Requirements analysis

- **What do stakeholders want from the system?**
  - What should it do?
  - What should it look like? Sound like? Be like?

- **Analysis starts with a project description**
  - Usually written (or otherwise expressed) by major stakeholder
    - a.k.a. “Client” – might be a customer, another department in the company, management, professor, …
    - Or project team writes it for an anticipated market

- **Results in a series of RA artifacts**: 2 purposes
  - Shows the client what they will be getting
  - Used to kick-off and guide later development activities
RA starts in UP Phase I: Inception

- Purpose is to explore project *feasibility*
- Target length: only about a week
- Identify most use cases and actors
  - And write 10-20% of use cases in detail
  - Used to make rough estimate of costs
- Most important requirements artifacts: *vision, use cases*
Project descriptions

- **Client’s view:** system is basically a “black box”
- Probably vague, repetitive, confused, …
  - But remember: client thinks it “says it all”
- Often has too many details, or misguided focus
  - e.g., implementation details – too limiting at this stage
  - e.g., too many “ilities” – distract from the purpose
- May contain contradictions or impossible parts
  - Often just “wish lists” without clear goals
- So, always expect to re-express as requirements
Doing requirements analysis

- Basically: detailing the requirements
  - But still in language that the user understands
    - i.e., all artifacts continue to treat the system as a black box – focus on what goes in and what comes out
  - For CS 48: write a vision (beefed up) and use cases

- Study much more than the project description
  - Interview users, managers, sponsors, experts, …
  - Learn about current practices, existing systems, business rules (general and specific), memo trails, …
  - But no need to become a domain expert
    - Could take years! A “knowledgeable layperson” is sufficient.
Vision 1: problem statement

- Should answer two fundamental questions:
  - What problem(s) will the system solve?
  - How is the system expected to solve the problem(s)?
- Stakeholders must approve it before proceeding
  - Becomes basis for contract (if real client)
    - Bounds the client’s expectations
    - Establishes scope of work
  - Note: might also state what the system will NOT do
- Narrows the focus of the project team
  - Limits the range of system goals
Vision 2: system goals

- Essentially, the system’s *major responsibilities*
  - Should solve problems for stakeholders, inc. users
- High-level goals apply to overall system
  - What will the system do, and/or be like?
  - Typically span use cases of a complex system
  - Each stakeholder expects some value from the system
    - What value?
- User-level goals apply to particular actors
  - i.e., typically apply to particular use cases
  - Each user expects some result from using the system
    - What result?
Vision 3: system features

- What the system must be able to do
  - i.e., particular actions, events, processes, …
- X is a feature only if it makes sense to say: “The system shall do X.”
- Usually expressed in a list like:
  - Display chess board/pieces to players
  - Allow player to move a chess piece
- Clarifies the system’s requirements, and helps assign responsibilities to classes during design
Vision 4: other requirements and constraints

- Not functional requirements (like “features” are)
  - e.g., fast, cheap, scalable, extensible, …
  - One such characteristic may relate to several features
- Not responsibilities to assign to any class
  - Instead: things to consider throughout development
- Quantify if feasible
  - e.g., “will retrieve data record in 2 seconds or less”
- CS 48 note: this part of vision replaces most of the “supplementary specification” (section 7.4, Reading #7)
  - All except the functionality part of FURPS+
    - See next slide
The FURPS+ Model

- **Functional**
  - features, capabilities, security
- **Usability**
  - human factors, help, documentation
- **Reliability**
  - failure frequency, recoverability, predictability
- **Performance**
  - response time, throughput, accuracy,...
- **Supportability**
  - adaptability, maintainability, configurability,...
- **+** - implementation, operations, packaging, legal, …
What are use cases?

● **Answer:** *domain processes* in which the *system is a participant* – best described in story format
  – Note: a *scenario* is a particular instance of a use case

● Other participants are termed *actors*
  – Include users, other systems, and/or more abstract external things (like a specific date and time)

● **The system interacts with these actors**
  – An actor will initiate each use case
  – The system will respond in some way
  – An actor may respond to the system’s response
    ● And so on … until the use case terminates
Why describe use cases?

- Beneficial to the client
  - Shows exactly how the system works for users
    - Via step-by-step descriptions of user-system interactions
    - In non-technical language the client understands
  - Not as distracting as prototypes
- Can be used to drive the process
  - Analysis: “harvest” classes from use case descriptions
  - Design: begin/terminate system sequences, satisfy user interface needs, and more
  - Implementation/testing: insure each case is realized
- Can expose “abuse cases” and “useless cases”
Use case diagrams

- UML to show the functionality of the system from the user’s point of view
Use case descriptions

- No strict format, but probably best to include at least the following:
  - **Name** of use case – first word should be a verb
  - **Primary Actor** (or actors; never including the system)
  - **Main Success Scenario**
    - a.k.a. “Basic Flow” or “Typical Course”
    - step-by-step *interactions* – steps are numbered for easy referencing – can be 1 or 2 columns (2 are easier to read)
  - **Extensions**
    - a.k.a. “Alternative Flows” or “Alternative Courses”
    - Listed at the end, and referenced by step number
    - All conditional branches should be here, not in the basic flow
About types of use cases

- Often useful to classify in terms of importance:
  - **Primary** – for major common processes, such as “Buy Items” in the POST system
  - **Secondary** – for minor or rare processes, such as “Request for Stocking New Product”
  - **Optional** – may or may not end up in the system

- And a continuum of types in terms of detail:
  - **Essential** (no design details) – “user identifies self”
    - Most appropriate for early stages of development
  - **Real** (more explicit) – “user enters ID on keypad”
    - Defer to design stage – otherwise limits design possibilities
Defining use cases in practice

- Iteratively/incrementally – like everything else!
  - First, “façade” iterations – brief or casual, and essential
  - Next iterations add details – fully dressed, still essential
  - Later iterations get real – some implementation details
- Best if domain language only (no computer-speak)
  - Watch for clues of design details creeping in:
    - Too many consecutive system steps
    - References to database or other non-domain concepts
    - “if/else” structures in typical course of events
- *Extend* or *include* use cases if it simplifies things
UML <<extend>> stereotype

Passenger

PurchaseTicket

<<extend>>

OutOfOrder

<<extend>>

Cancel

<<extend>>

NoCharge

<<extend>>

TimeOut
UML <<include>> stereotype

Passenger

PurchaseSingleTicket

PurchaseTicketPackage

<<include>>

<<include>>

CollectMoney

<<extend>>

NoCharge

<<extend>>

Cancel

<<include>>
Use cases and development

- Assign a use case to an early iteration if it
  - significantly influences core architecture
    - i.e., has many domain classes/concepts, is a primary system purpose, involves risky technology, …
  - requires lots of research, or has complex calculations
    - So might have to start it early to get it all done in time
  - is a “time-critical” case (needed early by the client)
- Secondary, and or optional use cases can be developed later (incrementally)
- So can complicated use cases (iteratively)
Planning development iterations

- One iteration includes: analyze, design, code, test
  - “Analyze and design a little, code and test a little, …”
  - Best if about 2-10 weeks (in CS 48, 3-4 weeks each)

- Main reason for an iterative/incremental process: manage complexity
  - Can easily lose focus if iteration is too long, and/or tries to tackle too many details

- Note: also plan to synchronize artifacts to code after each iteration
  - Analysis and design occur during coding and testing too
2 implementation issues to plan

● Overall system architecture – typically use “layers” in an object-oriented system
  - Top layer: Presentation
  - Middle layer: Application logic (domain, services)
  - Bottom layer: Storage

● Test risky ideas early
  - Especially connections to other systems
  - Also any tricky or complicated algorithms