Objectives

- Want to identify and articulate in a way that avoid implementation bias
  - Requirements - Conditions and capabilities that describe a problem – to be met by a solution, for the solution to be acceptable
  - Specification – A complete, precise, verifiable expression of requirements of a software or system solution.

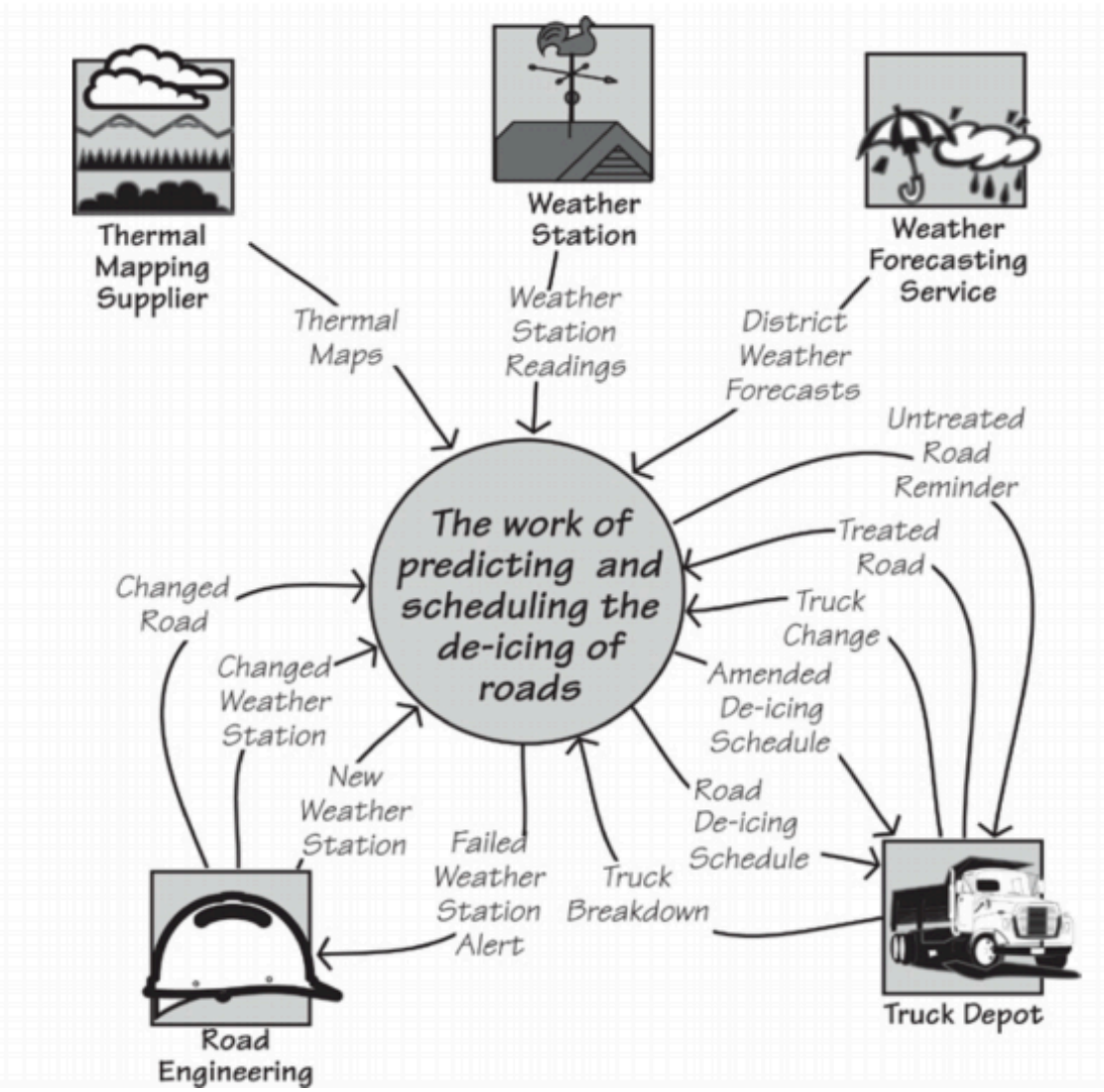
Specification

- A specification is a description of the proposed system
  - desired changes to the world
  - requirements re-expressed in terms of interface phenomena
  - places no constraints on the design or implementation of the system giving the designer maximum freedom
  - Intersection of the environment and the system under test (SUD)
- i.e. Facebook
  - Env = connect people, make money
  - SUD = ads in the sidebar, sponsored posts, account data
  - Intersection = accounts, walls, timeline, photos, ads
- i.e. Park User Fees
  - Suppose that the city of Waterloo decides to raise funds by instituting users fees for public parks
  - Requirements:
    - R1: Collect $1 fee from each user on entry to the park
    - R2: Ensure that anyone who has paid may enter the park
    - R3: Ensure that no one may enter the park without paying
  - Shared Phenomena
    - Wall, Hole in the Wall, Coin Slot, Gate
  - Specifications
    - S1(R1): Detect coin inserts in advance of park entry
- i.e. Thermostat
  - R: Want to keep the air temperature at or above the set temperature
  - SUD
    - sensed temperature (actual temperature)
    - input of desired temperature (set temp)
    - furnace
  - S1: sensed temp $\geq$ input set temp = turn furnace off, sensed temp $<$ input set temp = turn furnace on
  - A: sensed temperature $\approx$ actual temp (sensed temp sensed freq enough to reflect actual temp)
  - A: furnace works — is capable of bringing the room up to the desired temp
- i.e. Traffic Light
  - S1: Only one direction sees green light
  - S2: Some direction sees green/yellow
  - S3: Directions that green/yellow alternate
  - S4: Yellow light precedes red, long enough for drivers to stop
  - A: Drivers will stop at yellow/red lights go at green lights (the system can't enforce everything, some responsibility is still on the user)

Scoping

- **Purpose of the project:** a rationale for why the project is wanted, what benefit it brings to the business

- **Goal:** Some high-level measurable criterion of success
- **Scope of the work:** The business area affected by the installation of the system. You need to understand this work to specify the most appropriate project
- **The stakeholders:** The people with an interest in the system
- **Constraints:** Restrictions on the scope or style of the system
  - predetermined design solutions that must be used
  - constraints on changing current business processes
  - time and money that area available for the project
- Context Diagram
  - A graphical model of the **context** (environment) in which the Work exists
  - Phenomena that are relevant to the descriptions of the requirements
  - Modularizes the phenomena into domains
  - Precursor to domain models
  -

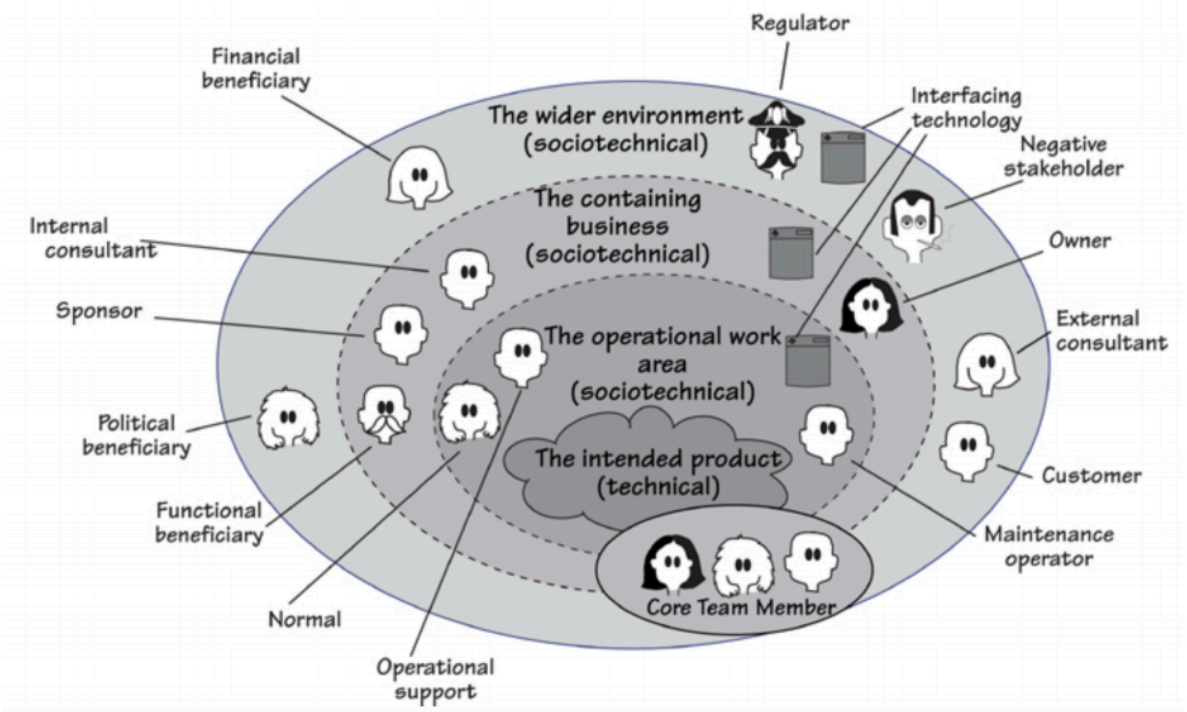


- Volere Req. Spec. Template
  - Project drivers
    - The Purpose of the Project
    - The Stakeholders
  - Project Constraints

    * Mandated Constraints
    * Naming Conventions and Terminology
    * Relevant facts and Assumptions

Stakeholders

- Anyone who has a stake in the ultimate success of the project

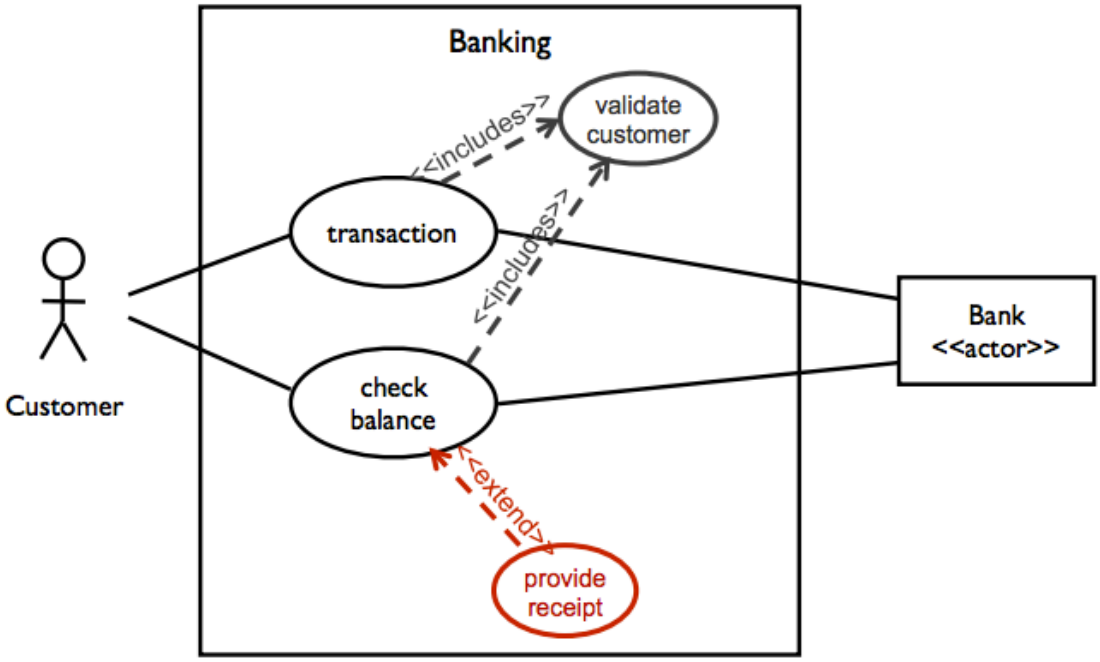


- Owner/Client
  - The person paying for the software to be developed
  - Ultimate stakeholder, always has the last say
- Customer
  - A person who buys software after it is developed
  - Possibly a manager
  - Must be an active participant in the project (could be active through a proxy like a product manager)
- Users
  - May be simulated by a *persona*
  - Experts of the existing system — tell us which features to keep and which need improvements
  - Experts on competitors' products — give suggestions about how to build a superior product
- Domain Expert
  - Know the problem domain well, familiar with the problem that the software must solve
  - Are familiar with typical users and their expectations
  - Are familiar with typical deployment environments
- Software Engineer (technology expert)
  - May include managers too
  - Represents the rest of the development team (devs, testers)
  - Ensures that the project is technically and economically feasible
  - Accurately estimates the cost and development time of the product
  - Educates the customer about innovative hardware or software technologies
- Other

  - Inspectors (experts on government and safety regulations)
  - Market researchers
    * May assume the role of the client if the software is being developed for the mass market and there is no identifiable customer
  - Lawyers
  - Industry Standards
  - Experts on adjacent systems
    * They know about the interfaces to adjacent systems, and any special demands for interacting with the adjacent system
  - Negative stakeholder
    * It is best to understand why they do not want the project to succeed
    * They may simply have competing requirements that you'd be best off considering

Use Cases

- Understanding the work
  - Goal is to understand what the previous work is, not how it is implemented
  - abstract inputs and outputs
- Business use cases decompose the work into work-pieces to manage complexity
  - Vertical slices
  - Represents some end-to-end functionality
  - Triggered by an external event
  - captures a complete response to a triggering event
  - Ideally orthogonal to each other
- Use Case Models = express business use cases in a manner that is easy for all stakeholders to visualize — big picture view of the problem
- Actor Generalization
  - Use actor generalization when actors have common *interesting* behaviour
  - Children inherit all relationships with use cases of the parent
- Special relationships
  - `<<include>>` is a sub use case that is used within multiple other use cases
  - `<<extend>>` is a sub use case that extends or replaces the end of an existing use case



–

Modelling and Documentation

Domain Models

- Requirements Models
  - Uses
    - Can guide elicitation
    - Provide a measure of progress
    - Uncover problems, especially omissions
    - Check our understanding
  - Requirements are expressed in terms of environmental phenomena
  - A **domain model** visualizes the elements of the environment that are relevant to the Work
  - Conceptual, focus on real-world entities, not system entities
  - When creating objects look for noun phrases

Lightweight Models

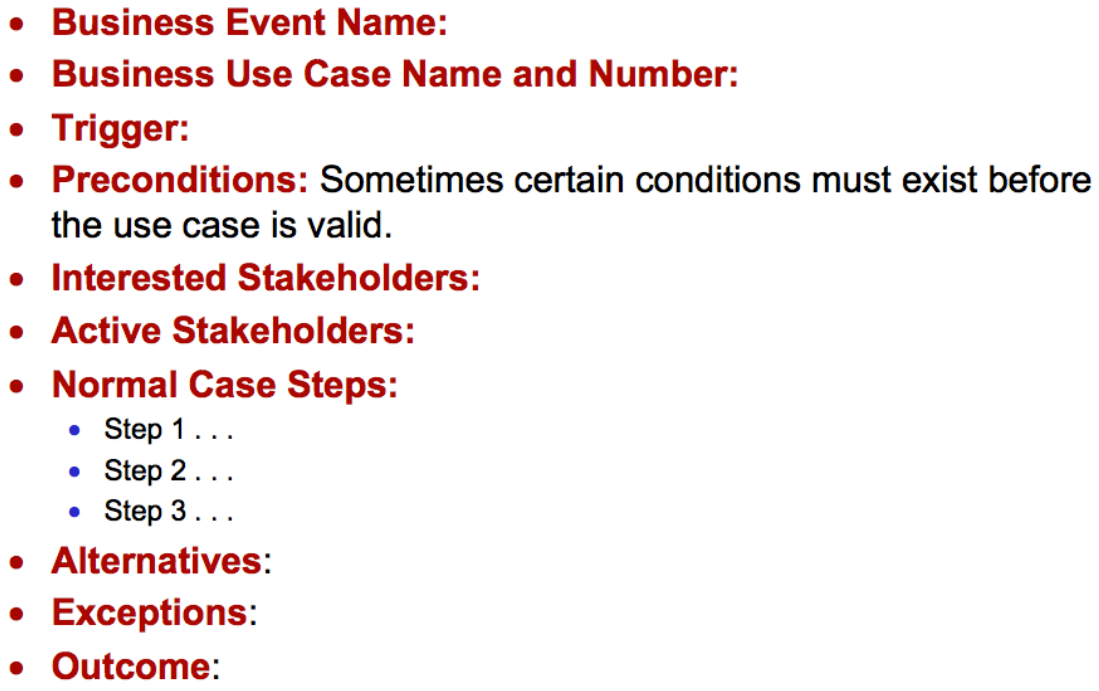
- Mind maps
- Scenarios
  - One full execution path through a use case, listing only observable actions
  - i.e. Banking scenario
    1. User requests to withdraw funds, specifies amount
    2. Bank authenticates user
    3. Bank checks that the account has sufficient funds
    4. Bank dispenses cash and receipt
    5. Bank records the transaction
    6. User takes cash and receipt and leaves

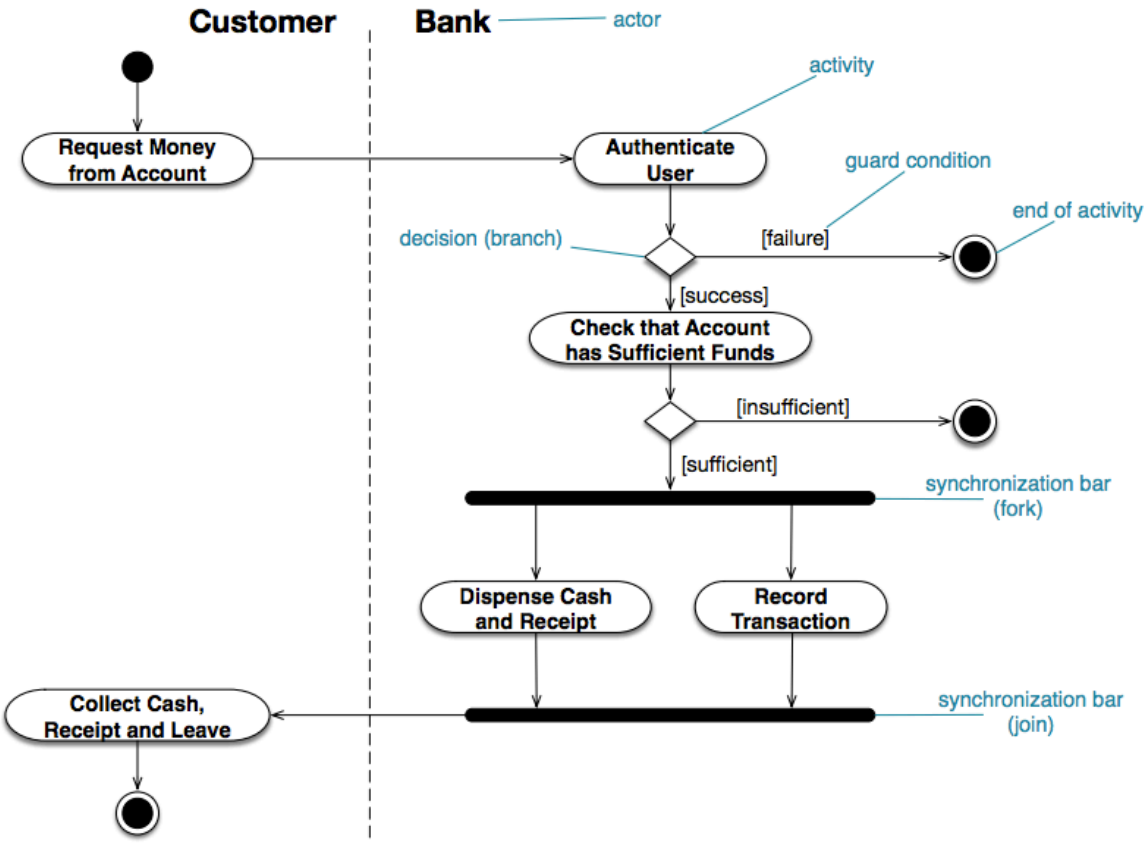
- **Alternative**: a sub use case that achieves the main goal of a use case through a different *intended* sequence of actions
  * Ex:

  1. User requests to withdraw funds, specifies amount
     A1.1 User cancels transaction
  2. User takes cash and receipt and leaves
     A6.1 User takes cash and receipt, initiates a new transaction
- **Exception**: a sub use case that captures an *unwanted* but inevitable deviation
  * Ex:

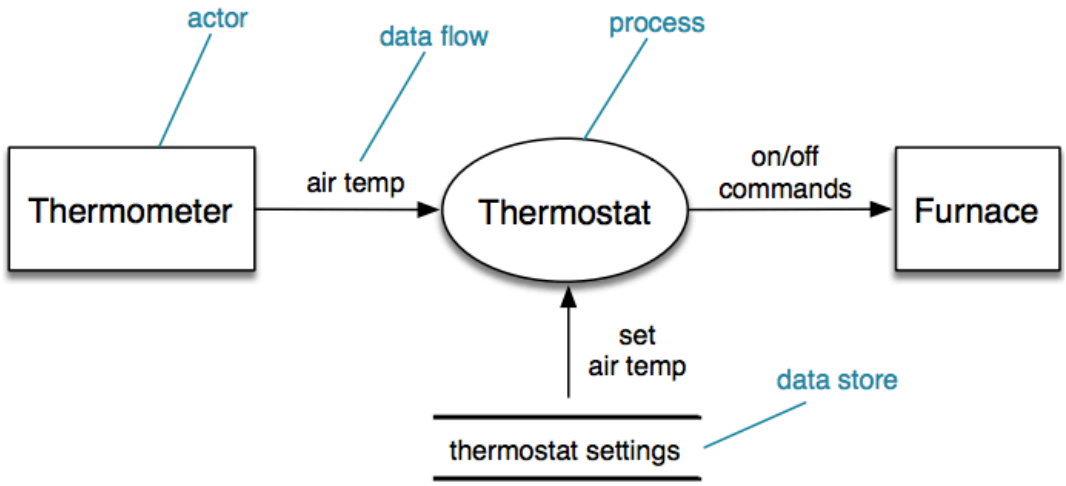
  3. Bank checks that account has sufficient funds
     E3.1 Bank cancels withdrawal due to insufficient funds
- Negative scenario: a scenario that is prohibited by the system
  * i.e. Banking scenario
    1. User requests to withdraw funds, specifies amount
    2. Bank authenticates user
    3. Bank checks that the account has insufficient funds
    4. Bank dispenses cash and receipt
- Misuse case: a scenario that captures undesirable inputs from the user or environment
  * Ex

  2. Bank authenticates user
     M2.1 User is an impostor
- Use case vs. Scenario: A **use case** is a collection of *success* and *failure* **scenarios** initiated by an external actor, to achieve a particular goal
- Template





- Activity Diagram
- Process model
  - A functional decomposition of the work and the data dependencies between function

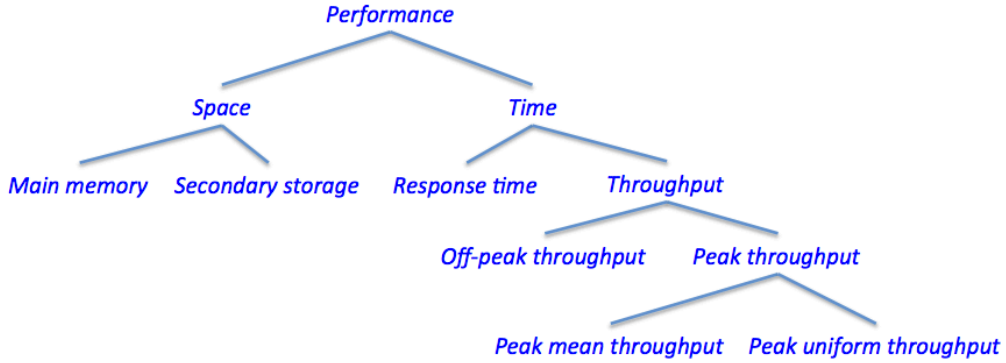


  -

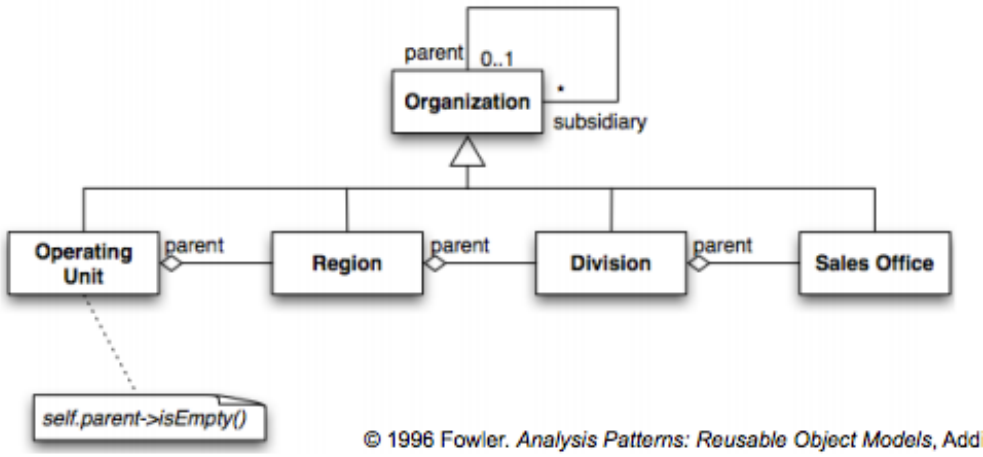
Elicitation

- Means to "to bring out, to evoke"
- Purpose is to learn the requirements of the system by
  - investigating the current work and current goals

  - asking about requirements and goals
  - understanding the environment in which the system will operate
- Types of information to elicit
  - Required functionality — what the software should do, i.e. record keeping, data computations, process control, etc.
  - Quality attributes — desired characteristics of the software, i.e. performance, efficiency, safety, security, etc.
  - Design constraints — customer-specified limits on solution space, i.e. mandated hardware components, resource constraints, mandated adjacent systems, etc.
  - Environmental assumptions — assumed context of the software, i.e. working status of hardware, inputs, operating conditions
  - Preferences — priority rankings of requirements
- Techniques
  - Artifact-based
    * idea: learn as much as we can by studying documentation, systems, artifacts, etc. before asking for stakeholder's time
    * i.e. document studies, similar companies, domain analysis, etc.
    * norms — "Build a better X"
    * requirements taxonomy — classification of requirements; the classification can act as a checklist of details to be elicited
      · i.e. domain-independent taxonomy for performance-related NFRs

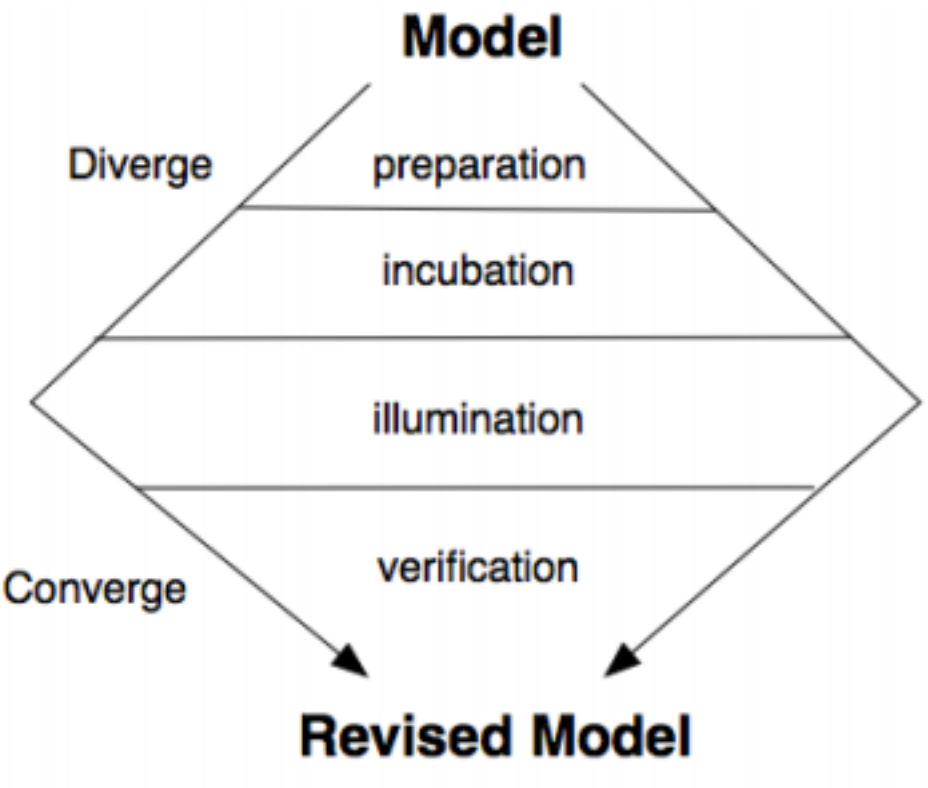


  - Model-based
    * idea: to re-express the requirements in a different language, which can raise new questions
      · i.e. modelling, analysis patterns, mockups/prototyping, pilot experiments
    * Effective models restrict the amount of detail we include in our models of the current system
    * Analysis patterns
      · Templates for modelling common business problems
      · i.e. pattern for organizational relationships



    * Mockups and Prototypes
        · Sketch the essence of a solution, and use to bait stakeholders into providing new requirements details

– Stakeholder-based elicitation
    * idea: acquire detailed information about the system-to-be that is problem specific or stakeholder specific
    * i.e. stakeholder analysis, questionnaires, interviews, observation, etc
    * goals of analysis
        · what is used, what isn't, and what's missing
        · what works well, what doesn't
        · how the system is used, how it was intended to be used, what new ways we want it to be used
    * closed questions gather opinions
    * open questions (who, what, when, where, why) gather suggestions
    * interview guidelines
        · start with stakeholder background, goals
        · follow with use-case specific questions
        · feedback your understanding of stakeholder's answers
        · Use the stakeholder's terminology
        · write down everything you are told
    * ethnographic analysis
        · a direct, first-hand observation of user behaviour
        · attempt to discover the social/human factors in a system
        · identify the used and critical existing features
        · focuses on existing solutions
    * apprenticeships: the apprentice sits with the master craftsman (the user) to learn the job
    * personas: useful when real users are not available or are too numerous to interview them all
– creativity-based
    * idea: to invent undreamed-of requirements that bring about innovative change and gives competitive advantage
    * i.e. systemic thinking, brainstorming, creativity workshop
    * brainstorming, two part
        · part 1 – ideation generation
        · part 2 – assessment
    * creativity workshop



Functional Requirements

- What the product has to do to support and enable the Work
- Should be independent of the technology used by the eventual product
- Atomic (functional) requirements are derived from scenarios or activity diagrams
  - What the product has to do to achieve each step/activity
  - Expressed using vocabulary from the domain model
- Written as a single sentence with a single verb
  - i.e. JobMine: "Product will require a ranking for each job where the student user has been ranked by the employer"
- Do *not* use a mixture of modal verbs (shall, must, might, could) to indicate the priority of a requirement
- Include atomic requirements that are derived from scenario alternatives and exceptions
  - These requirements are conditional on the event/condition that leads to the alternative or exception
  - i.e. The product will display the set of available interview slots and allows the student user to select one of them.
    * If the student user neglects to select an interview time, then two days before the interview date the product will automatically schedule the student into one of the remaining interview slots.
- Rationale
  - Each atomic requirement should be accompanied by a rationale that explain why the requirement exists
- EARS template
  - Easy Approach to Requirements Syntax
  - A template for structuring the syntax of the one-sentence description of an atomic requirement
  - Acknowledges that the vast majority of requirements will be expressed in natural language
  - Promotes a gently constrained use of natural language

  - Generic syntax = `<optional precondition> <optional trigger> the <system naem> shall <system response>`
  - Specializations
    * A **ubiquitous** requirement is an invariant behaviour of the system (that should always hold) = `The <system name> shall <system response>`
    * An **event-driven** requirement applies when a triggering event is detected at the system boundary = `WHEN <trigger> the <system name> shall <system response>`
    * A **state-driven** requirement applies as long as the system is in a particular state, or mode of operation = `WHILE <in a specific state> the <system name> shall <system response>`
    * An **option** requirement when system behaviour is dependent on the presence of a particular feature = `WHERE <feature is included> the <system name> shall <system response>`
    * Special syntax distinguishes system responses to **unwanted events** = `IF <trigger>, THEN the <system name> shall <system response>`
- User Stories
  - A light-weight approach to managing requirements
  - Short statement of new functionality or feature
  - Written from the point of view of the user
  - Its details are fleshed out *just in time* before the story is placed in the sprint
  - Three C's
    * **Card** – stories are traditionally written on note cards, using a structured syntax `As a <role> I want <something>, so that <benefit is achieved>`
    * **Conversation** – discussions with the product owner reveal details of the requirements
    * **Confirmations** – acceptance criteria for objectively determining whether an implementation meets the requirements
  - Easy for stakeholder to understand, and to remember
  - Shift the focus from written documentation to discussion
  - Encourage iterative development, with stories being appropriate sized increments fro planning
  - Delay the elicitation of requirement details until just before development
  - Support participatory elicitation
- Graphical function Model
  - Expresses a system-level function as a rule that **matches and replaces** patterns in the world state

  -

Conflict

- **Data conflict**: multiple conflicting understandings of an issue
  - i.e. student information system should track attendance. But, there is no room to record attendance on a transcript
- **Interest conflict**: stakeholders have subjectively or objectively different goals or interests
  - i.e. Province demands that student information system track attendance, for accounting purposes. But, student insist that attendance not be recorded, to protect their privacy
- **Value conflict**: stakeholders express different preference

  - i.e. Engineering Faculty wants to rank students in courses. But, students think that rankings are too fine a distinction between student's performances

Priorities

- There are often more requirements than can be implemented
  - Customers ask for way too much
  - Need to balance requirements against limitations in budget, staff, schedule
  - Need to decide which features go into the next release
- Requirements triage
  - Some requirements must be implemented
  - Some requirements may be obvious choices to exclude
  - The rest are desirables that we need to select from
- Prioritization Aspects
  - Business Value Added
  - Penalty/Harm Avoided
  - Risk
  - Cost
  - Time
- Combine aspects: Most companies want to prioritize requirements by their potential value and cost
  - **Value** is a requirement's potential contribution to customer satisfaction
  - **Cost** is the cost of implementing the requirement
  - Can prioritize requirements according to their cost-value ratios
- Groupings
  - Most common prioritization technique is numerical assignment or grouping of requirements into 3-4 groups
    * Critical
    * Standard
    * Optional
- Ranking
  - Each requirement is assigned a unique rank (1, 2, …) but it is not possible to see the relative difference between ranked reqs
- Kano Model
  - A method fro grouping reqs based on customer perception, in order to select the reqs that deliver the greatest customer satisfaction
    * Basic: reqs that the customer takes for granted
    * Performance: reqs that the customer specifically asked for
    * Excitement: reqs that the customer does not request or expect
    * Indifferent: reqs that the customer does not care about
    * Reverse: "requirements" that the customer hates
  - Survey
    * Ask customer what their reaction would be if the req were included in the product
    * Ask customer what their reaction would be if the req were NOT included in the product
    * Classic scale, 1 = Like, 3 = Neutral, 5 = Dislike

Customer Survey Responses		Disfunctional Question Answer				
		Like	Expect	Neutral	Tolerate	Dislike
Functional Question Answer	Like	*Questionable*	Excitement	Excitement	Excitement	Performance
	Expect	Reverse	*Questionable*	Indifferent	Indifferent	Basic
	Neutral	Reverse	Indifferent	Indifferent	Indifferent	Basic
	Tolerate	Reverse	Indifferent	Indifferent	*Questionable*	Basic
	Dislike	Reverse	Reverse	Reverse	Reverse	*Questionable*

–

- 100-dollar test
  - Cumulative voting
  - Customer is given 100 prioritization points to distribute among the reqs
- Analytical Hierarchy Process
  - Technique that analyzes stakeholders' pairwise comparisons of reqs, and produces a relative ranking of reqs
  - Acknowledges that absolute values and costs are hard to estimate
  - Values
    - 1 = Equal value
    - 3 = One is slightly preferred
    - 5 = Strongly preferred
    - 7 = Very strongly preferred
    - 9 = Extremely strongly preferred
  - If pair (x,y) has rank n then (y,x) has 1/n
  - Process
    - Build table
    - Normalize columns by dividing each entry by the sum of the column
    - Sum each row
    - Normalize sums
    - Report relative values
  - Consistency
    - Multiply comparison matrix by priority vector (result of last point)
    - Divide each element by the corresponding element in the priority vector
    - Compute principle eigenvalue (sum results and divide by number of reqs)
    - CI = (eigenvalue - n) / (n - 1)
- Challenges
  - All reqs deemed to be essential
  - Large number of reqs to prioritize
  - Conflicting priorities
  - Changing priorities
  - Stakeholder and developer collaboration
  - Subjective prioritization
- Benefits
  - Improves custom satisfaction, by implementing most important reqs first
  - Helps to determine how to prioritize the allocation of limited project resources
  - Encourages stakeholders to consider all reqs (not just their own)

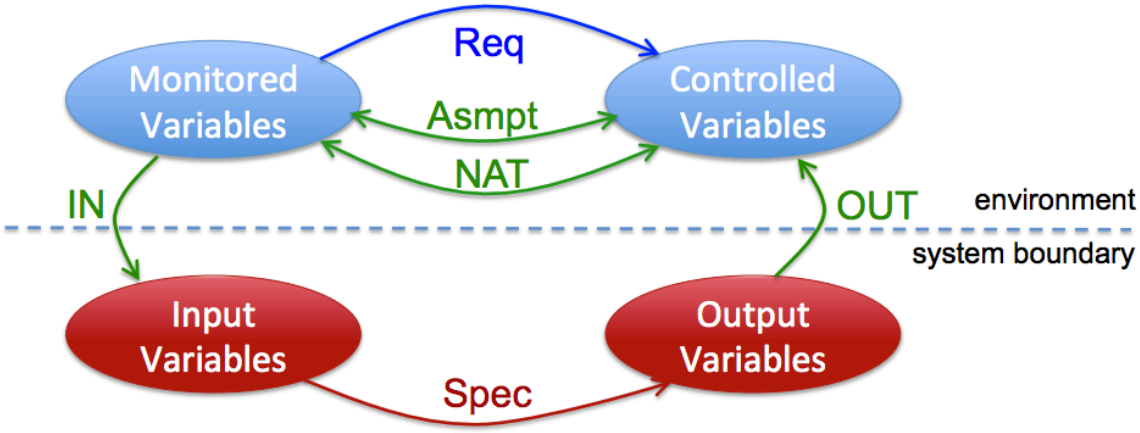
Business Rules and OCL

- **Business rule**: an assertion that defines or constrains some aspect of the work
- ex:
  - rental agreements must be no longer than 4 weeks long
  - A customer must be at least 25 years old
  - A customer cannot rent more than 3 cars
- OCL = Object Constraint Language
  - Complements UML
- Constraints Expressions are expressions over
  - Attributes
  - Navigations derived from associations (and aggregations)
  - Role name on far end of association
  - Class name on far end of association
  - Literal values
- ex.
  - Rental agreement must not exceed 4 weeks
    ```
    context r:RentalAgreement inv:
    self.end - self.start < 28
    ```
  - A person cannot rent more than 3 cars
    ```
    context Person inv:
    self.RentalAgreement.rental.car->size() <= 3
    ```
  - Rental car companies never own red cars
    ```
    context RentalCarCompany inv:
    self.owns.color->excludes(red)
    ```
  - Red cars are never rented for less than $100/day
    ```
    context Vehicle inv:
    self.color == red implies
    self.RentalAgreement->forall(r.RentalAgreement | r.price / (r.end - r.start) > 100)
    ```
- Syntax
  - `->` = operations collections (as opposed to methods on class which use .)
  - `allInstances()` = the set of all instances of that class
  - `<>` = not equals
- Set operations
  - size, sum, isEmpty, notEmpty
  - functions that take an instance = count, includes, excludes
  - "mutation"
    - insert = `including`
    - remove = `excluding`
  - usual set operations like union, intersection
  - Filters = `select(pred)`, `reject(pred)`
  - Assertions, i.e. `exists(pred)` = some/at least one
  - Quantification = `forAll`

Specifications

- Objectives
  - Want to articular a software specification that meets the stakeholders' reqs
- Complete, precise, **verifiable** expression of reqs of a software or system solution
- Software boundary is identified
  - With respect to what reqs it will implement
  - Input data

  - Output data or commands
- Re-expresses reqs in terms of interface phenomena
- Places no constraints on the design or implementation of the system giving the designer maximum freedom



- $Spec \wedge Dom \models Req$
  $Dom = Asmpt \wedge NAT \wedge IN \wedge OUT$

Quality Requirements

- Functional reqs describe what the software is supposed to do
  - What services or tasks the software should provide
  - Black box input/output behaviour
- Quality reqs describe (extra) constraints on what constitutes an acceptable software solution
  - e.g. How fast should the system respond, which deployment platforms should be supported
- categories
  - Performance
  - Reliability
  - Robustness
  - Adaptability
  - etc
- Other non-functional reqs
  - Design constraints
  - Operating constraints
  - Product-family reqs
  - Resources
  - Documentation
  - etc
- "Motherhood" reqs
  - Expressions such as "reliable", "user-friendly", and "maintainable"
  - Nobody would explicitly ask their opposite
  - Discuss to what degree each attribute is required, relative importance between them
- Fit criteria
  - Quantifies the extent to which a quality req must be met
  - i.e. 75% of users shall judge the system to be as usable as the existing system, after training 90% of users shall be able to process a new account within 4 minutes
  - Levels: Outstanding, Target, Minimum
  - Problem: can't test before delivery. Possible approaches

        * Measure the attributes of a prototype
        * Measure secondary indicators
        * Estimate
        * Monte Carlo technique
            · i.e. plant x bugs intentionally, the percentage of them that aren't caught indicates how many unintentional bugs there are
- Prioritization
    - Many reqs conflict with one another, i.e. performance vs. reuse

Risk Management

- An uncertain factor whose occurrence may result in some loss of satisfaction of some corresponding objective
    - has a **likelihood** to occur
    - has **consequences**
    - product-related or process-related
- Trying to completely eliminate risk from your software project is unrealistic and can be prohibitively expensive
- Risk Management attempts to manage the degree to which a project is exposed to risk of quality, delay or failure
- Defect Detection and Prevention (DDP)
    - DDP is a process developed by NASA for systematic risk assessment and mitigation
    - Process
        * Identify most critical requirements
        * Identify potential risks
        * Estimate the impact of each risk on each requirement
        * Identify possible countermeasures
        * Identify the most effective countermeasures
    - Result is an optimized collection of mitigating actions that may be applied to the project
- Risk Consequence Table
    - Used to identify critical requirements/potential risks and to estimate impact
    - Goal is to develop a prioritized set of risks to be addressed

		Risks (Failure Modes)				
Requirements	**Weight (req)**	**Requirements** (incomplete, incorrect, ambiguous, changing)	**Project Management** (estimations, project, team management)	**Technical** (complex problem, lack of experience with technology)	**Dependencies** (on adjacent systems, components, other people)	**Loss of Objective**
Likelihood (risk)		**0.5**	**0.8**	**0.7**	**0.2**	
Creating a product that users would like	**0.7**	0.9	0.3	0.9	0.5	**0.7**
Completing the product on time	**1.0**	0.8	1	0.9	0.3	**1.96**
Risk Criticality		**0.715**	**0.968**	**1.071**	**0.13**	

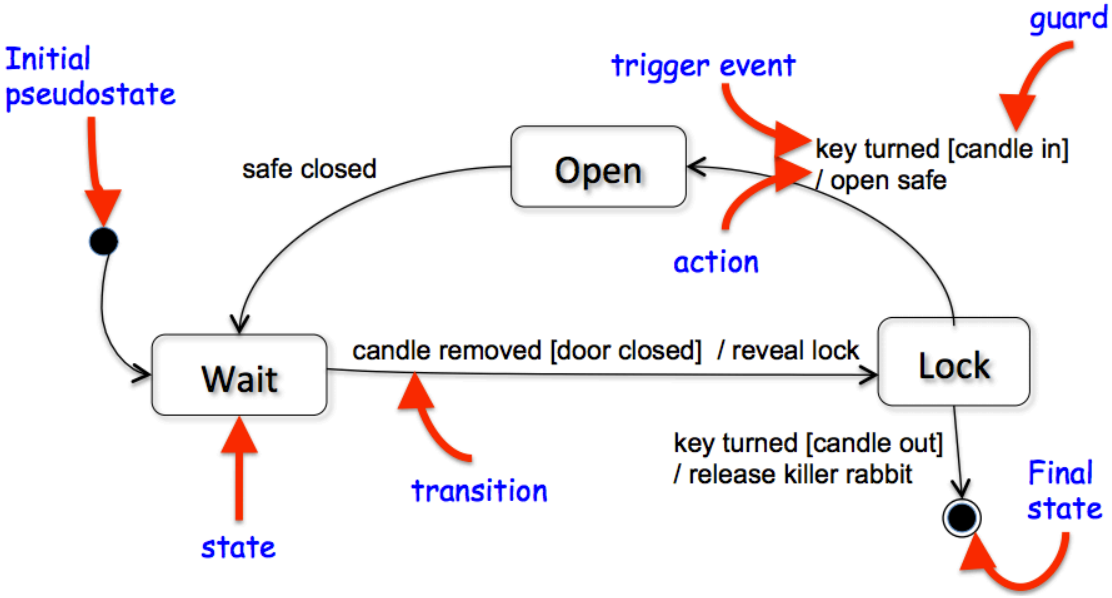
    -
    - $\text{Loss}(\text{req}) = \text{Weight}(\text{req}) \times \sum_{\text{risk}} (\text{Impact}(\text{req, risk}) * \text{Likelihood}(\text{risk}))$
        * Risk-driving reqs are the reqs that are most at risk of not being achieved
    - $\text{Criticality} = \text{Likelihood}(\text{risk}) \times \sum_{\text{req}} (\text{Impact}(\text{req, risk}) * \text{Weight}(\text{req}))$
        * Tall poles are the most critical risks, having the most severe consequences
- Risk Countermeasures Table
    - Identify options for preventing or detecting failure modes
        * i.e. preventative measures, analyses, process controls, tests, mitigations
    - Perhaps to identify the most effective countermeasures

	Risks (Failure Modes)				
Countermeasures	**Requirements** (incomplete, incorrect, ambiguous, changing)	**Project Management** (estimations, project, team management)	**Technical** (complex problem, lack of experience with technology)	**Dependencies** (on adjacent systems, components, other people)	**Overall single effect of countermeasure**
Criticality (risk)	**0.715**	**0.968**	**1.071**	**0.13**	
Collaborative elicitation process with extensive user involvement; modelling; mock-ups	0.5	0.3	0	0.1	**0.6479**
Continually estimate costs; use shorter development iterations	0.25	0.7	0.2	0	**1.09655**
Prototype novel or risky requirements; plan time for learning and experimentation	0.25	0.25	0.5	0	**1.02125**
Investigate suppliers; monitor their progress; develop backup plans	0	0	0.1	0.5	**0.1201**
Combined Risk Reduction	**0.71875**	**0.8425**	**0.64**	**0.55**	

  -
  - $\text{OverallEffect(cm)} = \sum_{\text{risk}} (\text{Reduction(cm, risk)} * \text{Criticality(risk)})$
  - $\text{Combined Reduction(risk)} = 1 - \Pi_{\text{cm}} (1 - \text{Reduction(cm, risk)})$
  - Apply Optimal Countermeasures
    - $\text{MitigatedLikelihood} = \text{Likelihood(risk)} * \Pi (1 - \text{Reduction(cm, risk)})$
- Optimizing the Residual Risk
  - Goal: Select the optimal combinations of countermeasures based on
    - Their joint effectiveness in reducing risk
    - The associated cost of implementation

Behaviour Modelling

- Finite State Machines
  - Models behaviour that depends on the history of inputs received so far



  -

- UML state machines
  - States represent equivalence classes of input traces
  - Inputs are events and conditions on `<<interface>>` phenomena
  - Outputs are actions on `<<interface>>` phenomena
- States
  - Partition the behaviour of the system
  - Each state represent a possible mode of operation, such as
    * Equivalence set of history of inputs
    * Equivalence set of future behaviours
      · Distinct set of input events of interest
      · Distinct reactions to input events
    * Internal processing of input (delimited by transitions)
  - There always needs to be a designated starting/initial state
    * pseudo-state, since no time is spent here
  - Final state is a real state
- Transitions represent **observable** execution steps
  - Parts `event(args)[condition]/action`
    * `event(args)` – input event/message that triggers the transition
    * `[condition]` – (boolean) guard condition; the transition cannot fire unless the guard is true
    * `/action` – a simple, fast, non-interruptible action
  - Events are a significant or noteworthy change in the environment
    * Input message from the environment
    * Some change to `<<interface>>` phenomena
    * Passage of time
    * Multiple events on a transition label are alternatives
  - Conditions
    * Boolean expression
    * Can involve state-machine vars or `<<interface>>` phenomena
  - Actions
    * The system's response to an event
    * output message or change to `<<interface>>` phenomena
    * Non-interruptible (atomic)
    * Multiple separated by a `;` and executed sequentially
- Process for creating a behaviour model
  1. Identify input and output events (look to the domain model)
  2. Think of a natural partitioning into states
     - Activity states – system performs activity or operation
     - Idle states – system waits for input
     - System modes – use different states to distinguish between different reactions to an event
  3. Consider the behaviour of the system for each input at each state
  4. Revise (using hierarchy, concurrency, state events)

State machine vs. sequence diagrams

State machine diagrams	Sequence diagrams
specifies behaviour	*Illustrates* behaviour
all allowable scenarios	one allowable scenario, showing end-to-end behaviour (better feel for overall system behaviour)
developer oriented	customer oriented
identifies system states, which represent equivalent input histories	
	can help developer validate state diagrams

-
- Hierarchical states are used to cluster states that have some similar behaviours/exiting transitions
- History H is a pseudo-states that designates the child state of H's parent state that the execution was in when the parent state was last exited — resume
  - Deep History H^* is a pseudo-state that designates the descendant state of H^*'s parent state that the execution was in when the parent state was last exited
- Concurrent Regions
  - Some system have orthogonal behaviours that are best modelled as concurrent state machines (use a dashed line in the parent state to separate the processes)
  - Execution of a concurrent state is considered finished when all its regions are in their final states
  - Termination state (big X) is used to represent the end of system execution, not like a final state that waits for the other regions to finish
  - Convention `[inState(x)]` to represent a check to see if another region is in state x
- A time event is the occurrence of a specific date/time or the passage of time (`at` or `after`)
- A change event is the event of a condition becoming true — `when(pred)`
  - Different from `[X]` because that condition would be checked once when the state becomes idle
- State actions
  - Written inside the state
  - entry actions: actions that occur every time the state is entered by an explicit transition `entry /action`
  - exit actions: actions that occur every time the state is exited by an explicit transition `exit /action`
  - internal actions: on events `event [condition] /action`
- State activities
  - A computation of the system that takes time and *can* be interrupted
  - `do /`
- Validating behaviour models
  - **Avoid inconsistency**: multiple transitions that leave the same state under the same event/conditions
  - **Ensure completeness**: specify a reaction for every possible input at a state

        * if there are transitions triggered by an event conditioned on some guard, what happens if the guard is false?
    - **Walkthrough**: compare the behaviour of your state diagrams with the use-case scenarios
        * All paths through the scenarios should be paths in the state machines

Mockups

- Sketch the essence of a solution, and use to bait stakeholders into providing new requirement details
- User Experience
    - Encompasses all aspects of a user's interaction with a company, its services, and its products
        * Meet the exact needs of the customer
        * Be simple, elegant, intuitive – a joy to own and use
        * Be a seamless merging of services
    - Not to be confused with usability, which is a quality attributes the concerns the product's ease of use, ease to learn, etc.
- Sketching — Initial drafts of the UI should focus on content: the scope of what to include on a screen and how to lay it out
- Wireframing — A visual representation or mock-up of a user interface, using only simple shapes
    - Focuses on content, info hierarchy and flow
    - Defers details about presentation (colour, fonts, images)
- F-Layout design: Users tend to focus their gaze, reading left to right, and top to bottom, but less to the right the lower they gaze
- Apple's Design Process
    - 10 -> 3 -> 1
    - Paired Design Meetings
    - Pixel Perfect Mockups
- Mockup
    - Sketches the essence of a solution, in terms of screen contents and navigation among screens
    - A paper mockup includes
        * Sketches or wireframes of screen shots
        * Descriptions of each UI widget
        * Navigation diagram that shows how to navigate among the screens
    - State machines can serve as an executable navigation diagram
        * Inputs: Transitions are triggered by textual input, widget selection, or mouse clicks
        * Outputs: UI screen shots are displayed as sate or transition actions
- Annotating Screens
    - Annotate each UI widget (e.g. button, menu options)
    - Describe the purpose of each UI widget
    - Describe the effect of output events on window displays
    - Describe how to navigate among windows
    - GUI-dependent annotations
        * annotate models directly with GUI events
            · events (e.g. button pressed) to trigger behaviour
            · actions (e.g. Display "OK") as system reactions
        * The input events and transition actions in the Electronic Voting example are GUI dependent
    - GUI-independent annotations
        * annotate models with *macro* events and actions, which map indirectly to GUI events
            · declare macros for GUI events (e.g. button pressed)
            · declare macros for GUI actions (e.g. Display "OK")
            · annotate models with macros

Validation and Verification

- 50% of the errors in operations software are requirements errors
- Cheaper to fix if detected early
- **Validation**: are we building the right product?
- **Verification**: are we building the product right?
- Validation goals, we want to ensure that:
  - the specification captures the stakeholders' reqs
  - the customer/user is satisfied that the system as specified will fit their needs
  - the developers understand the specification
- Validation criteria
  - Well-formed
    - unambiguous
    - consistent
    - ranked for importance and stability
    - modifiable (easy to change spec)
  - Valid
    - correct
    - complete
- Reviews
  - Humans *read* and *analyze* artifacts, *look* for problems, *meet* to discuss the problems, and *agree* on a set of actions to address the identified problems
  - Reviews work
    - The find more errors than testing does
    - They find errors faster than testing does
    - Everyone believes in them
  - Number of techniques for reviewing specs
    - Peer reviews
      - Quota on positive and negative comments
      - Signing Off
    - Group Reviews
      - Walkthroughs
      - Focused Reviews
      - Formal Inspections
      - "Active Reviews"
  - Walkthroughs
    - Expert or author presents the spec (scenarios, models, sketchboards, etcs) and the audience review the work from the presentation
    - Advantages
      - no prep work, so reviewers may be more likely to attend
      - may be the only way to get feedback from busy stakeholders
      - if the reqs/specs are presented to a large audience then there is a hope that major oversights will be caught
  - Focussed Reviews
    - The reviewing task is decomposed into subtasks, where reviewers have roles and each looks only for specific types of errors
    - Advantages
      - avoids the problem of reviewers not having the time to read the whole document
      - each reviewer is more effective because he/she can focus on a handful or error types
      - the leader can assign each reviewer to tasks for which he or she is the most skilled

  - Formal Inspections
    * A managed review process
    * Rules concerning participants and roles
    * Strict entry and exit criteria for each step in the process
    * Primary goal is to improve the quality of the SRS
    * Secondary goal is to improve the quality of the development process
    * What can an analysis of detected errors tell us?
      · It can reveal new types of errors that should be added to the checklists to help with future inspections
      · It can point to projects or use cases that are likely to be problematic, because significantly more errors were reported than usual.
      · Evaluation of entry and exit points can help determine whether the project is on schedule
    * NASA requires peer reviews and inspections to be performed on reqs, designs, code, and test plans for project developing the following software classes
      · Human Rated Space Software Systems
      · Non-Human Space Rated Software Systems or Large Scale Aeronautics Vehicles
      · Mission Support Software or Aeronautic Vehicles, or Major Engineering/Research Facility Software
      · Basic Science/Engineering Design and research and Technology Software (if safety-critical)
  - "Active Review"
    * Inspection process where reviewers (who are often outsiders) act as users of our spec artifacts
    * Testing team can design tests based on the spec artifacts
    * Technical writers can write the user's guide or help pages, based on the spec artifacts
  - Advantages
    * More effective than testing (code)
    * Simple, doable, only "costs" time and effort
    * Inspires authors to improve their work
    * Can apply to all sorts of software artifacts
    * Improvement claims are backed up by experiment
  - Disadvantages
    * some find it dull work
    * requires preparation, paperwork
    * Discussion can be bogged down in disagreements or details

- Testing
  - can "run" an executable model and see if its executions match expected behaviour
  - Advantages
    * Customer can see animation of the product executing
    * Checking of low-level details is usually done reliably when done by tools
    * Can be done earlier in the development lifecycle than other testing
  - Disadvantages
    * Can be labour intensive and costly
      · Hand-holding of tools
      · Designing of test cases
    * Many notations are not executable
    * Testing can be used to show the presence of errors, but not their absence

- Prototyping
  - Presentation Prototype
    * used for proof of concept; explaining design features; etc.
    * Explain demonstrate and inform – then throw away
  - Exploratory Prototypes
    * used to determine problems, elicit needs, clarify goals, compare design options

        * Informal, unstructured and thrown away
    - Breadboards or Experimental Prototypes
        * Explore technical feasibility; test suitability of a technology
        * Typically no user/customer involvement
    - Evolutionary (e.g. "operational prototypes")
        * Development seen as continuous process of adapting the system
        * "prototype" is an early deliverable, to be continually improved
- Safety Analysis looks for safety hazards in the system: safety-related failures, their causes, their consequences
    - Software faults: e.g. data sampling rate, data collisions, illegal commands, unsafe modes, etc
    - Failures in Environment: e.g. Broken sensors, memory overwritten, bad input, power fluctuations
- Software Fault Tree Analysis
    - A top-down backward-analysis approach to safety analysis that starts with an undesired software event and determines all the ways it can happen
- Independent V&V
    - Performed by a separate contractor
    - Fulfills the need for an independent technical opinion
    - Cost between 5% and 15% of development costs
    - Studies show up to fivefold return on investment

Estimation

- Why?
    - assess economic feasibility
    - provide a basis for agreeing to a job
    - make commitments that we can meet
    - understand resource needs
- an **estimate** is a prediction of how long a project will take or how much it will cost
- a **target** is a statement of desirable business objectives
- a **commitment** is a promise to deliver
- we must control projects to ensure that they meet the target
    - remove noncritical requirements
    - redefine requirements
    - replacing less-experience staff with more-experienced staff
- estimation error comes from *uncertainty* about how numerous req details and design decisions will be made
    - estimation accuracy improves rapidly for the first 30% of the project
    - sources
        * omitted activities
            · omitted functional/quality reqs: initialization, data conversion, glue code
            · Software development activities: integration, test data, performance tuning, technical review
            · Non-software development activities: vacations, sick days, training, company meetings
        * Optimism
            · Developer estimates tended to contain an optimism factor of 20-30%
        * Bias — wanting estimates to align with targets
        * Subjectivity
- Influences
    - Project size
    - Kind of software

  - Personnel factors
- Data to collect, to compute estimates we need data
  - Project size: lines of code, reqs (function points, scenarios, stories, UI screens)
  - Effort (staff months)
  - Time (calendar months)
- calibration based on project, historical or industry data
  - estimation by analogy
- can use tools to perform complex analysis and simulation
- function point analysis
  - a size metric of functionality
  - goal is to estimate cost based on what we know at requirements time. Break the problem down into three parts
    1. Estimate the number of function points from the requirements
    2. Estimate code size from function points
    3. Estimate resources required (time, personnel, money) from code size
  - number of function points is based on the number and complexity of the following
    * external inputs
    * external outputs
    * external queries
    * internal logical files
    * external interfaces/APIs
- Structured expert judgement: Use expert judgment as a last resort, or for a second/third estimation
  - people doing the work will make the best estimates
  - decompose estimation – and estimate tasks that require no more than 2 days of effort
  - best case to worst case range
  - PERT formula: Expected case = (BestCase + 4*MostLikelyCase + WorstCase) / 6
- Effort
  - informal comparison to past projects (analogous)
  - effort = PastEffort * (size / PastSize)
  - can use ISBSG method to estimate if you don't have historical data
- Schedule
  - Basic Schedule Equation: $\text{ScheduleInMonths} = 3 * \text{StaffMonths}^{\frac{1}{3}}$
    * Multiplier can range from 2 to 4
  - With historical data you can make different estimates
    * Medium to large project (> 50 staff months): $\text{ScheduleInMonths} = \text{PastSchedule} * (\text{EstimatedEffort}/\text{PastEffort})^{\frac{1}{3}}$
    * Small project: $\text{ScheduleInMonths} = \text{PastSchedule} * (\text{EstimatedEffort}/\text{PastEffort})^{\frac{1}{2}}$
- Best practices
  - Decompose the estimation task into multiple subtasks and combine. Can cancel out errors
  - Use multiple techniques
    * Convergence implies good accuracy
    * Spreads implies that you're overlooking something
  - Define the estimation procedure in advance
- Probability Statements
  1. Take the standard deviation of the best and worst case for each use case
  2. Sum the variances and take the square root. The is the combined standard deviation
  3. Use table of standard deviations and percentage confidence to compute the expected case