Glossary of terms for SE 463
You need to pass the final to pass the course.

Please edit/fixup stuff that's wrong, and clean up stuff that's verbose.
My bs filter was set on "High", but a bit escaped.

No tutorial notes are in this document.

Acronyms: (TODO: sort)

- SUD - System Under Design (Development?)
- SRS - System Requirement Specification
    - RS - Requirement Specification
- SSMD - System-level State Machine Diagram
- DBBS - Dan Berry Bull Specifications
- UC - Use Case
- RS - Requirements Specification
- CBS - Computer Based System
- EFSM - Extended FSM
- NSB - No Silver Bullet
- FSM - Flying Spaghetti Monster or Finite State Machine, depending on religion (Pastafarianism vs DBerry).
- UML - Unified Markup Language
- RE - Requirements Engineering
- NFR - Non-Functional Requirements
- SR&R - Security, Reliability, & Robustness
- SW - Software (Soft Ware)
- US - Unnecessary Synonymy
- CP - Confusing Polysemy
- Recommended Practice for Software Requirements Specifications (RPSRS)
- S - Scenarios
    - Honestly, this guy is an acronym freak
- NL - Natural Language
- FL - Formal Language
- RFP - Request for Proposal
- MB FL - Math-based Formal Language
- SD - Subconscious Disambiguation
- SA - Subconscious Ambiguation
- SH - Synchrony Hypothesis
- RA - Requirements Analyst
    - Requirements Analysis
- FR - Functional Requirement
- QFD - Quality-Function Deployment
- LOC - Lines of Code

- KLOC - Thousand Lines of Code
- RD - Requirements Determination
- PotLoBoaDtiP - Problem of the Lack of Benefit of a Document to its Producer
- OCL - ???????? (JoAtleeTempLogic.pdf)
    - “Constraints on all world states of domain model.”
- IEEE - ? [standard acronym]
- LTL - Linear Temporal Logic
- LRM - LTR Mark / Left Directional Mark
- RLM - RTL Mark / Right Directional Mark
- LRE - LTR Embedding
- RLE - RTL Embedding
- RLO - ?
    - RTL ordering?
- LRO - ?
    - LTR ordering?
- PDF - ?
    - Pop ???
- CEL - Character Embedding Level
- AN - Arabic Number (e.g. ١,٤, …)
- EN - European Number (e.g. 1, 2, …)
- AL - Arabic Letter
- EL - European Letter
-

- [9-11] Sept
    - https://www.student.cs.uwaterloo.ca/~se463/BriefIntro2RE.pdf
        - Requirements are the things we want to do, and what the customer wants
        - Specification is the description of proposed behaviour of a system, like assignments
    - https://www.student.cs.uwaterloo.ca/~se463/RE-referenceModel.pdf
        - Hardware/software is the **System** that is designed to operate with an **Environment**, both of which are in the **world**.
        - **Systems (S)** are constructed artifacts by people, as a mix of software, hardware, humans, and processes.
        - **Environments (Env)** are the minimal subset of the real world relevant to the problem, (i.e. all that’s necessary to express the spec and reqs)
            - The **application domain** is the generalized environment for some types of problems.
        - **Interface (Intf)** are the shared phenomena between the environment and the system. (e.g. an API of interactions, shared resources, etc)
            - Interface = Environment ∩ System
        - **Requirements (R)** are desired changes to the world described in a high-level.

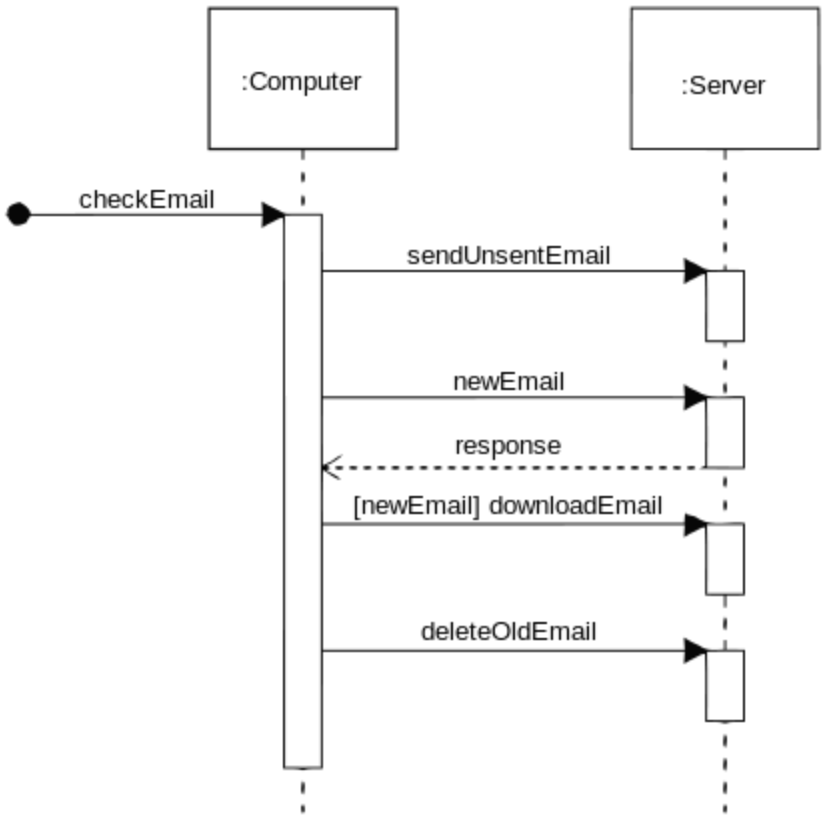
        - **Specifications (S)** are the plan to implement requirements.
            - Specifications describe systems that realize requirements.
            - **Design** is a way of implementing a specification. (think about pseudocode).
            - **Code** can be a realization of a design.
            - The vocabulary that the spec is written is a subset of the vocabulary of the interface.
        - A **requirements specification** (a 'spec') is a description of the proposed behaviour, expressed in terms of shared phenomena.
        - **Domain knowledge (D)** is the set of properties we know to be true of the environment. We need it to ensure our systems are correct.
        - Michael Jackson is a person that Specs & Reqs fantasizes about.
        - If you can't argue that the (specification + domain knowledge) are sufficient to satisfy requirements, then you must do (n >= 1) of:
            - Strengthen the spec
            - Strengthen the domain knowledge
            - Weaken the requirements
        - **Designation** is a mapping between a term in the reqs or spec to something in the environment that it represents.
- [16-19] September - https://www.student.cs.uwaterloo.ca/~se463/classes.concepts.pdf
    - Code is too detailed to see the architecture it contains.
    - It takes a while to write code, and you should probably think about it before you write it.
    - UML!!!!!!!!!!!!!!!!
    - Learn U some UML, fool!
        - It's defined by the **OMG** specification
        - It's so cool
        - It even uses the word **omg**. It's got to be cool
    - Requirements engineers are expected to be able to re-implement problem descriptions as UML models, even if they don't understand the words.
- [23-24] September
    - BiDi formatting: https://www.student.cs.uwaterloo.ca/~se463/brief.reading.lesson.pdf
        - Chunks are maximal length subsequences of characters all in the same direction.
            - LR text is read from left to right
            - RL text is read from right to left
        - **Time Order** of chars is when you consistently display text in the order it is entered, without reversing direction.
        - **Visual Order** of chars is when they appear in the order that you'd read them, the logical way that you'd read the chars in their native language.
        - The two orderings have the same number of chars per line, since each char has the same length.

        - We can swap between the two: reverse the order of each RL chunk.
    - **Documents** are the things that contain sub-documents. They have a directionality.
    - **Sub-documents** contain chunks of characters and other sub-documents. They have a directionality.
    - **Not the Unicode BiDi algorithm:**
        - Pseudocode:

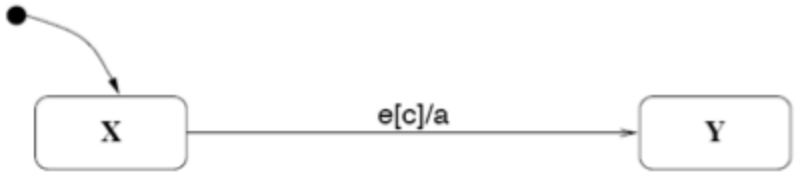
```
for each line in the file:
     if the current doc direction is LR
          reverse the ordering of each chunk of
RL characters in the line.
     else (is RL)
          reverse the ordering of each chunk of
LR characters in the line.
```

- [25 sept-1 oct]
    - Use Cases https://www.student.cs.uwaterloo.ca/~se463/scenarios.usecase.pdf
        - Ways to use a system S is called a **Use Case** of S
            - A use case of S is a set of scenarios where the choice of scenario being executed is decided by the situation that the use case is being executed in.
        - **Scenario** of a system S is a detailed description of the sequence of steps constituting one use of S by a user towards an objective.
            - that you'd use S is a sequence of interaction steps of a user of S and S
            - A single Use Case contains many scenarios.
        - A **typical scenario** is the scenario that happens during typical usage of the system.
        - An **alternative** of a use case A is a different use case that tries or fails to accomplish the same goals as A through different or nearly similar steps as A.
        - An **exception** of a use case A is the use case that arises from the result of typical scenarios being run with exceptional input.
        - A **misuse case** in a scenario is a use case designed to capture (and handle) illegal/malicious behaviour (cheating, etc).
        - All scenarios of a use case may share subsequences of steps.
        - By presenting users with scenarios, they'll explain how they use or will use systems.
        - Questions help you prevent:
            - Inconsistencies
            - Incompletenesses
            - Aliases
            - Omissions

        - etc.
    - **Use Case Diagram**
        - Stick Figures are connected by lines to bubbles (indicating use cases) inside a box.
        - dashed <<MBA>> Arrows indicate order
        - sub-use-cases are connected using dashed <<include>> arrows to the ones they are needed by.
    - **Sequence Diagram**
        - Refer to image

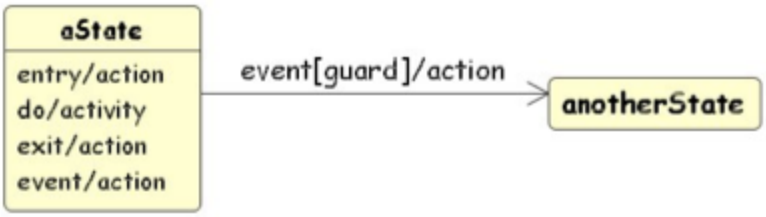


- [2-3] October
    - UML State machine diagrams: https://www.student.cs.uwaterloo.ca/~se463/StateMachineDiagrams.pdf
    - UML-SMD is useful for describing how classes conform to state-machines
        - UML can S My D
    - Elsewhere, we don't assume "inner" states exist in the individual state machines.
        - UML-SMD is badass, and has extra "inner" states
    - Finite state machines.
        - Read about their impl elsewhere. Berry is Berry.
        - They're deterministic
        - They will recognize all regular languages
            - recognized -> in set of regular languages

        - if A and B are regular languages, then:
            - union - (A ∪ B) is regular
            - concatenation - (AB) is regular
            - star - (A*) is regular
                - A* = AA*∪epsilon
        - FSMs can simulate all regular expressions
        - Regular expressions can express all FSMs
        - 

            - When in state X, if event e occurs and condition c is true, do a, and move to state Y
            - All parts (e, c, a) are optional.
    - **Extended FSM** - FSMs with variables to reduce the number of states in the model (not necessarily finite)
        - Transitions can depend on the value of conditions on variables
        - Outputs can be sent messages or assignments
        - Probably not a regular language
        - UML state machines are EFSMs
    - While requirements engineering, we may build a SSMD
        - The SUD is the outermost machine, and we compose it of SMDs indicating systems connected using transitions.
        - We start to build something that looks like the final implementation
    - In a **Design Model**, we create state diagrams for the design classes, and build a state machine connecting these.
    - States can have actions and activities associated with them:
        - **Actions** are instantaneous, non-interruptable, and simple. They are performed once in a state (entry, exit, etc)
        - **Activity**ies are interruptable
    - Events without handling state transitions are ignored.
    - **Over-specification** is when there are:
        - Responses to an event that can't occur in a state
    - **Under-specification** can occur when responses to an event don't exist where they should exist.
        - Just use brute-force to verify, you should be ok.
    - **Hierarchical States** are submachines that exist inside EFSMs
        - One transition leaving a superstate can rep many transitions in a flat state machine.
        - When there are conflicting transitions with the same input within state hierarchies, use the one that is closer to the scope that you are in (the

lower one)
        - **Final States** inside hierarchical states lead to unlabeled exits of the hierarchy.
        - **History** is a pseudo-state that designates the immediate sub-state that we were last at, almost like a coroutine.
            - **Deep History** (denoted H*) denotes that all sub-machines are at the same state, not just the single sub-machine being resumed.
    - Composition icon - “has real content, but is defined elsewhere”; uses entry and exit points
    - Junction points exist
    - Naked transitions come out of things with **do/** activities
    - Change events: when(X), like events but on conditions involving variables
    - Time event: at(9:00 am, 9 Oct, 2010); after(20 minutes)
    - The **Synchrony Hypothesis** assumes that the system can respond to input faster than other input can be provided.
        - i.e. there are no concurrency issues
    - For a SSMD, we have the following reference nodes



        - Running multiple actions can be accomplished using a semicolon
    - Good state machine diagrams are clear ones
- [9-10] October
    - UML State machine diagrams: https://www.student.cs.uwaterloo.ca/~se463/StateMachineDiagrams.pdf
        - See 2-3 October
    - User Manuals https://www.student.cs.uwaterloo.ca/~se463/UserManual.pdf
        - Opinions and ideas expressed herein are due to Prof. Dan Berry and his collaborators…
            - Some of them are very personal and even controversial!
            - wtf, I don’t even
            - This was written in 1991
            - Godfrey removed like 45% the slides for us, since they’re useless
                - I like this guy
                - (heart)
        - The most useful docs for RE are the UM
            - They force the users to fix things they don’t agree with/etc
        - Writing a UM ameliorates problems that discourage production of RSs.
        - Good UM should:

        - Describe the CBS's function (not impl)
        - Describe the CBS from the user's view
- Apparently, the UM for Lisa & Mac Computers were written and given to devs as impl instructions.
    - I'm not sure if Berry is correct (neither is he), but this would be cool.
- UMs give us:
    - **(Requirements)** The UM creation process gives us information for the RS.
    - **(Stakeholders)** The UM creation process helps reconcile differences between stakeholders.
    - **(Pre-validation)** A UM allows customers validate that the solution will be what they want before building it.
    - **(Specification)** A UM makes it clear what needs to be implemented.
    - (**Post-validation**) A UM allows us to write test cases and paths to verify the implementation
- If RSs are too long and hard to write, write a UM instead.
- DBerry suggests writing a set of UMs, one for each kind of user.
- Writing a UM requires a clear conception of what the system is supposed to do, enough that the writer can visualize use cases.
- Good UMs "seem to have":
    - Descriptions of fundamental concepts of the software (a lexicon)
    - A graduated set of examples, each showing:
        - A problem situation
        - A possible set of user responses
        - The software's response
    - A systematic summary of all the commands
- DBerry's Protip: if you want English Majors to write very technical and in-depth UMs, refer to Bob Glass.
- Outline for a UM:
    1. Introduction
        a. Product overview and rationale
        b. Terminology and basic features
        c. Summary of display and report formats
        d. Outline of the manual
    2. Getting Started
        a. Sign-on
        b. Help-mode
        c. Sample run
    3. Modes of operation
        a. Commands/dialogues/reports
    4. Advanced Features

5. Command Syntax and System Options

- You can use a UM as a RS if the UM easily encompasses all functionality, and it isn't awkward to describe in prose.
  - This fails for:
    - Autonomous systems
    - Systems with NFRs not specifically visible to users (security, reliability, and robustness requirements)
- All methods for writing RSs forces the details to be worked out, and UMs have (questionably) justifiable utility, which is more than a (previously-defined) RS
  - Changes need to take place in one of:
    - Software
    - Formal Specification
    - Traditional SRS
    - Complete, scenario-based UM
  - It's cheaper to make UM changes than in the other places, since that doesn't require dev work, and can be done while other development work occurs
  - It's harder to BS on a UM, since it's supposed to be actually delivered to users

- UM Advice https://www.student.cs.uwaterloo.ca/~se463/users.man.pdf
  - Choose between (and do not mix the two)
    - second person imperative sentences with implied subject of "you",
      - You enter 'exit'
        or
        Enter 'exit'
    - "the user"
      - The user enters 'exit'.
      - He enters 'exit'.
      - NOT: they enter 'exit'
  - Maintain a Glossary of terms to prevent US and CP
    - US - Unnecessary Synonymy
      - Using more than one word for the same concept (cursor vs indicator)
    - CP - Confusing Polysemy
      - Using one word for two different concepts (OS for Operating System vs Open Source)
  - Pluralization of acronyms
    - Never pluralize the contents of acronyms
    - Add an s to the end of an acronym to pluralize it, even if it would be added to the middle
    - Ex:
      - Request for Proposal = RFP

        - Requests for Proposals = RFPs
    - UMs should be written well enough that people are convinced that they exist, even if they don't.
    - Always use present to talk about the CBS.
    - "Shall" is reserved for describing requirements, be careful of how you use it.
        - After clicking, the system shall XYZ
        - Don't use: ~~After clicking, the system will XYZ~~
    - Parentheses should only contain content that can be skipped when reading UMs.
    - Quotation marks can be used to enclosed a quotation or phrase used to indicate irony, or a quotation:
        - 'The word "word" has a "w" and three other letters.'
        - Dan Berry "doesn't know" requirements
    - Quote I/O using a specified style. (font/etc)
    - Specifying "what" a system does and not "how" gives implementers the greatest freedom.
    - By example, dberry sees SRSs as being incomprehensible compared to UMs.
        - Users are more likely to accept SRSs that they don't understand vs incomplete UMs.
    - Use notation that makes sense to the people reading the UM.
        - Look at what they use when they provide specifications, this is what they use.
- 11 Oct
    - Unicode BiDi Algorithm https://www.student.cs.uwaterloo.ca/~se463/UnicodeBiDiAlgorithm.pdf
        - For the typesetting of any document, each mounting of each font is given a direction, and its period has that direction.
        - You're better off learning about the algorithm here http://www.w3.org/International/articles/inline-bidi-markup/uba-basics
            - The ordering chars in memory (logical) is not the same that they are displayed in (visual)
            - The base direction of a phrase, paragraph, or block is contextually important to rendering of the text.
            - The bidi algorithm renders a sequence of arabic or hebrew characters one after another from LTR or RTL, respectively.
            - Most unicode letters are strongly typed - they have a direction, and it's either RTL or LTR.
                - Sequences of strongly-typed RTL characters will always be drawn in a RTL fashion.
            - When text alternates, the bidi algorithm automatically produces separate directional runs out of each sequence of contiguous

characters.
        - i.e. Hello(1), ׳(2)דן Berry(3)!
    - The order in which directional runs are displayed depends on the base direction.
        - In a LTR document, we have as before:
            - Hello(1), ׳(2)דן Berry(3)!
        - In a RTL document, we have:
            - Berry(3)! ׳(2)דן Hello(1),
            - Fucked up, amirite?
    - Yet, some characters are neutral (or weak) - commas, spaces, etc. The Bidi algorithm determines how to handle weak characters by looking at surrounding characters.
        - Notation
            - Rep Neutral as N
            - Rep RTL as R
            - Rep LTR as L
        - Then:
            - char seq: RNNRNRN is displayed as char seq: RRRRRRR
            - char seq: LNNLNLN is displayed as char seq: LLLLLLLL
    - When weakly typed characters are in boundaries, they are treated as having directionality as the base direction:
        - char seq: LNR in base of RTL displayed as char seq: LRR
        - char seq: LNR in base of LTR displayed as char seq: LLR
        - char seq: RNL in base of RTL displayed as char seq: RRL
        - char seq: RNL in base of LTR displayed as char seq: RLL
    - Weak characters (arabic numerals, currency symbols, etc) do not finish the run of the inclosing segment, even though they may appear in a run of RTL characters in a LTR document.
        - i.e. abcd הוזחט 1234 אבגד efgh
        - the 1234 doesn't break the run into two parts.
- [16-17] October
    - SRSs: https://www.student.cs.uwaterloo.ca/~se463/SRSs.pdf
        - The IEEE has a spec for SRSs (RPSRS)
        - All SRS are essentially RS, but in a specific domain.
        - The RPSRS specifies how a SRS should be.
        - SRS should address the specifications of:
            - Functionality
            - External Interfaces
            - Performance
            - NFRs
            - Design Constraints

- SRS does not address:
    - Process requirements
    - Design decisions
- Here's a rundown of the IEEE segments:
    - Introduction
        - Purpose
            - Who is this for?
            - What does this solve?
            - ½ a page
        - Scope
            - Name
            - Overview
            - Summary of software, benefits, and golas
            - Boundaries of product
            - ¼ - ½ a page
        - Definitions, Acronyms, Abbreviations
            - Domain-level definitions used in the SRS
            - Naming conventions developed while writing the document
            - Notational conventions for deviations from standard UML
        - References
            - Sources of information
        - Overview
            - Brief description of the structure of the rest of the SRS
            - Organization of structure 3
            - Discussion of any deviations from the standard SRS format.
    - Overall Description
        - Don't state requirements here
        - Product Perspective
            - Describe the environment of the system.
            - YOLO use a diagram
            - Env components that may constrain requirements
        - Product Functions
            - Overview the system's features, a complete textual summary of UCs
            - UCs will be specified in detail later, so don't bother going in depth.
            - This describes desired functionality
        - User Characteristics
            - Document assumptions about anything.

                - User, background, training necessary, etc
            - This describes users' backgrounds that may affect usability.
        - General Constraints
            - Sources of other constraints on requirements
                - Standards / Regulations / Laws
                - Hardware
                - Parallel Operation
                - Auditing
                - Controls
                - SR&R
        - Assumptions & Dependencies
            - Assumptions about input/environmental behaviour, such that hardware fails, etc
            - Conditions that would cause the system to fail
            - Changes to the environment that could changes to the software runtime?
- Specific Requirements
    - Use this section of the SRS to contain all of the SRS.
    - This should contain descriptions of:
        - All interfaces to the system
        - All functions performed by the system
    - IEEE specifies the following organization:
        - External Interfaces
            - Contains descriptions of all inputs and outputs
        - Functional Requirements
            - UCDs
            - Sequence diagrams
            - Domain Model
            - Functional specifications
            - State machine model
            - Constraints
        - Performance Requirements
            - Should enumerate performance requirements, in concrete terms
        - Design Constraints
        - Quality Attributes
            - Nonfunctional properties expressed as testable constraints.
    - UW specifies the following organization:
        - 3.1: External interfaces
            - GUI

                - GUI events
                - Hardware Interface events
            - 3.2:
                - 3.2.1: Use Cases
                    - UCD
                    - Includes use-case descriptions and alternative flows that correspond to the functional specifications
                    - No sequence diagrams
                    - As usual, input and output definitions should be consistent
                - 3.2.2: Domain Model
                - 3.2.3: Functional Specifications
                - 3.2.4: State Machines
            - 3.3: Performance
            - 3.4: Design Constraints
            - 3.5: Quality Attributes
            - 4. Appendix
                - Requirements table
                - Functional Requirements
                - NFRs
                - Index of the entire SRS (ofc)
            - Glossary
                - Don't derp. Don't get the herp.
                - This is the glossary.
- [23-24] October
    - Why UIs Suck https://www.student.cs.uwaterloo.ca/~se463/user.interfaces.pdf
        - Implementers suck at implementing UIs, and we're good at what we do (Specifying things). Let's do that too.
        - UIs should be consistent with requirements
        - We need to validate UIs through usability testing
        - Honestly, the best way to spec UIs is to include them in your SRSs/UMs
            - Use figures, markers, and drawn steps on screenshots/figures/etc.
        - UIs only illustrate functionality from a users point of view, so they don't form a complete specification.
            - Use the same numbered marker to correspond between UCDs, FSMs, and UIs to tie them together
        - Software Sucks because UIs suck
            - i.e. jobmine
            - This is a cheat code for making good software. It's all in the UI. - scraig extrapolating from dberry's comments
        - Usually, there is no manual, help system, and users need to figure out

what's going on. They feel stupid.
    - Programs should be designed so that consulting a manual or help system becomes unecessary.
    - Instead, users are lost using UIs.
    - UIs suck because the programmers, and not the UI designers design and implement the program's UI.
        - Typical programmers program the UI for a user like themselves.
        - Typical users aren't programmers
    - "Know your user, for he is not thee" - Platt
    - Make your program easy to use.
    - Lostness can be quantified, of course:
        - R = min num of pages visited to do a task
        - S = num of pages actually visited to do a task
        - N = num of **different** pages actually visited to do a task
        - L = "lostness" of the user. $L \in (0, 1)$
        - $L = \sqrt{\left(\left(\frac{N}{S} - 1\right)^2 + \left(\frac{R}{N} - 1\right)^2\right)}$
            - Who knows if you need to memorize this.
                - Apparently you don't.
            - 1 is totally lost
            - 0 is not lost at all
- [28-29] October
    - Ambiguity in RS https://www.student.cs.uwaterloo.ca/~se463/Ambiguity.pdf
        - Most RSs are written in NL.
        - NLs:
            - Someone can always write these
            - Pretty well understood by stakeholders, but it can be understood differently
            - They are ambiguous
            - We fucked, yo.
        - MB FLs
            - Unambiguous by design
            - Few people can write it
            - Not understood by many, but those that do agree on what it means.
        - This makes NLs inevitable, as SD causes the reader of an ambiguous phrase to not realize that it's ambiguous.
        - We're really screwed if the RA is the one who doesn't notice the ambiguous discussions he's having with clients.
        - Semi-formal languages (UML) can be written from ambiguous conceptions of the model, and can be ambiguous when written
        - Avoid problems:
            - Write less ambiguously

        - Detect ambiguity
        - Use a restricted, unambiguous NL
- There are 11 papers he references of people building tools to find potentially ambiguous phrases
- Don't vary what you call an object, because it's ambiguous as to if they're different.
- **Lexical Ambiguity** - When a word has several meanings
    - **Homonymy** - when two different words have the same written and phonetic representation
    - **Polysemy** occurs when a word has several related meanings but only one etymology
    - **Systematic Polysemy** is when the reason for the polysemy is confusion between two classes (ie. between unit and type vs process and product)
        - TODO: I don't get this
- **Syntactic Ambiguity** - When a given sequence has another grammar structure with a different meaning.
    - **Attachment Ambiguity** is when a sentence can be split into two parts:
        - The police shot the rioters with guns
    - **Coordination ambiguity** is when conjunctions are used ambiguously:
        - I saw A and B and C saw me.
        - A young man and woman (is the woman young?)
- **Semantic Ambiguity** - When a sentence has more than one way of reading it within its context
    - **Coordination Ambiguity** (see above)
    - **Referential Ambiguity** - see pragmatic ambiguity
    - **Scope Ambiguity** - e.g. All linguists prefer a theory
- **Pragmatic Ambiguity** - When a sentence has several meanings in its context.
    - Every student thinks she is a genius.
- **Generality & Vagueness** -
    - Generality:
        - Cousin is general wrt gender.
        - "She is visiting her cousin" is general
    - Vagueness:
        - Fast isn't precise. In fact, it's vague.
- **Language Error (NEW!)**
    - Basically, don't use the words "only" or "all" when writing shit, and you won't go wrong.
    - In fact, don't use:
        - Almost, also, even, hardly, just, merely, mostly, nearly,

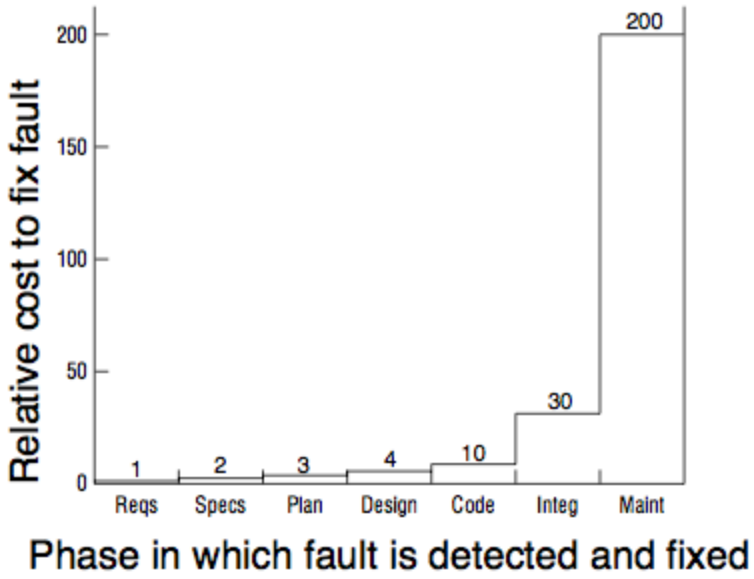
really, only, all, also, this
- Don’t rely on domain knowledge to infer context in plural quantities:
  - Bad:
    - Students enroll in six courses per term
    - Students enroll in hundreds of courses per term
  - Better:
    - Each student enrolls in six courses per term
    - The student body enrolls in hundreds of courses per term

- [30-31] October
  - Non-Functional Requirements (NFRs) https://www.student.cs.uwaterloo.ca/~se463/NFRs.pdf
    - NFRs are attributes and characteristics of the system.
    - **Functional Requirements** describe what the system is supposed to do.
    - **Non-Functional Requirements** describe extra constraints on how the SUD should satisfy the functional requirements.
    - NFRs strongly affect how a product is experienced by the user.
    - Generally expressed for these qualities:
      - Performance
      - Usability
      - Reliability
      - Security
      - Robustness
      - Scalabilitny
      - Adaptability
      - Efficiency
      - Accuracy/precision
    - We also have these other types of NFRs:
      - Process requirements
        - Resources
        - Documentation
        - Complexity
        - Standards
      - Design constraints
        - Interfaces
        - Programming Language
      - Product-family requirements
        - Modifyiability
        - Portability
        - Reusability
        - UI
      - Operating Constraints
        - Location, size, power consumption, temp, humidity, etc.

        - **Motherhood Requirements** are the requirements that nobody would not ask for -- “reliable” “User friendly”, “maintainable”, etc.
            - We establish a degree that the these are important.
            - We can measure these through unambiguous fitness criterion.
                - “Computation errors should be fixed within 3 weeks of being reported”
        - These metrics are measurable:
            - Performance
            - Efficiency
            - Reliability
            - Security
            - Robustness
            - Scalability
            - Cost
            - Portability
            - Maintainability
            - Usability
        - We can use a monte carlo technique to estimate an unknown quantity using a known one:
            - i.e. the number of bugs remaining in a program:
            - Plant a known number of errors into the program, which the testing team doesn’t know about.
            - Find the seeded errors the team detects with the number of total errors it detects.
                - detected seeded errors/seeded errors = detected errors/errors in program
        - Quality-Function Deployment
            - QFD is a way to relate an unmeasurable or hard-to-measure NFR to functional requirements
            - He didn’t go into details in the slides, so I’m not going into it here.
            - #YOLO
        - Prioritizing NFRs
            - Usually, there are tradeoffs between Cost, Schedule, and Features.
            - Get people to fill in a quality grid that enumerates all NFRs by necessity, so they agree on what’s more important.
- [4-5] Nov
    - Cost estimation https://www.student.cs.uwaterloo.ca/~se463/CostEstimation.pdf
    - It’s hard to estimate the time it’ll take to write code.
        - Measurements make sense only when you have stuff to compare it to.
    - We want to do this, because we want to commit to time specifications, roughly track progress, and provide a basis for agreeing to a job.
    - **Delphi Method**

        - Each expert submits a secret prediction
        - The average estimate is sent to all experts
        - Repeat until no expert wants to revise their estimate.
    - It's hard to measure by lines of code, since not all lines are equal.
    - We want to estimate cost based on what we know at requirements time.
        - Number of function points from the requirements
            - Apply a "complexity" heuristic to a function, based on the number of references, the number of inputs and outputs.
        - Estimating code size from function points.
            - There are tables that specify the number of lines per FP for each language.
            - These are specific to each shop, domain, etc.
        - Estimating resources required (time, personnel, money) from code size.
            - Use COCOMO
                - Effort in person-months $= a\,X\,(KLOC)^b$
                - Development time in months $= c\,E^d$
                - Where
                    - a, b, c, d are empirically found
                    - X is defined from project attribute multipliers
                        - (memory constraints, complexity, etc)
                    - KLOC is the number of lines of thousands of lines of code.

- [6-7 Nov]
    - Requirements Determination is Unstoppable https://www.student.cs.uwaterloo.ca/~se463/RDisUnstoppable.pdf
    - RD may or may not be part of any conscious RE process.
        - WTF
        - Let's be tested on this bs
        - /fucksgiven
        - /ragequit
    - **RD during coding** is where programmers and testers determine requirements as they write code and test cases
    - **Upfront RE** is where as much time as necessary goes into requirements before proceeding with design/impl
    - We need to meet somewhere in the middle
        - With less upfront RE, RS is incomplete, which prevents programmers from continuing before they decide what the missing requirements are.
        - While programmers should go to the client, they just choose the simplest thing to implement.
    - The sooner that we find modifications that need to be made, the cheaper it is to fix. This is an exponential growth.



- Doing only upfront RE has possibility of not halting.
- We can prevent that by:
  - RE for a scope is done when the RS is complete enough that every programmer can program without having to ask anyone to clarify a requirement, and without having to invent any requirements on the spot.
  - Same goes with testers.
  - Since programmers and testers need to be part of this, we should put at least one of each on the RS writing team.
- **PotLoBoaDtiP** happens when docs aren't kept up to date, or those who have knowledge to write a document aren't the ones who benefit from the document.
  - In the best case, there's much drudgery for all when writing these docs.
  - In the worst case, the writer is rebuked for not having a spec, and the beneficiaries are implementing a spec that doesn't exist.
  - PotLoBoaDtiP can be solved with incentives

- [11-14] Nov - Temporal Logic - https://www.student.cs.uwaterloo.ca/~se463/JoAtleeTempLogic.pdf
  - Descriptive spec notations allows one to describe constraints on the behaviour of a system.

-- Alex’s notes start here - format them if you want --

- - OCL: Constraints on all world states of domain model.
- - Functions: Constraints on consecutive pairs of world states during execution.
- - Temporal Constraints: Constraints on object models, states, events, variables over execution traces.
- - State Formula (s |= p): Property p evaluated wrt particular execution state s
- - Execution trance: σ = s_0, s_1, ...
- - Explicit Time: All functions take time parameter (eg. (forall)t. coin(t) => not locked(t + 1))
- - Implicit Time: Use connectives for always, never, eventually, etc.
- - LTL: Linear Temporal Logic:
- - (σ, j) |= f iff f is true in state s_j of σ
- iff s_j, s_{j+1}, ... |= f (All previous states and state j entail f is true)
- - σ |= f is same as (σ, 0) |= f (True in initial state)
- - Henceforth (box): True if predicate is true in current and all future states.
- - Eventually (diamond): True if predicate is true in current or some future state.
- - Next State (circle): True if predicate is true in the next state.
- - Until (cursive U): f UNTIL g is true if g is eventually true and f is true until g is true.
- - Unless (cursive W): f UNLESS g is true if f holds indefinitely OR f holds until g holds.
- - Tautologies:
- - (NEXT p) => EVENTUALLY p
- - (f UNTIL g) => EVENTUALLY g
- - g => (f UNTIL g)
- - (f AND (NEXT g)) => (f UNTIL g)
- - Patterns:
- - Initially p: p
- - Globally p: HENCEFORTH p
- - p after Q: HENCEFORTH (Q => HENCEFORTH p)
- - p before first R: p UNLESS R
- - p between Q and R: HENCEFORTH (Q => (p UNTIL R))
- - p after Q unless R: HENCEFORTH (Q => (p UNLESS R))
- Mutual exclusion of a, b:
- - Never a and b: HENCEFORTH NOT (a AND b)
- - Not before first R: NOT (a AND b) UNLESS R
- - Not after Q: HENCEFORTH (Q => HENCEFORTH NOT (a AND b))
- - Not between Q and R: HENCEFORTH (Q => (NOT (a AND b) UNLESS R))
-
- - Response: b is response to a: (a => EVENTUALLY b) in various scopes.
- - Precedence: a becomes true before b does: (NOT b UNLESS a) in various scopes.
-

Software Verification and Validation

- Validation: Are we building the right system?
    - Spec, Domain |= Reqs
    - Testing can be used to show the presence of errors, but cannot guarantee the absence of errors.

- Model Checking: Reachability, Mutual Exclusion, Deadlock, Temporal Logic Formulas.
- Manual reviews work! Need both domain experts and software engineers.
- Walkthrough: Informal group meeting - stepping through specifications. Meant to find problems with RS.
- Formal Inspection: Very detailed, meant to improve RS.
  - Fagan Inspections: Iterate using 2h inspection process.
- Focussed Inspection: Each reviewer has different role.
- Active Review: Author poses questions to reviewer - must read document to answer.

- Verification: Are we building the system right?
- Design |= Spec
- Code |= Design
- Test Cases |= Spec
- Code Inspection/Review: Find causes of errors, not just symptoms of errors. Can enforce coding standards.

Problems with Models (Icebergs or something)

- Waterfall Model leaves no room for changing requirements.
- Spiral Model: Develop, verify next-level product -> Simulations, models, benchmarks -> Risk analysis -> Evaluate alternatives; identify and resolve risks -> Determine objectives, alternatives, constraints -> Plan next phase -> (Back to beginning but on smaller, quicker spiral)
- Around 75%-85% of errors can be traced back to requirements and design phases. [Barry Boehm]
- E-type System: Solves problem or implements an application in some real-world domain.
- No avoiding RS unless you do no testing or user documentation - so better do it up-front to avoid costly errors after development begins.