] so that [reason]
2. You can use index cards and a sharpie or software tools like Jira, Trello, Pivotal tracker
3. Make it testable with acceptance criteria or demo plan
4. Connect the dots

From: http://codesqueeze.com/the-easy-way-to-writing-good-user-stories/

As a [role], I can [feature] so that [reason]

- Role – a person;
  feature – something your project does;
  reason – a solution to a problem the person has
  - This is a pattern that is commonly used for stories

  As a account owner, I can check my balance online so that I can access my daily balance 24 hours a day.

- Variations
  - As a [role], I want [feature] because [reason]
  - As a [role], I can [feature]
  - As a [role], I can [feature] so that [reason]

Developing user stories

- There is software out there to help you with this
  - Jira, Trello, Pivotal tracker
- You can also use index cards and a sharpie
  - It also enables you to doodle/draw the outline of the user interface
- Keep them short and sweet and unambiguous
  - Goal is to aid communication, not overly detailed or long-winded
- If it doesn't fit, break up the story into sub-stories

Make it testable with acceptance test or demo

- If they are short and sweet and without detail, how do we know when they are "done"?

Story: As a [role], I can [feature] so that [reason]

- Include an acceptance test (what to demo when done):

  Scenario 1: Title
  Given [context]
  And [some more context]…
  When [event]
  Then [outcome]
  And [another outcome]…

Example

Scenario 1: Account balance is negative
Given the account's balance is below 0
And there is not a scheduled direct deposit that day
When the account owner attempts to withdraw money
Then the bank will deny it
And send the account owner a nasty letter.

- Tests should be simple
  - You should be able to ***code*** them in a few lines of code
  - If they don't, break up the story into two

References

Recommended reading:

http://www.boost.co.nz/blog/2012/01/use-cases-or-user-stories/

http://codesqueeze.com/the-easy-way-to-writing-good-user-stories/

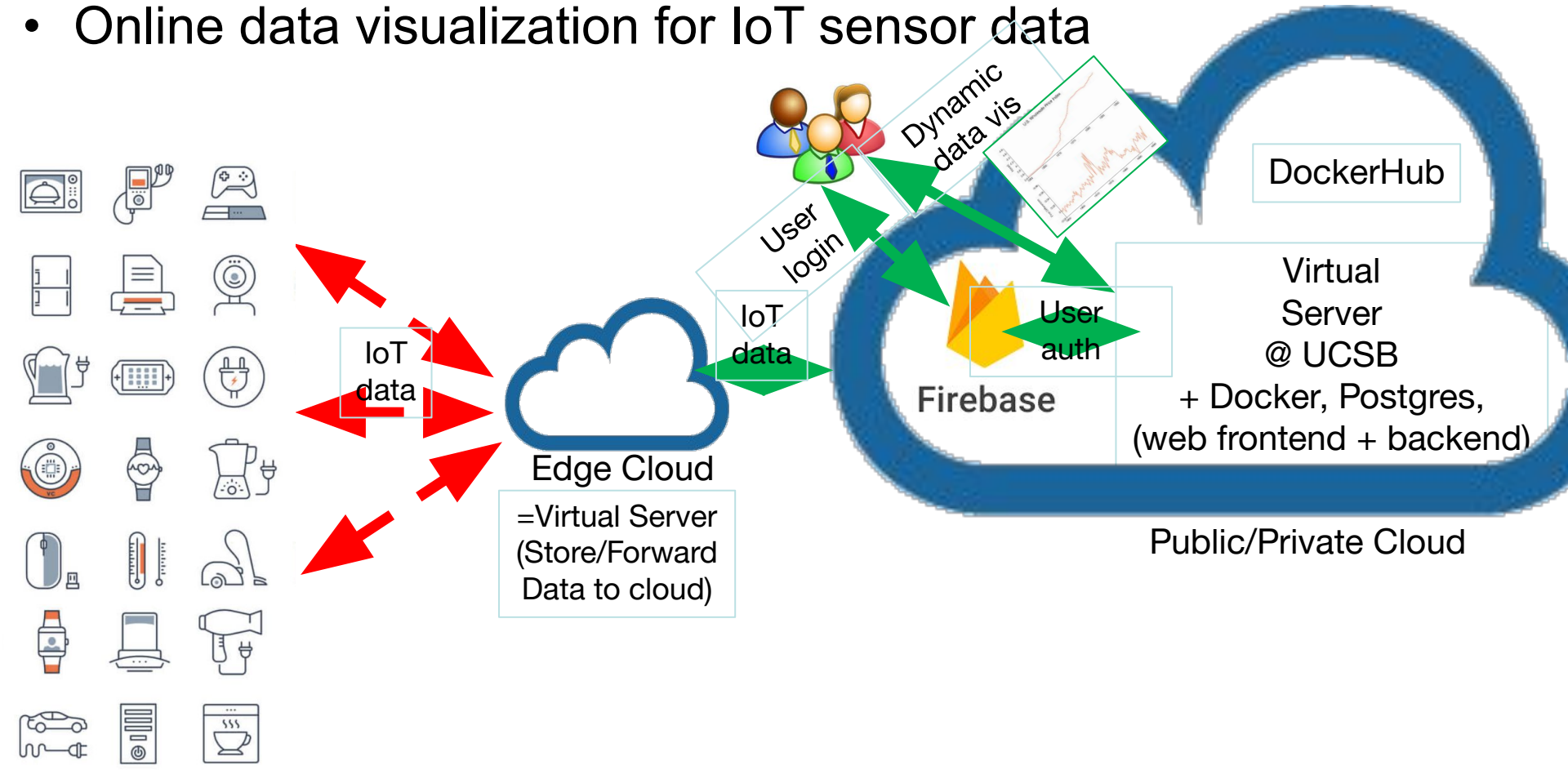
PRDv1: Your **Living** Requirements Document: A Shared Google Doc (due in 1 week)

- Authors, Team, Project Title
- Intro – including problem, innovation, science, core technical advance (2-3 pages)
  - Define project specifics, team goals/objectives, background, and assumptions
- System architecture overview
  - High level diagram (1 page)
  - User interaction and design (1+ page)
- Requirements (functional and non-functional)
  - User stories or use cases (links) : 10 for PRDv1 prioritized
  - Prototyping code, tests, metrics (5+ user stories): github commits/issues
- System models: contexts, sequences, behavioral/UML, state
- Appendices
  - Technologies employed

A Hypothetical Example Project

Example Project: IoT Sensor Data Viewer

- Online data visualization for IoT sensor data



Example Project: IoT Sensor Data Viewer

- As a [role], I can [feature] so that [reason]

  Title: Given [context], when [event], Then [outcome]

1. As a user, I can login, so that I can use the system
   - Login: Given a user name and password, when saved on a web page form, then the user is logged in and can access viewer services (test = a test page is loaded that is only accessible to logged in users)

Example Project: IoT Sensor Data Viewer

- As a [role], I can [feature] so that [reason]

  Title: Given [context], when [event], Then [outcome]

1. As a user, I can login, so that I can use the system
   - Login: Given a user name and password, when saved on a web page form, then the user is logged in and can access viewer services (test = a test page is loaded that is only accessible to logged in users)
2. As a sensor, I can upload my data to the viewer over the Internet, so that it is persisted there
   - Sensor Upload: Given a sensor connected to the Internet, when a script invokes an upload API, then a window of data from the sensor is uploaded to the viewer and the viewer saves the data (test = see the data locally, run the script, see the on the viewer)
     - Note that there is no notion of “how” to persist: database, file, etc
     - That can be included or pushed off until later…

Example Project: IoT Sensor Data Viewer

- As a [role], I can [feature] so that [reason]

  Title: Given [context], when [event], Then [outcome]

3. As a developer, I can save my environment to DockerHub, so that I download and use it on different machines
   - DockerHub: Given a container, when exited and pushed to DockerHub, then the container can be downloaded and run on a different machine (test = upload, download, and run of container)

Example Project: IoT Sensor Data Viewer

- As a [role], I can [feature] so that [reason]

  Title: Given [context], when [event], Then [outcome]

4. As an API request, I can store incoming data to a database, so that it is persisted in a structured format

  - Valid DB Storage Request: Given a API request over the Internet, when a valid request occurs, a window of data is stored in a DB (test1=make an valid request with data: view data separately in DB, view schema)
  - Invalid DB Storage Request: Given a API request over the Internet, when an invalid request occurs, an error is logged/returned (test2=make an invalid request: no change to DB, throw error message))

Example Project: IoT Sensor Data Viewer

- As a [role], I can [feature] so that [reason]

  Title: Given [context], when [event], Then [outcome]

5. As an API, I can handle multiple, concurrent requests at once, so that throughput is maximized given available server resources
   - Throughput: Given an API, when multiple requests come in, multiple threads handle the requests, #threads = VALxCPUs (test = measure throughput for different numbers of threads and CPU resources)

Example Project: IoT Sensor Data Viewer

- As a [role], I can [feature] so that [reason]

  Title: Given [context], when [event], Then [outcome]

5. As an API, I can handle multiple, concurrent requests at once, so that throughput is maximized given available server resources
   - Throughput: Given an API, when multiple requests come in, multiple threads handle the requests, #threads = VALxCPUs (test = measure throughput for different numbers of threads and CPU resources)
- Others:
  - As a user, I can access API operation, so that … //define/implement different API functions
  - As a user, I can access multiple web pages for the service, so that… //define/implement UI and/or front end
  - As a user, I can view sensor data over the Internet, so that… //add Chart.js for fake, static data
  - As a user, I can view dynamic sensor data over the Internet, so that… //add Chart.js + GraphQL for fake, dynamic data
  - Next: connect vis to DB, add API ops, add edge server support (revisit earlier features), handle errors if/when sensors go down or send bad data, support multiple vis options, add data analysis/ML algorithms on data for extracting insights from data, …

CS189A Schedule

Week 1 (Oct. 2)	project pitch meeting, team formation, project selection	
Week 2 (Oct. 9)	contact mentors, select team lead and scribe	sprint 1 starts
Week 3 (Oct. 16)	vision statement due on Oct. 17	
Week 4 (Oct. 23)		sprint 1 ends, sprint 2 starts
Week 5 (Oct. 30)	Requirements and Design PRD version 1 due Oct. 31	
Week 6 (Nov. 6)		sprint 2 ends, sprint 3 starts
Week 7 (Nov. 13)	Requirements and Design PRD version 2 due Nov. 14	
Week 8 (Nov. 20)		sprint 3 ends, sprint 4 starts
Week 9 (Nov. 27)		
Week 10 (Dec. 4)	Final presentations and demos	sprint 4 ends

Sprint planning

- Sprint planning (using Trello): Backlog, on deck, in progress, done
  - Break up into tasks **with durations** (hours, part/days, points)
  - Assign 2+ members to each (implementer and tester/reviewer)
    - Fill 10 days according to durations for each member
    - Order tasks by priority (top = highest): top total 10 days * 5 members
  - Any new tasks identified put onto bottom of backlog for next time
- Setup burndown using google worksheet (shared w/ all)
- **Daily standup/scrum** (scribe records in google doc)

Sprint 2 Plan

- Retrospective: go around team, state 1 good, then 1 bad
  - Discuss bad; identify 2 things to improve/fix for next sprint (scribe this)
- Sprint 2 plan
  - 10 use cases or user stories (w/ one to construct/complete PRDv1 – *Due in a week*) **w/ acceptance test**
    - Add these to trello story board (add link to PRD)
  - PRDv2 tasks (see following slides)
  - Break into tasks
    - Estimate/discuss timings (1/2 – 1 day each; or points)
    - Demo/prototyping plan
    - Total up to have timing per story
  - Choose tasks until tasks are covered
  - Setup up burndown graph and update each scrum

- **For Instructor and/or TA**
  - Demos from last sprint
  - Improvement plan
  - Top story for Sprint 2
    - Plan for demo
  - Sprint 2 Tasks (5x10days)
    - 2+ members each
    - Plan for prototyping/acc. testing
  - Burndown (Sprint 1 and 2)
  - Sprint2 backlog
    - Remaining tasks
    - Pursue these if you get done early
  - Product backlog (Story board)
    - Prioritized stories/use cases
    - Tasks in trello linked/colo coded

PRDv1: Your **Living** Requirements Document: A Shared Google Doc (due next week)

- Authors, Team, Project Title
- Intro – including problem, innovation, science, core technical advance (2-3 pages)
  - Define project specifics, team goals/objectives, background, and assumptions
- System architecture overview
  - High level diagram (1 page)
  - User interaction and design (1+ page)
- Requirements (functional and non-functional)
  - User stories or use cases (links): 10 for PRDv1 prioritized
  - Prototyping code, tests, metrics (5+ use cases or user stories): github commits/issues
- System models: contexts, sequences, behaviorial/UML, state
- Appendices
  - Technologies employed

PRDv2: Your **Living** Requirements Document: A Shared Google Doc (due 1 month after PRDv1)

- Authors, Team, Project Title
- Intro: problem, innovation, science, core technical advance
  - Define project specifics, team goals/objectives, background, and assumptions
- System architecture overview
  - High level diagram (1 page)
  - User interaction and design (1 page)
- Requirements (functional and non-functional)
  - User stories or use cases (links): 20+ for PRDv2 prioritized
  - Prototyping code, tests, metrics (10+ user stories): github commits/issues
- System models (1+ pages)
  - Contexts, interactions, structural, behavioral (UML)
  - Use cases, sequencing, event response, system state, classes/objects
- Appendices - Technologies employed

PRD Tasks for Sprint 2

- PRDv1: 10 stories/ucases **prioritized** w/ acceptance tests/testable postconditions
  - **5+ with implementation started or completed** w/ tests
  - Give links to github commits for 5+ in PRDv1 writeup next to story
- PRDv1: architecture/system diagram
- PRDv1: 2-3 page in depth writeup: problem, innovation, science, core technical advance; project specifics, team goals/objectives, background, & assumptions
- PRDv2: more detailed system diagram + detailed design
- PRDv2: 10 additional stories/ucase (20 total), 5+ additional implementations/tests (10 total)
- PRDv2: 3+ sequence diagrams, 3+ UI interaction/sequence diagrams + mockups