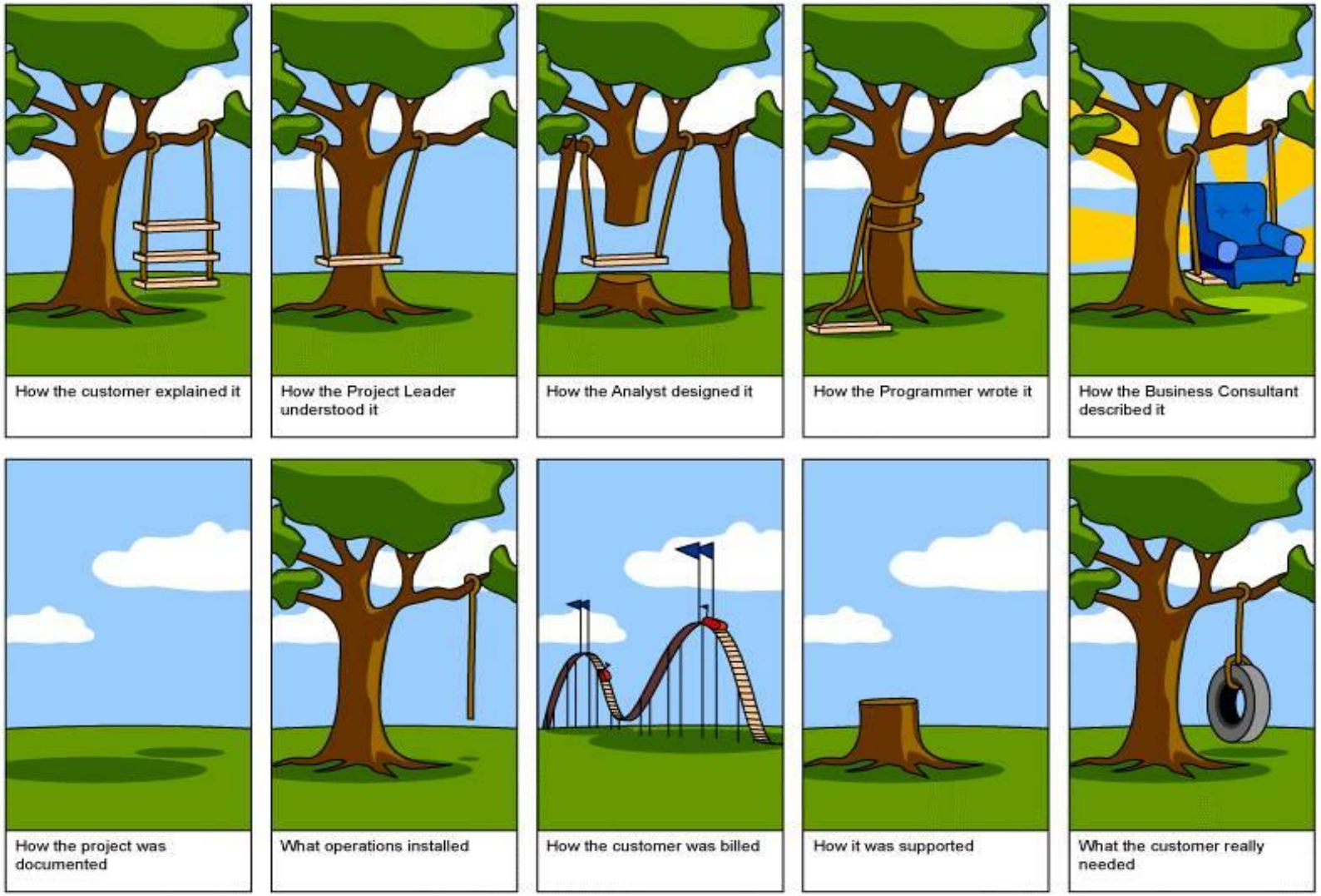
SE3430/5430 OOAD F13
Lecture Notes 04

Kun Tian

Requirements Engineering

- **What** will the system do?
  - Easy thing to do is postpone thinking about the details until later – "we'll get to this stuff later."
    - You must have a clear understanding of "what is required" before you implement the solution.
- Requirements engineering systematically identifies requirements to minimize errors and risks.
- Ensures requirements are necessary, sufficient and good quality: determine:
  - Is it a real requirement
  - Is it actually needed
  - Improve the quality of reqs
- It is necessary to let stakeholders have the same view of reqs.

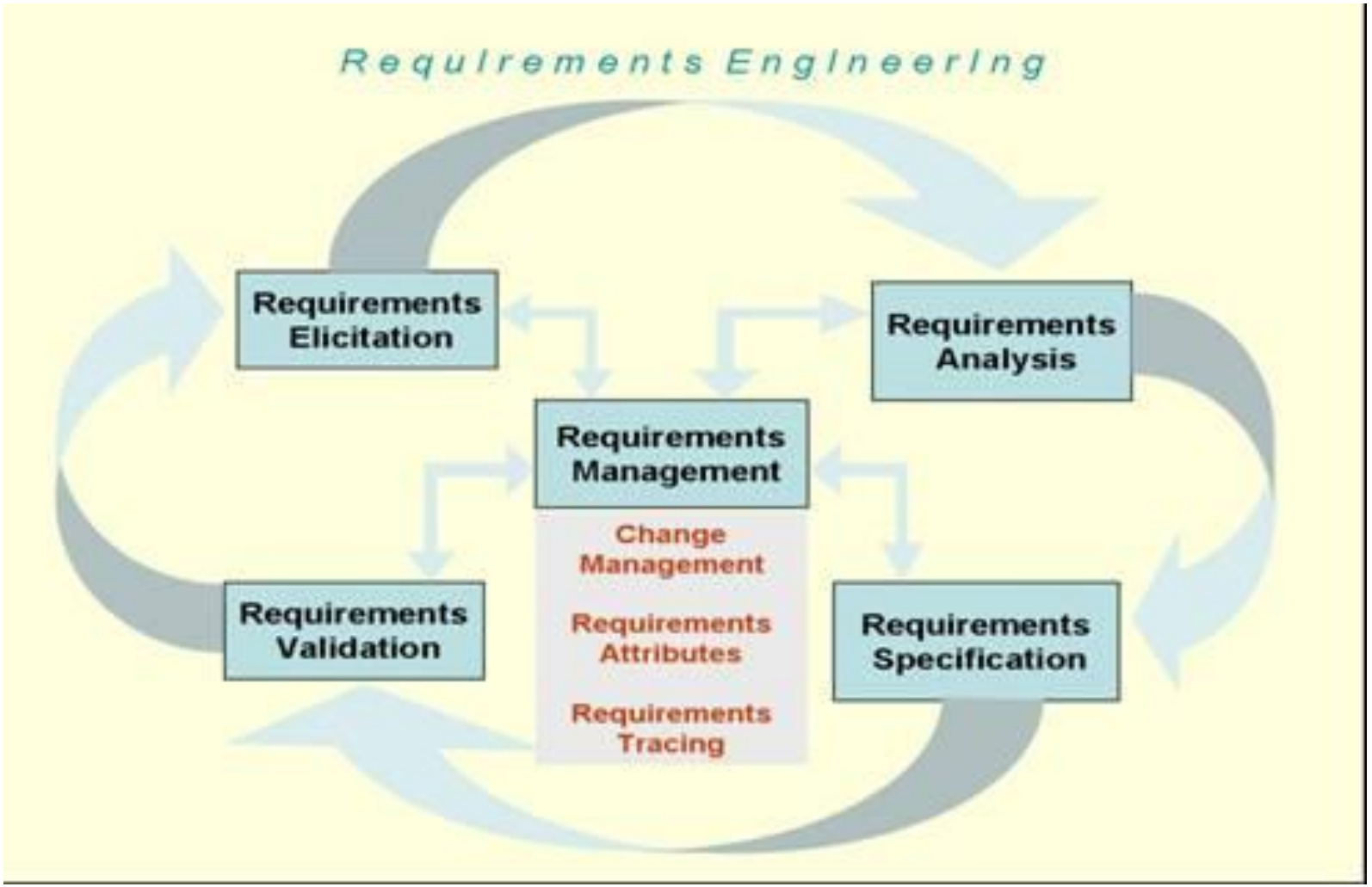
Requirements Engineering



Phases of RE

- Four Key RE phases.
  - Elicitation – Obtaining enough data/information/knowledge about a system so that a complete and formal specification can be produced.
  - Specification (and Analysis) – Involves the assimilation and analysis of the data/information/knowledge about a system so that we can organize and produce a formal record of what the system should do.
  - Validation – Testing the formal specification of the requirements w.r.t. their Completeness, Consistency, Verifiability, Modifiability, Traceability, Priority, …
  - Management -Tracing, prioritizing and agreeing on requirements and then controlling change and communicating to relevant stakeholders.

Phases of RE


Requirements Engineering
Requirements Elicitation
Requirements Analysis
Requirements Management
Change Management
Requirements Attributes
Requirements Tracing
Requirements Validation
Requirements Specification

Obtaining Requirements -- Elicitation

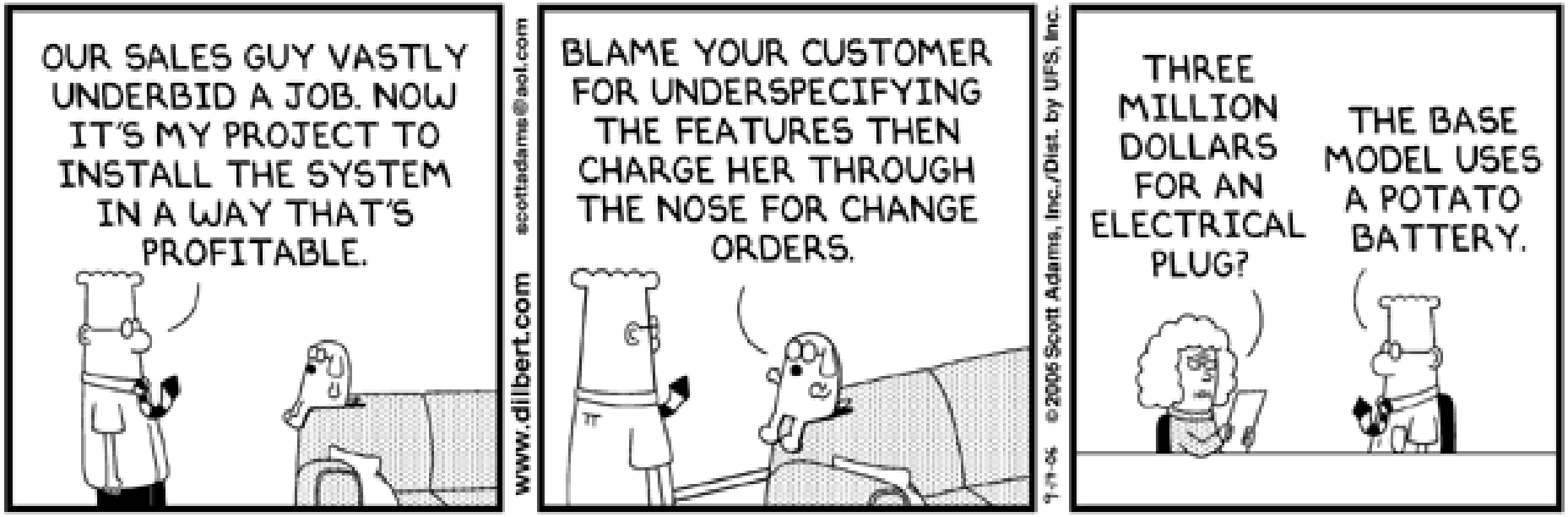
- What are the challenges of requirements engineering?
  - Only the simplest systems can be **fully specified :Too much = Too few**
  - Determining what is **important/critical** and what is **fluff**.
  - What are the **benefits** associated with each requirement? What will be missing in a system if any requirement is left out.
  - Users may be very knowledgeable about their domain. But,
    - They may have trouble **expressing the details** in words.
    - They may use jargon with which we are unfamiliar.
    - They may not understand the language of Software folks.
      - Grandma doesn't know what a database is.

Obtaining Requirements -- Elicitation

- Users may have no idea **what will be difficult to implement** and what is a piece of cake.
- It is difficult to **talk abstractly** about a problem with most users – they don't see that generalization is possible or present.
- Requirements Engineers may not have any idea **how a particular domain functions**.
- No one person may know **the big picture.**
  - We may need to extract bits and pieces from many users and then integrate what one gets into the big picture (jigsaw) (beware of gaps).
- Different stakeholders may have **non-equivalent views** of requirements and their priorities.
- Requirements must be **verifiable**. (arguably)

Obtaining Requirements -- Elicitation

- Bad Requirements Leads to Misunderstanding the Project



Ways to improve requirement quality

- Go and **study the domain directly**; go sit in a bank; go watch Campbell's make soup, go look at a satellite; go to a cheese plant, go on a brewery tour, … go to the domain and watch it.
- Have **multiple stakeholders review requirements and agree** on
  - validity,
  - importance,
  - completeness, and
  - priority.
- **Track changes** in requirements and **get concurrence** on validity, importance, and priority of the changes. (Requirements Management)

Ways to improve requirement quality

- Requirements need to be understood by many stakeholders. Having requirements that are only understood by a developer may not be good requirements.
  - Requirements should be stated in the terminology of the customer's domain.
- We as software folks may want to include a glossary of terms that are non-common so that we can make sure that we, as a requirements engineer, and others who will use a requirements document will understand terms.
- Feasibility / Cost - with stakeholders, identify those requirements that are absolutely required.
  - Estimate costs so the customer can weigh the value of each requirement. They may change their mind on what is required based on the cost.

Characteristics of good requirements

- Make sure each requirement can be verified (**testable)** – a good test of a requirement is one which you can write a test case to verify.
  - Help identify **risks**.
- **Language**: readable, precise, unambiguous, break complex sentences into simple sentences, use terms consistently – use your project glossary, organize the requirement (by functionality, real-world components – something that the users and developers both can follow.

Practices

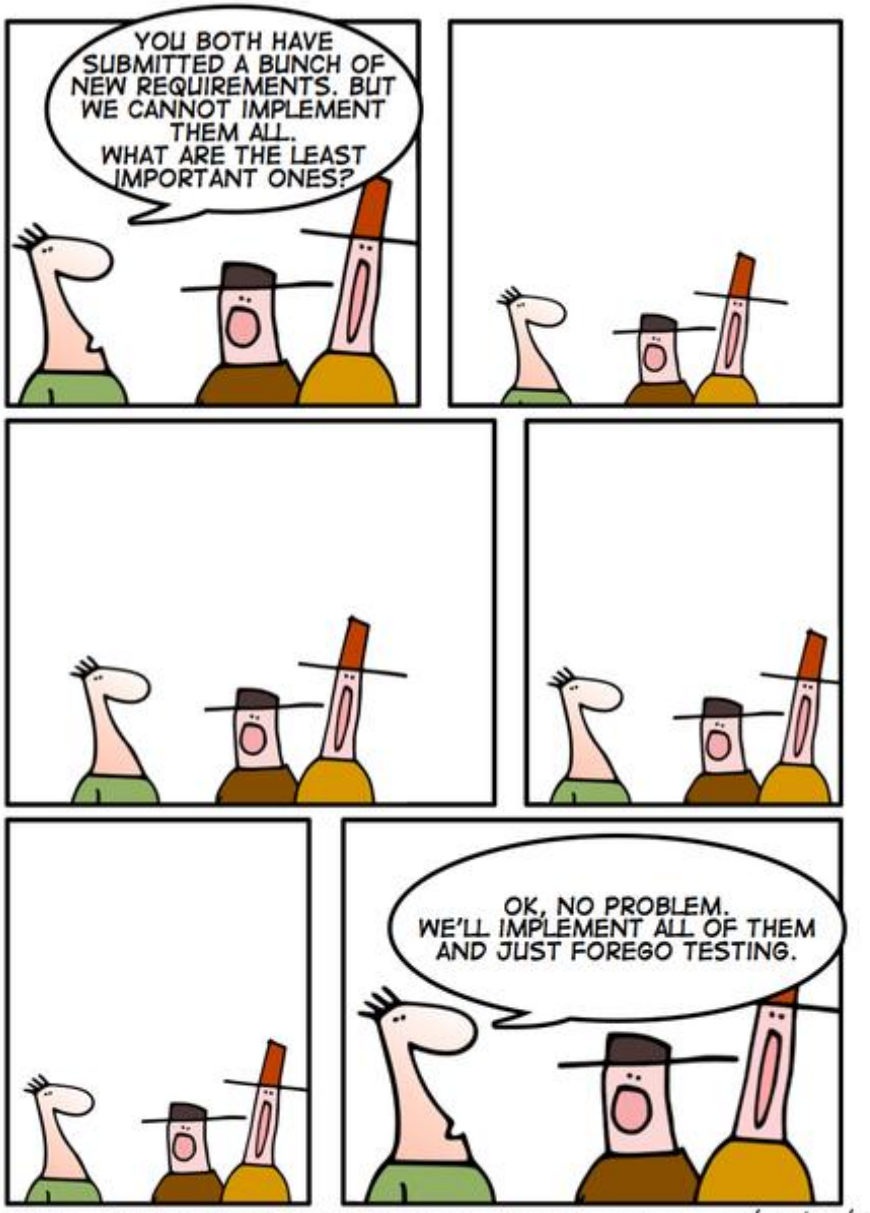
- **Baseline** requirements as early as possible.
  - **Baselining** involves getting the customer to approve them and putting them under configuration control. This way everyone knows the state of all requirements, including:
    - Their content
    - Who approved them
    - If they were changed – how were they changed
    - Who approved the changes?
- In practice: **need to start analysis and design before all requirements can be collected.**
  - We may never get all requirements collected.
  - Try to get the critical ones as early as possible to get enough for a good start.

Practices

- Be prepared for **requirements** to **change** – they always will.
  - Hopefully, the major requirements will be relatively stable and it will be the minor requirements that change most. (reqs for domain abstraction will be stable, reqs for solution space absctraction?)
  - Phased development approaches help as you implement the overall structure first and then add details as you go on.
  - If major requirements keep changing, one solution is to let the customer know what the impact of each change is w.r.t. Schedule and Money, … (also a tactic to push customers to fix them)
- **Traceability of requirements** – since we must expect changes in requirements, (when something is changed, the change can be propagated throughout the system.)

Practices

- **Requirements Engineers** must have many **skills** –
- Must be able to rapidly **assimilate** the key features of new domain
- Must **communicate** well (Talk, Read and Write Well)
  - Be able to talk with users at all levels
  - Be able to read and comprehend technical material and documentation
  - Be able to write well to capture the requirements in a form that others can understand.
- Must be able to **relate at many** levels to many **stakeholders** from the deep carpet all the way on down to the concrete.
- Must have **political skills** – refrain from telling a stakeholder they are "full of crap".
  - Must help customers resolve differences in priorities and wishes.



THE CONSULTANTS HANDBOOK PART 9:
THE CUSTOMER IS KING

geek and poke

The [IEEE STD (830-1998)] defines eight (see table1) major quality criteria for *Software Requirements Specifications*, which are also applicable to individual requirements.

Core Quality Criteria for Requirements IEEE STD (830-1998)]	
Correctness	A requirement is correct iff the constraint / capability stated therein is one the software shall meet. VALID
Unambiguity	A requirement is unambiguous iff all information stated therein has only one interpretation.
Consistency	A requirement is **internally consistent** iff no individual constraint/capability stated therein conflicts, and a requirement is **externally consistent** if, and only if, no other requirement conflicts with it.
Completeness	A requirement is complete iff it defines the response of the software to all realizable classes of situations. This includes both valid and invalid inputs.
Ranked for **Importance** (Priority, Necessity and Stability)	A requirement is ranked with respect to importance and/or stability. **Often we use the identifier to encode this "R" – required; "O" – objective or optional (not required). May use "P" for pending.**
Verifiability (Testable)	A requirement is verifiable iff there exists some finite cost-effective process with which a person or machine can determine if the software meets the requirement.
Modifiability	A requirement is modifiable iff its structure and style are such that changes can be made easily, completely, and consistently. Lack of coupling and redundancy across requirements helps keep requirements consistent when changes occur. ***Hint**: Reference or link details in other requirements rather than restating them.*
Traceability	A requirement is traceable iff it has a clear origin, all modification reasons are tracked, and all future artifacts can be easily mapped back to requirements.

Other characteristics not specifically covered by IEEE 830	
Understandable	To facilitate validation, requirements should be understandable by the end user. This includes using the terms of the domain and the user. A glossary may be useful for the software people to understand the user domain terminology. **Keep the language level at the level of the end user.**
Non-prescriptive	Should stick to WHAT the system needs to do and NOT HOW the system will be designed or implemented.
Concise	"Less is more", don't add material that is not necessary. Rambling hides information and confuses old folks like me.
Precise	A hybridization of Completeness and Unambiguity. See below.
Feasibility	Reasonable with respect to economic, schedule, resource and technological parameters.

Testable (verifiable) Requirements

- If **all parties** cannot **agree** that a requirement has been satisfied it is of little value (and is a potential liability).
  - Example: "The software error rate will be at most one failure per 1000 hours of use."
    - untestable: what load will the system be under during the 1000 hours, idle or heavy?
    - qualifying it with exactly under what conditions we will test for 1000 hours. Need something more specific than something like "normal use".
    - Also to be reasonably certain about 1 failure per 1000 hours we would need to test it over many 1000 hour periods!
      - Can we test something too much?
- **vague**: "software will be user friendly"/"respond quickly"/"efficient" use "pretty colors"
- Go beyond precision: must be able to establish a test to determine if system meets a given requirement.
  - We must be able to answer YES or NO to whether requirement has been satisfied.
- **Writing test plans** while writing requirements helps find untestable requirements!

Non-prescriptive Requirements

- ***what***, not ***how***
- *We want the opportunity to analyze and design and then determine technologies*
- Constraints deal with technological issues (must use MS .Net or Linux). This may just be reality and is not prescriptive.
- Types of reqs:
  - Functional
  - Non-functional
  - Constraints

Requirements should be Concise (succinct)

- **Who likes to read overly wordy descriptions**?
  - Quote from book:
- "We feel that good systems provide the end user with good value. Because of this, we think that the system should provide adequate performance with a 200 GB disk, since this is the least expensive disk that we may purchase from the designated vendor. Of course, the user may elect to configure the system with a larger disk, and we recommend this, but we have attempted to come to grips with most of the problems raised by use of the smaller disk, and we feel that they can be, by and large, satisfactorily resolved."

Requirements should be Concise (succinct)

- How could the above be made better?
  - Better: The system shall fulfill all specified functions when configured with a 80-GB disk.
- Rambling prose hides information. Less it more!

Requirements should be Precise

- Don't allow for misinterpretations or room for assumptions.
  - Example: "The system shall accept valid employee ID numbers from 1 to 9999."
    - Are all values between 1 to 9999 valid, or can some be invalid?
    - Are leading zeros, like 0001, acceptable/required?
    - Better: "The system shall accept only valid ID numbers as defined in [reference]. All valid ID numbers are in the range 1 to 9999 inclusive, represented without leading zeroes.

Requirements should be Unambiguous

- Worst case: both developer and client have a different meaning in mind, but neither one realizes the other is thinking something different
  - Example: "The system shall be capable of seeing the person in the park with a telescope."
- Be careful with **pronouns** and **prepositional** phases:
- Issue: what's a requirement and what's a design suggestion
  - use **shall** to indicate an actual requirement
  - use **may** for suggestions or wishes.

Requirements should be Modifiable

- Keep information in a single place
  - Rather than referring to a limit such as "support 1000 customers" over and over and over each time we need to talk about the limit, we need to specify the limit in one place and refer to that specification elsewhere.
  - Example:
    - R-4.1 The system shall allow up to 1000 customers simultaneously logged on.
    - R-8.1 If the maximum number of customers (see R-4.1) are logged on, new attempts shall be rejected with the error message "The maximum number of customers are currently logged on, please try again later."

Requirements should be Modifiable

- Don't merge a number of requirements together
  - Example: R0: The program will process each customer, flagging all those customers that are out of order and creating an error log for those customers, which includes both the customer before, the invalid customer and the customer after.
  - Better:
    - R1: The system shall process each customer one at a time.
    - R2: The system shall flag a customer if they are out of order.
    - R3: The system shall create an error log of out of order customers that includes the following: preceding customer, the out of order customer, and the customer following the out of order customer.
  - This also helps in status tracking. For example R1 and R2 may be implemented and tested, but R3 may be implemented and fails its test. Whereas if we have just R0, then we do not have as much precision in tracking status.

Requirements should be Feasible

- **Economic feasible:**
  - May not know that something is impossible till later. Try to identify risks A.S.A.P.

Verification techniques for Requirements

- Requirements are the input to the Design Process. The quality of our requirements will impact the quality of the Design, Code, Tests, and the final deployed system.

Inspection

- Inspections help to improve the quality of requirements by detecting defects in the requirements documents. In addition to that, inspecting a document in a systematic manner teaches the developers to write better requirements documents.

Inspection

- **ad-hoc technique:** a non-systematic way of identifying defects.
- **checklist-based technique:** the inspectors are provided with a list of general defect classes to check against.
- **review:** a manual process that involves multiple readers checking document for anomalies and omissions.
- **walkthrough**: a peer group review of a software document
- **scenario-based technique:** the use of scenarios to guide inspectors on how to find required information as well as what that information should look like.
- **perspective-based reading:** one step deeper than scenario-based, various people read the (requirements) document from a particular point of view according to the perspective represented by different stake-holders in the project.
- **usage-based reading:** a prioritized use case model is taken as input. It makes the inspectors focus on the defects that are important for the future users of the system.