1. Introduction to Software Engineering: Stakeholders and Tasks

Wednesday, February 4, 2015

What is a stakeholder?

A stakeholder is a person or group of people involved in, or affected by, project activities.
e.g.:

- The project sponsor
- Dev. management and project dev team
- Support Staff
- Customers
- Users
- Suppliers
- Opponents as well as champions of the project.

List of Design & Requirements Tasks (For reference only, don't memorize)

- Architectural Design
    - High-Level model of a thing. Key concept: Abstraction (omit unnecessary detail)
- Database Design
    - Data model design, data structures & relationships, etc.
- Detailed Design
    - Focuses on module view. Often has high level (modules are defined) and detailed design (logic specified for each module, in more or less detail)
- Documentation
    - Facilitates understanding between different people involved and bridge the gap between different phases of development.
    - Also helps you remember what you were thinking of when you wrote something...
- Feasibility Study
    - After an idea is generated, stakeholders verify feasibility.
- Maintenance
    - Activities that aim to keep the app operational/useful after deployment.
        - Corrective Maintenance: error/bug fixing
        - Adaptive maintenance: Adding functionality to cater to changed requirements
        - Perfective maintenance: make software run better/faster/etc.
- Production (Aka Deployment)
    - Installation, data migration, training users, fix any deployment bugs, etc.
- Programming
    - Goal is to implement design in best possible manner using chosen programming languages and suitable tools.
- Project Management
    - Every project must be managed.
- Requirements Engineering
    - Aims to create a complete, consistent (i.e. unambiguous), verifiable,

and as formal as possible expression of user requirements.

- System Integration
    - Software is often part of a bigger system; need to integrate with other parts & test!
- User Interface Design
    - User interface very important.
- Validation
    - Making sure system meets needs of users. "Are we building the right product?"
- Verification (Testing)
    - "Are we building the product right?"

2. Software Dev. Lifecycle Models

Thursday, February 5, 2015 2:00 PM

What is a Software Development Lifecycle?

The processes by which a system is developed. (A sequence of tasks/phases...)

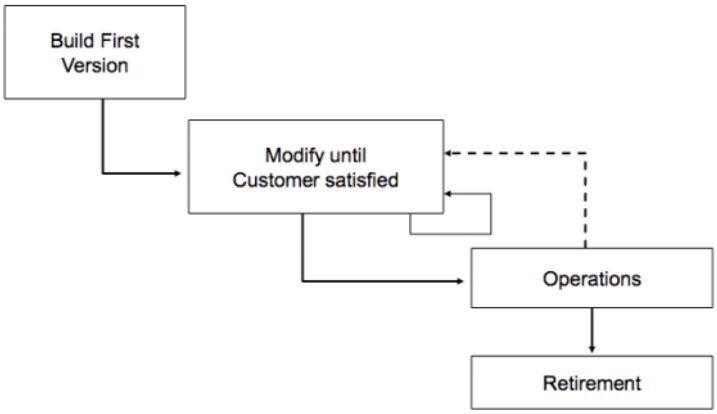
SDLC Models

List some different Software Development Lifecycle Models...

- Code-and-Fix
- "Code-Like-Hell"
- Waterfall
- Waterfall with Feedback
- Incremental
- Spiral
- "Pinball Machine"

*What is the **Code-and-Fix Model**?*

- Start with empty program, code and debug and until it works
- Useful for small programs, "hacking" single use/personal programs
- Problems when applying to bigger/serious projects:
    - Versioning, config mgmt, change mgmt, etc.
    - Difficult to coordinate multiple programmers



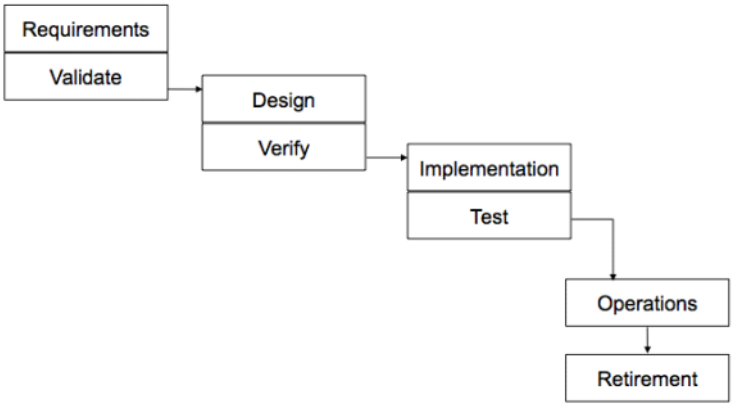
*What is the **"Code-Like-Hell" Model**?*

- Like previous (Code-and-Fix), but more hectic
- Focuses on software development, with little attention to planning/design
- Not a good model... Burnout time!
- Might work in startups (short term), if team is highly motivated

*What is the **Waterfall Model**?*

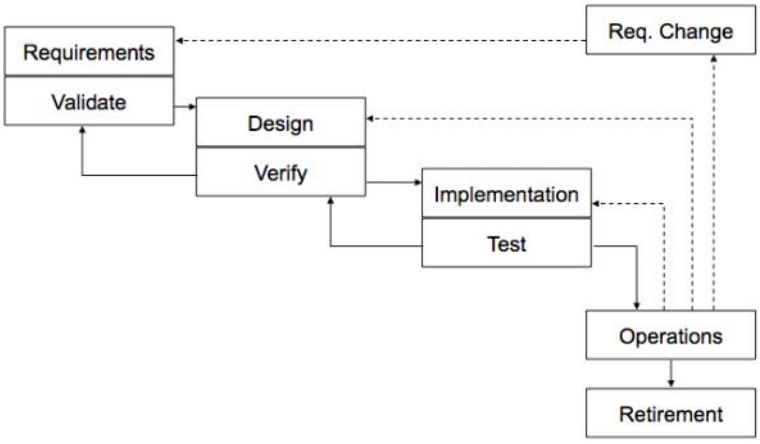
- Copy of assembly line process
- Fixed sequence and strict separation of steps

- Well-defined deliverables
- No going back... once a phase is done, it's done
- Advantages:
    - Measurable progress to sooth the fears of upper mgmt...
    - May work in an environment that is well understood and stable
- Disadvantages:
    - Lots of paperwork up front, little results visible until late in dev cycle
    - Means customers have to be patient and trusting
    - Tech problems not encountered until implementation phase
    - Changes become costly, esp. if they require backing up phases
    - Requirements can change which means you have to back up....



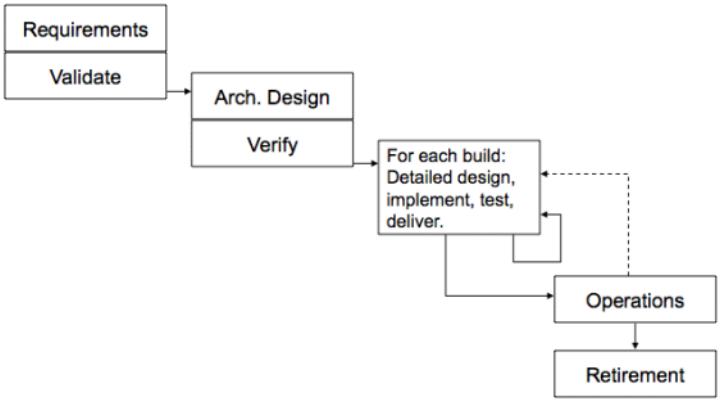
*What is the **Waterfall Model with Feedback**?*

- A patch to try to fix some of the problems with waterfall; not a true sol'n
- Adds a feedback step that allows to go back to previous phases



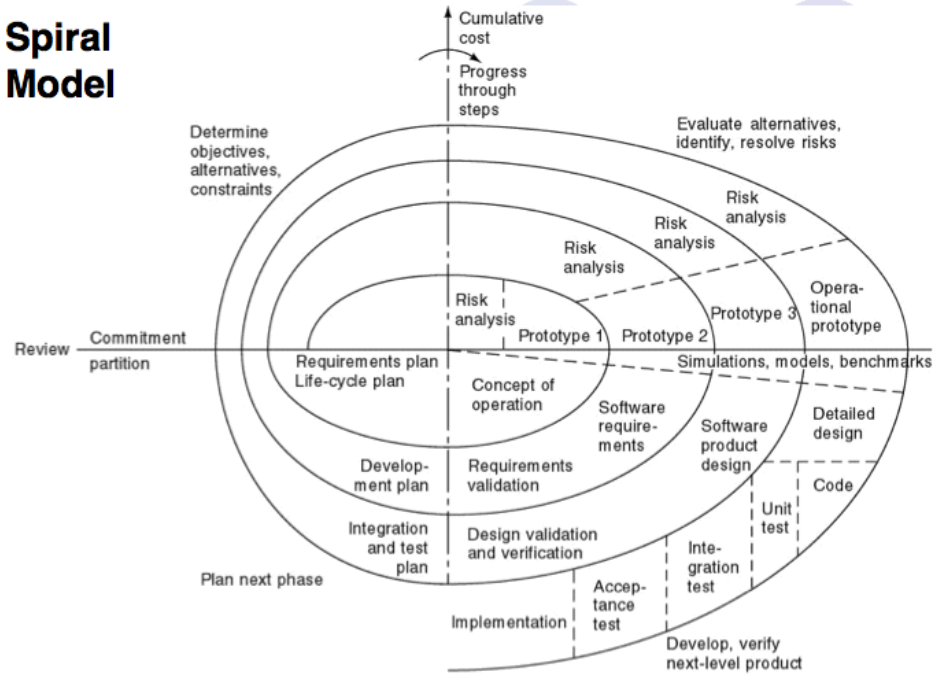
*What is the **Incremental Model**?*

- Iterations driven by features, each iteration is scaled-down waterfall
- Better than standard waterfall because shorter process



*What is the **Spiral Model**?*

- Similar to incremental (each iteration is scaled down waterfall)
- Each iteration driven by "risk management"
    - Typically most risky/most important parts of the system identified and implemented first



*What is the **"<Pinball machine>" Model**?*

- Unofficially unrecognized yet we all use
- Chaos? Madness?

Summary

Life cycle approach makes software development:

- predictable
- repeatable
- measurable
- efficient
- improvable

3. Requirements: an Introduction

Thursday, February 5, 2015 2:06 PM

What is a requirement?

A requirement is an expression of desired behaviour. (Should focus on customer's needs.)

Phases of Requirements

- Elicitation
  - "to elicit" means "to bring out or draw out"
  - The goal of this phase is to bring out what the user already knows about the future system (easy), and also to draw out what the user doesn't yet know or think about the system (much more difficult).
    - And then, organize this knowledge in useful form
  - Usually uses informal (customer) language
- Specification
  - Restates the requirements in a (semi-)formal way.
- Validation

Alternatively stated:

- Collect, analyze, refine, and organize the information necessary and sufficient for successful design of the system.

Functional Requirements

- The "What"
- features/functions the system should support

Non-functional Requirements

- The "how"
- the way in which the interactions are conducted
  - Usability
  - Reliability/availability
  - Performance (speed, latency)
  - Robustness
  - Security
  - Maintainability
  - Extendibility

Constraints

- Design Constraint
  - A design decision (such as platform or interface components) which may be dictated by backward compatibility or some other (possibly non-technical) reason
    - e.g. Operating system, programming language, GUI toolkit, DB management system, ...
- Process Constraint

    - Restriction on technique or resources

Traceability

- Important but often ignored quality.
- Source Traceability
    - links from requirements to stakeholders who proposed these requirements
- Requirements Traceability
    - links between dependent requirements
- Design traceability
    - links from the requirements to the design

Prioritization

- Different stakeholders have different sets of requirements.
- To resolve such conflicts, sometimes it helps to prioritize.
- Can prioritize however you want, but often we use three categories:
    - essential
    - desirable
    - optional

How formal should a requirement be?

- Users don't speak tech jargon and prefer informal prose
- Developers prefer something more formal (maybe even math notation)
- Natural language is *not* always the best way to specify a product
    - But requirements always start with natural languages
- So take two takes with requirements:
    - C-Requirements
        - **Customer Requirements**, specified in language they understand
        - Interview & write with customer.
        - Important step: Make customer sign off!

    - D-requirements
        - **Detailed Requirements**, uses more formal/semi-formal models, using notation familiar to developers
        - Use C-requirements as the starting point.
          Uses *models*: Abstractions that represent the system but without unnecessary detail
        - Customer should ideally sign off on these as well.

4. Intro to XP - Extreme Programming

Thursday, February 5, 2015 9:57 PM

Agile Software Development Manifesto

Value:

Individuals and interactions	over	Processes and tools
Working software	over	Comprehensive documentation
Customer collaboration	over	Contract negotiation
Responding to change	over	Following a plan

(While there is value in items on the right, those on the left are valued more.)

Customer's Bill of Rights

- To **set objectives** for the project and have them followed
- To **know how long it will take** and **how much it will cost**
- To **decide which features** are in and out of the software
- To know the **project's status** at any time during the development
- To make **reasonable changes** throughout the project, and know the costs of these changes
- To be apprised regularly of the **risks** that could affect cost/schedule/quality.
- To have ready **access to project deliverables** throughout the project.

Developer's Bill of Rights

- To **know the project objectives** and be able to clarify priorities
- To know in **detail** the project they are supposed to build.
- To have ready **access to the customer**/manager/marketer/anyone else responsible for functionality.
- To work each phase in a technically responsible way
- To **approve effort and schedule estimates** for any work they will perform
- To have project's **status reported accurately** to customers & upper mgmt
- To work in **productive environment** free from frequent (and unnecessary) interruptions

Extreme Programming (XP)

- An implementation of agile methodologies.

The basic problem: Risk

XP views risk as the main problem of software development:

- Schedule slips
- Project canceled
- Business Changes

- Staff Turnovers
- ...

And so XP is a methodology that "addresses risk at all levels of the development process"

What are the four "Control" variables of every project?

1. **Scope**: How much is to be done
2. **Resources**: How many people available?
3. **Time**: When the project/release will be done
4. **Quality**: How good the software will be, how well tested it will be.

External forces get to pick the value of any three...
The dev. team picks the value of the fourth

Visibility

- The value of the four variables need to be "visible"
- If stakeholders (customers, management) can see all four variables, they can choose which to control.
- If they do not like the resulting value of the fourth variable, they can choose to change the inputs or choose to control a different set of three.

Scope is critical

XP argues scope is the most important control variable

XP Rules of Thumb

- Focus on Simplicity (Do the simplest thing that could possibly work)
- Build for NOW (Build what you need currently, not for "just in case" in the future)
- Build Clean (Continuous Refactoring)
- Customers specify, Developers estimate (Iteration Planning)
- Steering to Success (Release Planning)
- Avoid Big Design Up Front (Design will emerge as we go along)

Risks of XP

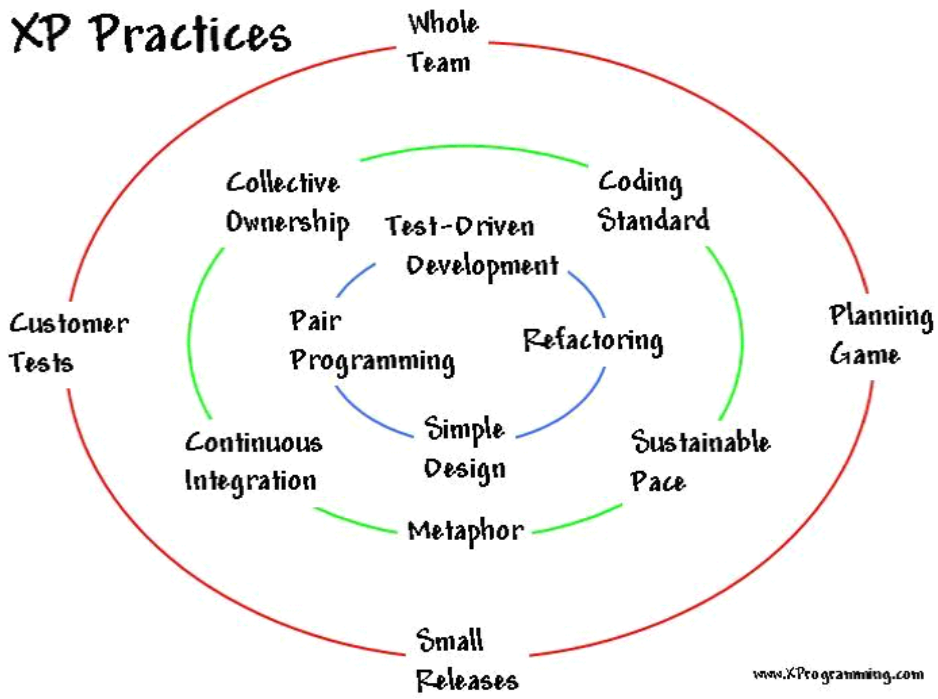
Risk	***How does XP help?***
Unmaintainable Code	Simple, clear code, relentlessly tested, mercilessly refactored
Might never get done	Clear schedule, small frequent releases with business value
Might not work	Highest priority first, write unit tests first, continuous integration
People might not know what to do	User stories, engineering tasks, pair programming

Prerequisites for XP

- Groups and people that can talk to each other
- Language in which programs can be read and changed
- Close contact with users and other stakeholders
- Business people make business decisions, tech people make tech decisions

Things we should do

- Staged delivery
    - which means we have a continuous stream of value to the customer
- Adaptation to change
- Constant feedback from the customer
- Regression testing, unit tests
- Constantly review design and code
- Refactor the code often to make it cleaner, simpler, faster, easier to maintain
- Do not work overtime!



XP Practices

- Simple Design
    - Do the simplest thing that could possibly work
    - Don't duplicate logic
    - State every intention important to the programmers
    - Try to have fewest possible classes and methods
- Test-driven development
    - Write tests before code
        - Unit tests before each class
        - End-to-end tests before any classes
    - Convert each user story to a set of tests

    - All unit tests must run 100% of the time
    - Don't worry if you can't test everything
        - Things like GUI, real time I/O, or complete test coverage are hard to achieve
- Pair Programming
    - All code written in pairs, one person at monitor (driver), the other at keyboard (navigator), with frequent switching
    - Switch pairs often
    - People are slightly less productive in pairs, but quality of product improved
    - Helps to pair people with different skill levels
- Refactoring
    - Remove redundancy, eliminate unused functionality
- Coding Standards
    - Adhere to common coding standards--makes it impossible to tell who wrote what
    - Standard requires least work possible, minimizes time required to understand someone else's code
    - Standards should not be forced upon the team; they should be accepted voluntarily by the entire team, and grow as the project moves on
- Collective Code Ownership
    - Everything is done together as a team
    - If you see a problem, fix it
    - Emphasize communication
    - "Egoless programming"
- Continuous Integration
    - Constant tests run as code changes
    - Dedicated release machine may help
- Metaphor
    - Try to find metaphor to help you think about system
    - Name classes, methods etc consistently
- Sustainable Pace
    - Developers should live normal lives
    - Overtime banned
    - Only stay after 5pm as an exception, not regular practice
- Small Incremental Releases
    - Deliver real business value, on a very short cycle. (two weeks?)
    - Benefits:
        - Business value sooner
        - rapid feedback
        - sense of accomplishment
        - reduce risk
        - customer confidence
        - adjustments to requirements
- On-site Customer
    - A customer representative must sit with the team, available to answer questions, resolve disputes, and set small-scale priorities
- The Planning Game
    - Used to figure out how to develop individual pieces for a release
    - All players are from dev. team (w/ customer rep. on hand)
    - Exploration -- Turn stories into tasks
    - Commitment -- Accept a task, estimate, set load factors, balance so that

no-one is overcommitted.
    - Steering -- Implement a task, record progress, recover when someone turns out to be overcommitted.
        - Reduce scope, shed nonessential tasks, get more or better help, etc

Why is XP/agile appealing?

- Stresses customer satisfaction
    - methodology design to deliver value to customer
- Emphasizes team work
    - Managers, customers, developers all part of team dedicated to delivering quality software

Benefits of XP/agile methodologies

- Software stays soft
- Handles changing requirements
- Quickly produces something useful, and keeps making useful enhancements
- Developers feel they're in control
- Average developers can produce useful software fast

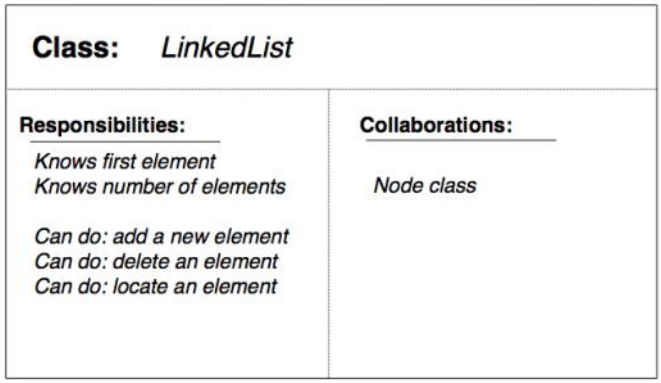
Downsides of XP

- Not everyone likes everything about it (managers/programmers)
- Needs discipline (like any other approach...)
- A lot of anecdotal evidence, little hard evidence in favour of XP
- Not suitable for all kinds of projects

5. User Stories and the Planning Game

Thursday, February 5, 2015 11:56 PM

User Stories

- Written by customer
- CRC Cards:
  - Class, Responsibilities, Collaborations
  - Defines what a class knows and can do, and which other classes the class will collaborate with, e.g.:
  - 

- Time Estimation
- The Planning Game
- ... see slides if more info needed...

6. Customer Requirements Specification

Friday, February 6, 2015 12:01 AM

What's wrong with *user stories*?

- They're informal
- Too short to describe sufficient detail
- Index cards/post-its not a great format for requirements specificaitions

The next best thing...

Use Cases

- Use case model (the set of all uses cases) should in theory be a complete description of system functionality and its environment
- Specifies user views of essential system behaviour
- ACTORS represent roles (a type of user of the system)

A use case consists of:

- Unique name
- Participating actors
- Trigger
- Entry conditions
- Flow of events, main and alternative (exceptional)
- Exit conditions: success and minimum guarantees
- Special requirements

Actors

- Models an external entity which communicates with the system:
  - User
  - External system
  - Physical environment
- e.g.
  - Passenger: A person in the train
  - GPS Satellite: Provides the system with GPS coords

An Example Use Case

- **Brief Description**:
  - This use case describes the Sae Report process.
  - This use case is called from the use case Manage Reports and from the use case Exit Report
- **Actors**:
  - User (Primary)
  - File System: Typical PC or network file system with access by user. (Secondary)
- **Triggers**: (what starts the use case)
  - User selects operations through tasks in the Manage Reports use case or

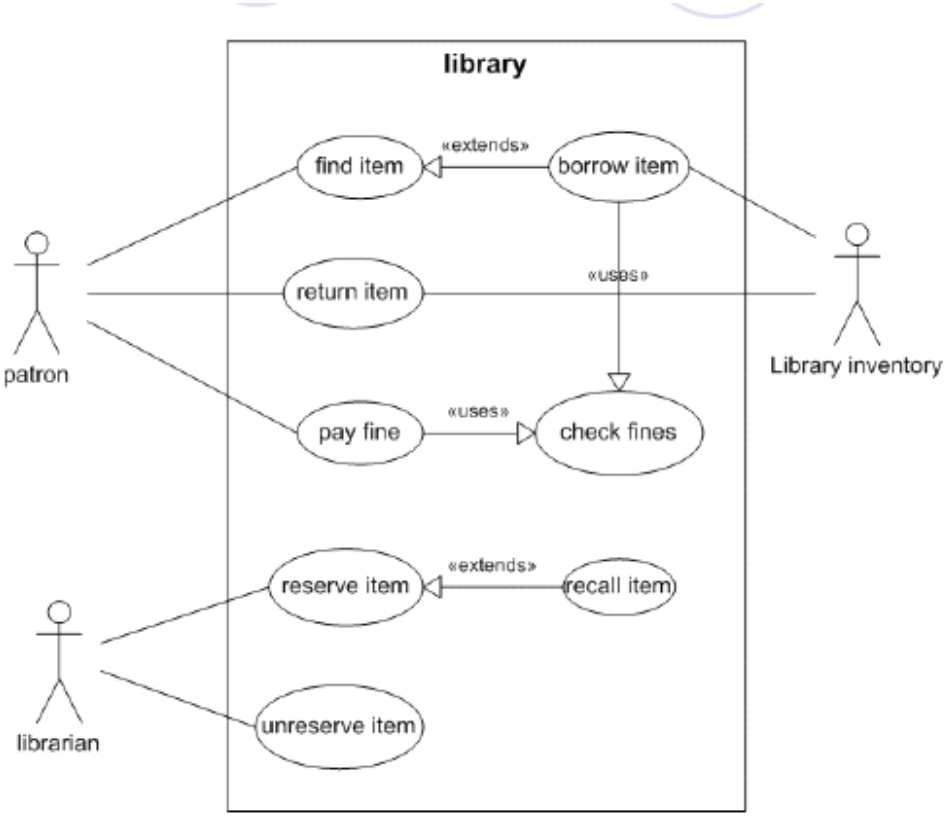
Exit Reports use case (which is included in the Manage Reports use case) to call this use case.

- **Flow of Events**
  - **Basic Flow -- Save New Report**
    - Use case begins when user selects Save Report
    - System detects that report name status is "not set" and prompts for new report name. User choose report name; system validates that the report name doesn't exist in Report List yet. Adds entry to Report List.
      - User cancels save operation... Use case ends
      - or
      - System updates Report List with Report information. System creates unique report file name if not already set, and saves report specs to file system.
      - Report is set as "unmodified" and name status set to "set".
      - Use Case ends with report displayed.
  - Alternative Flows and/or Subflows
- **Special Requirements**
  - None
- **Preconditions** (must hold for the use case to begin at all)
  - A data element exists on teh machine and has been selected as the "current element"
  - A Report is currently displayed and set as the "Current Report"
  - Report status is "modified".
- **Postconditions**
  - **Success Postconditions** (Success Guarantee)
    - System waiting for user interaction. Report loaded and displayed. Report List is updated with report name, etc. as required by specific Save operation. Report status is "unmodified". Report Name Status is "Set".
  - **Failure Postconditions** (Minimal Guarantee)
    - System waiting for user. Report loaded and displayed. Report status is "modified". Report name status same as at start of use case. Report list still valid (Report list cleaned up when save fails as necessary.
- **Extension Points**
  - None

Use Case Diagrams

- *Large box*: System boundry
- *Stick figures outside the box*: actors, both human and system
- *Each oval inside the box*: A use case that represents some major required functionality and its variant
- *A line between an actor and use case*: the actor participates in the use case.

e.g. use case for a library (simplified):



Other Special Relationships:

- **<<extends>>** relationships represent exceptional (alternative) flows which are factored out of the main event flow for clarity e.g. "recall item" in diagram above
    - The arrow of an <<extends>> relationship points to the extended use case
    - e.g. "Recall Item" <<extends>> "Reserve Item" (Arrow points to reserve item)
- **<<includes>>** or **<<uses>>** relationship represents behaviour that is reused by several use cases and so is factored out of the use case
    - The direction of <<includes>> relationship is towards the use case(s) that use it (unlike the <<extends>> relationships)
    - e.g. "Borrow Item" <<uses>> "Check Fines" (arrow points to check fines)
- Basically put the arrows as you would say it linguistically.

Estimating from Use Cases

- Step 1: Determine whether each actor is simple, average, or complex:
    - Simple Actor: Another system w/ a defined API
    - Average complexity actor: Another person that interacts through a protocol or text-based interface
    - Complex actor: Person interacting through GUI
- Count actors in each category, multiply the counts with 1, 2, or 3, and add the results

... skipping slides 20-24 for now...

7. Developer Requirements

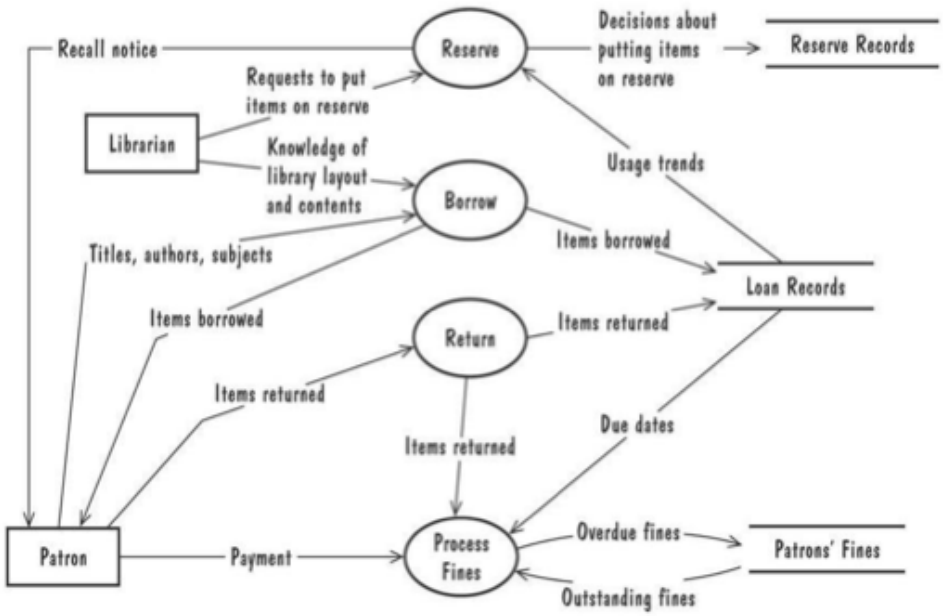
Friday, February 6, 2015 12:02 AM

Data Flow Diagrams

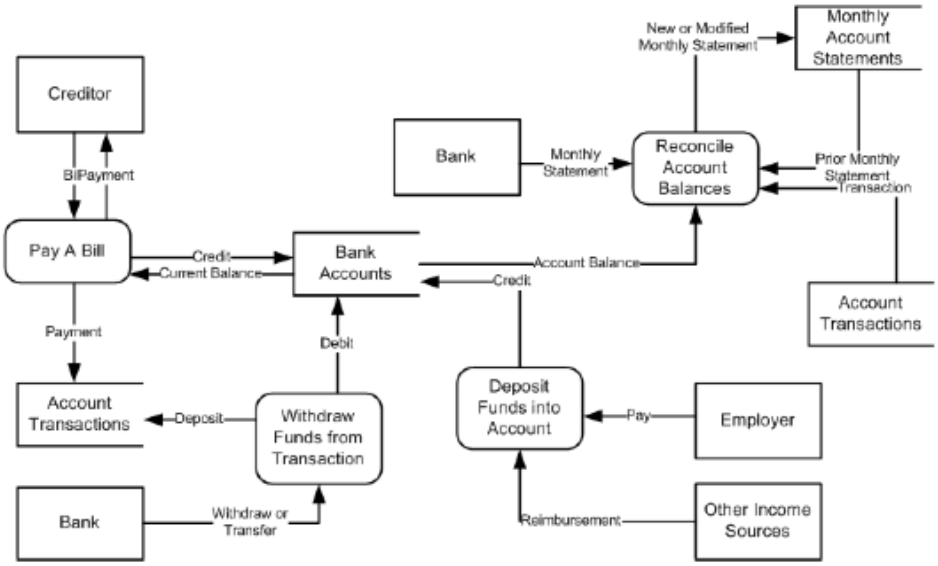
- Models functionality in top-down fashion
- Basic Elements:
  - Rectangles represent actors
    - Entities that provide input data or receive output result
  - A Data Store
    - A formal repository or database of information
  - A bubble or circle
    - Represents a process that transforms input data flows into output data flows
  - An arrow
    - Represents a data flow, from an external entity/process/data store to a process/data store/external entity
  - NOTE: Data can't flow between data stores or between external entities without being transformed in a process.

Examples:

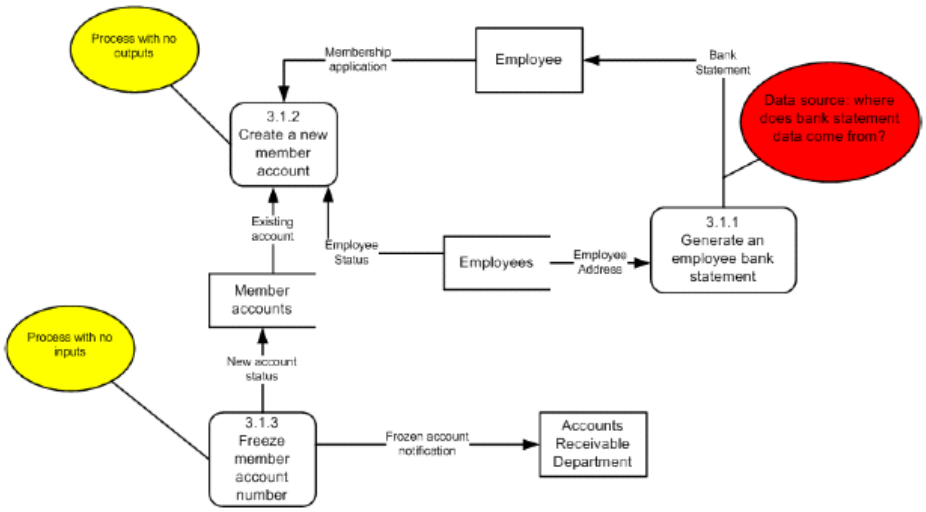
Context Diagram -- top level data-flow diagram for the library problem:



Context Diagram -- for Bank:



Data Flow Diagram -- COMMON ERRORS:



Data Flow diagram decomposition

- A process can be decomposed ("exploded") into another data flow diagram.
- There are no additional external entities here, and there may actually be no external entities at all.
- But there may be additional data stores.
- The process is then repeated.

What to do with data stores?

- CRUD
    - Create, Read, Update, Delete

Data Flow Diagrams

Advantages

- Provides an intuitive model of proposed system's high-level functionality and

of data dependencies among processes
- Can be transformed into a function-oriented design

Disadvantages

- Can be ambiguous and discouraging to software developer who is less familiar with the problem being modeled
- Suitable for certain types of problems but not others
- Can be transformed into an OO design but requires manual work

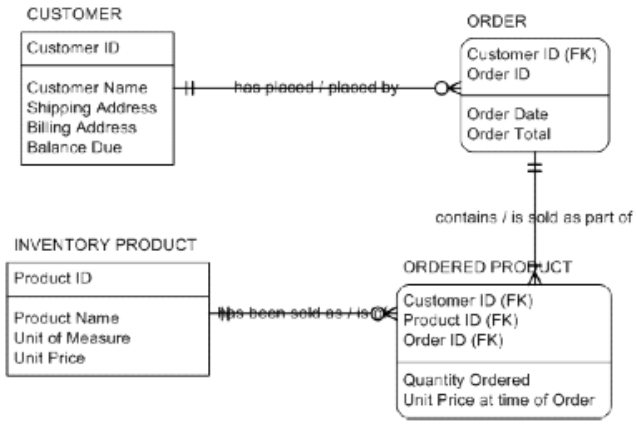
Entity Relationship Diagrams (ERD)

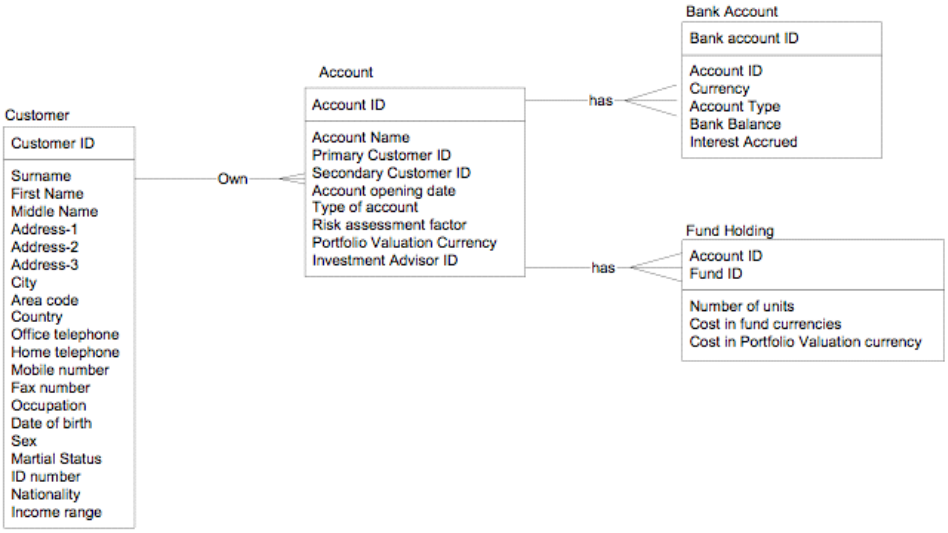
Three core constructs:

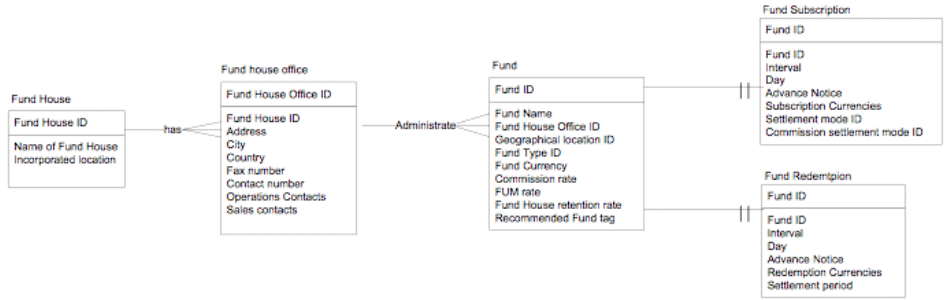
1. An **Entity**: Depicted as a *rectangle*, represents a collection of real-world objects that have common properties and behaviors
2. A **Relationship**: Depicted as an edge between two entities, with diamond in the middle of the edge specifying the type of relationship.
3. An **attribute**: An annotation on an entity that describes data or properties associated with the entity.

Examples:

- CUSTOMER has placed an ORDER
- ORDER consists of 0 to M(any) instances of ORDERED PRODUCT
- INVENTORY PRODUCT has been sold as ORDERED PRODUCT

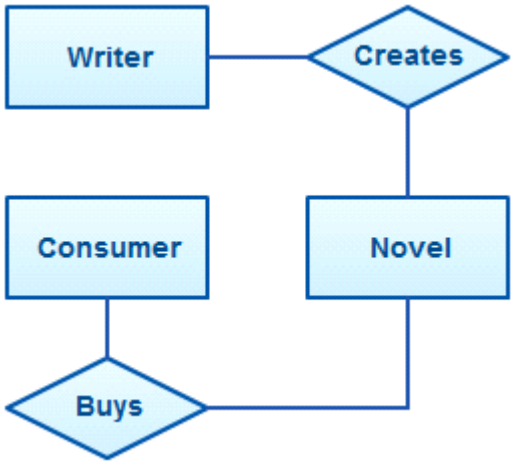




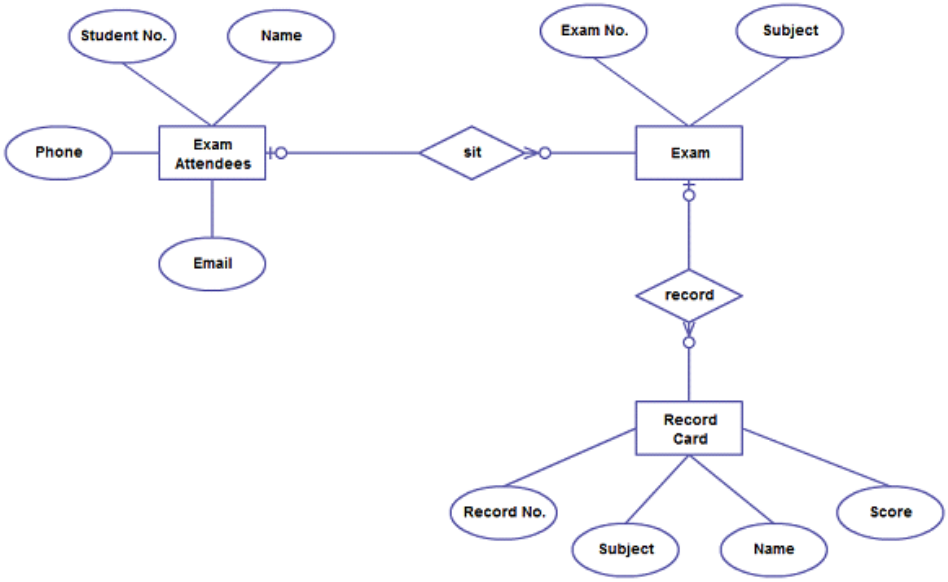


More Examples (from external online sources):

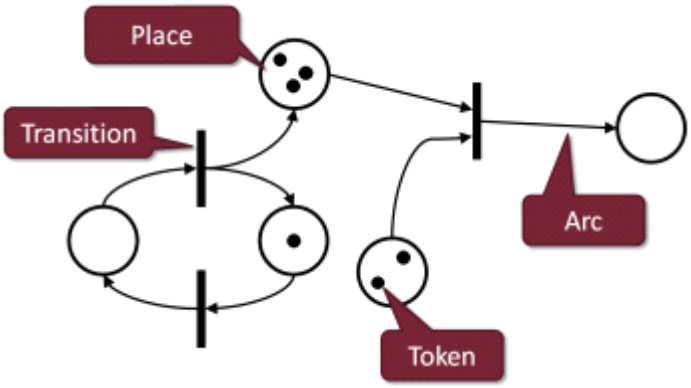
Very Simple Example:



Example, where attributes written as circles connected to entity:

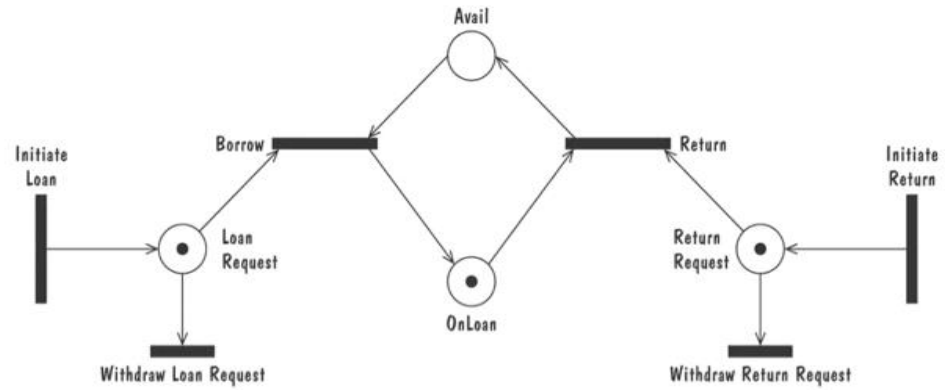


Petri Nets (State Machine)



- Circles (places): Represent activities or conditions
- Bars: Represents transitions
- Arcs: Connect a transition with its input places and its output places.
- The places are populated with tokens, which act as enabling conditions for the transitions.
- Each arc can be assigned a weight that specifies how many tokens are removed from arc's input place, when the transition fires.

- Petri net of book loan



8. Introduction to Design

Friday, February 6, 2015 12:03 AM

Design

Six Dominant Principles

1. Modularity
    - Coupling
    - Cohesion
2. Interfaces
3. Information Hiding
4. Incremental Development
5. Abstraction
6. Generality

Modularity

- Principle of separating various unrelated aspects of a system, so each one can be studied in isolation (aka separation of concerns)
- Each module should have a **single purpose** and be *independent* of the others
- This is good because:
    - Each module will be easy to understand and develop
    - Easier to locate faults
    - Easier to make changes to system (change in one module doesn't affect many other modules)

Module Independence/Interdependence

- Independent modules: If one can function without the presence of other
- Desirable because:
    - Modules can be modified separately
    - Can be implemented/tested separately
    - Programming cost decreases
- More connections between modules means they are more dependent on one another.
- Independence is not possible -- modules must cooperate with each other
- Modules should not depend on one another *more than is necessary to fulfill their duties*

Coupling

- The level of dependence / the strength of interconnections between modules
- **Highly coupled**: Strong interconnection (bad)
- **Loosely coupled**: Weak interconnections (good)
- Coupling decided during high level design (cannot be reduced during implementation)
- To reduce coupling:
    - Minimize number of interfaces per module (keep it simple)
    - Minimize the complexity of each interface

    - Pass information exclusively through parameters
- Coupling increases if:
    - Indirect/obscure interfaces used
    - Internals of module are used directly
    - Shared variables (state) used for communication

- Two kind of information flow: data or control
- Transfer of control information (e.g. passing a "what-to-do" flag):
    - Action of module depends on the information
    - Makes modules more difficult to understand
- Transfer of data information (e.g. pass integer to sqrt function):
    - Module can be treated as input-output function
- Lowest coupling: Interfaces with only data communication
- Highest coupling: Hybrid interfaces

- Coupling in OO
    - Least interaction coupling if methods communicate directly with parameters:
        - With least number of parameters
        - With least amount of data being passed
        - With only data being passed, rather than control info.

Coupling in Objects

Component Coupling

When *Class A* has variables that are actually instances of another *Class C*
- A has instance variables of C
- A has some parameters of type C
- A has a method with a local variable of type C

Inheritance Coupling

Two classes are coupled if one is a subclass of the other.

- Worst: When subclass modifies the signature of a method or deletes a method
- Also bad: When two ore more methods have the same signature but different implementations
- Best (Least Coupling): When subclass only adds instance vars and methods, but does not modify any of the inherited ones

Data Coupling

Output from one module is the input to another
- e.g. using parameter lists to pass items between routines
- in OO systems this is called *Common Object Occurence*:
    - Object A passes Object X to Object B
    - Then, Object X and B are coupled; any chance to X's interface may require a chance to B

Control Coupling

Passing a flag as a parameter.

- Common Occurence:
  - Object A passes a parameter X to Object B
  - Object B makes a decision on what to do depending on value of X
- Or:
  - Object A sends a message to B
  - B responds with control information (e.g. error code), then A uses that to determine what to do next
- Cure: Use exceptions.

Common Coupling

Two or more modules use the same global data (e.g. a global variable).

Stamp Coupling

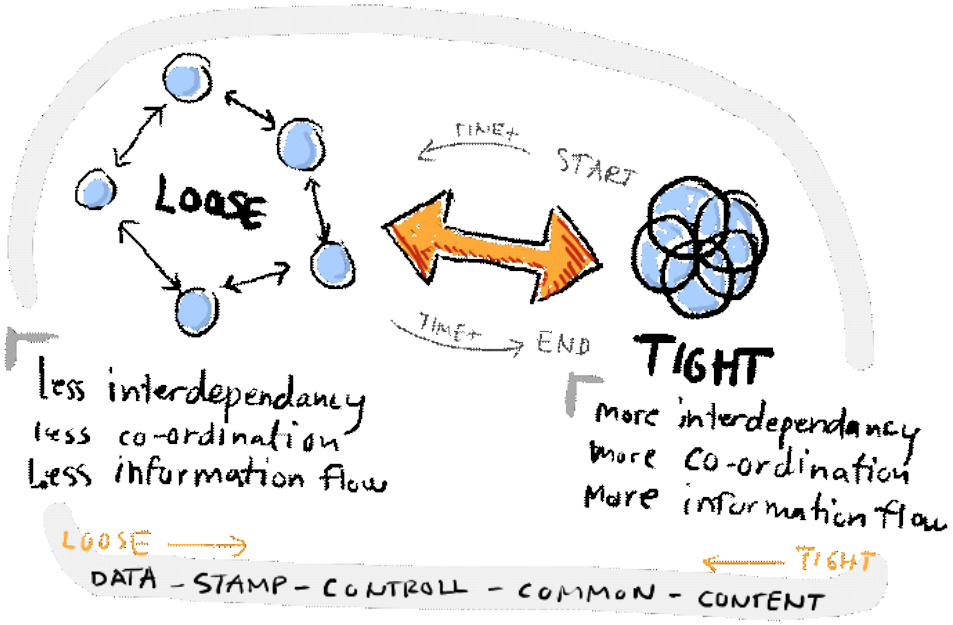
Two objects take the same data type as a parameter. Same data but they only use part of it. (Similar to common coupling but a bit better.)

Content (or internal data) Coupling

One module directly modifies data inside another module.
(Highest coupling)

Lexical Content Coupling

When one module refers to the inside of another one (as in C/C++ header files)



Cohesion

The degree to which the elements of a module belong together.
(How tightly bound are the elements of a module.)

Reduced by:

- Minimizing the relationship between components of different modules
- Maximizing relationship between elements of same module

Cohesion and coupling are interrelated: the greater the cohesion of modules, the less they are coupled.

Levels of Cohesion

- Coincidental (Lowest Cohesion)
- Logical
- Temporal
- Communicational
- Sequential
- Functional (the strongest and most desirable)

Cohesion in OO

Classes are the modules, different types of cohesion possible:

- **Method cohesion**: Why different functional elements are together in a method
  - Each method should implement a clearly defined function.
- **Class cohesion**: Why different attributes and methods are together in a class
  - Class should represent a single concept with all elements contributing towards it.
  - When multiple concepts encapsulated, cohesion is not as high.
- **Inheritance cohesion**: Focuses on why classes are together in a hierarchy.
  - Two reasons for subclassing: generalization/specialization and reuse
  - Cohesion is higher if the hierarchy is intended to provide generalization/specialization

Open-Closed Principle

Principle: A module should be *open for extension* but *closed for modification*.

- Helps in achieving modularity
- Behaviour can be extended to accommodate new requirements, but existing code is not modified.
- Minimizes risk of having existing functionality breaking due to changes
- In OO this principle is satisfied using inheritance and polymorphism

Liskov's Substitution Principle

Principle: Program using object o1 of base class C should remain unchanged if o1 is replaced by an object of a subclass of C.

If hierarchies follow this principle, the open-closed principle gets supported.

Interfaces

Interface defines what services the software unit provides to the rest of the system, and how other units can access those services.

- e.g. interface to an object is collection of object's public operations and the operations' signatures (with name, params, return vals)
- Define required services, assumptions
- Describes what the unit requires of/provides to the environment
- Interface Specification should communicate everything other system developers would need to know to use software unit correctly:
    - Purpose
    - Preconditions (assumptions)
    - Protocols
    - Postconditions (visible effects)
    - Quality attributes

Information Hiding

> "Information hiding is the principle of segregation of the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed.
>
> "The protection involves providing a stable interface which protects the remainder of the program from the implementation (the details that are most likely to change).
>
> "Written another way, information hiding is the ability to prevent certain aspects of a class or software component from being accessible to its clients, using either programming language features (like private variables) or an explicit exporting policy."

- Wikipedia

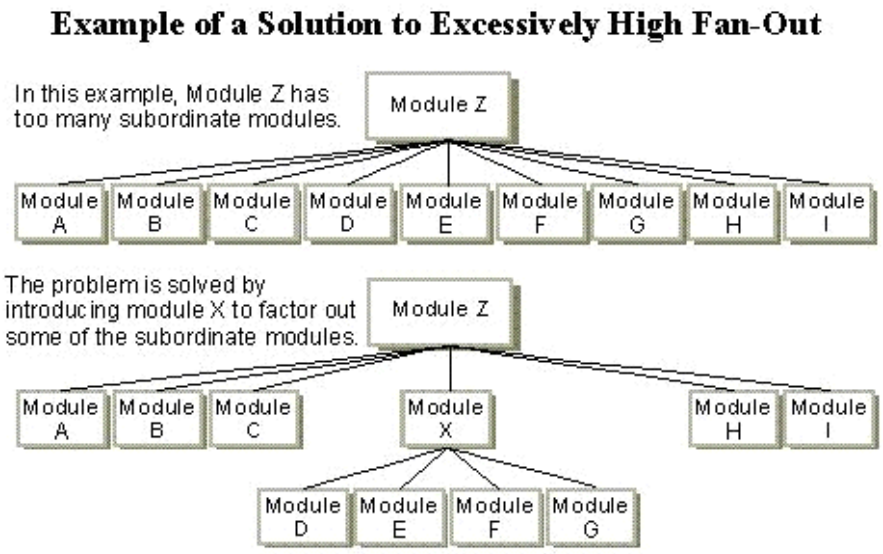
Advantage: Resulting software units are loosely coupled

Incremental Development

- Start by mapping out the units' **uses relation**
    - Relates each software units to other software units on which it depends
- **Uses graphs** can help to identify progressively larger subsets of our system that we can implement and test incrementally

- **Fan In**: The number of units that use a module (number of superordinate modules).
    - High fan-in generally improves the design of a system
    - High fan-in shows that an object is being used extensively by other objects which indicates re-use.
- **Fan Out**: The number of number of units used by a module (number of subordinate modules).
    - Optimum fan-out is 7 +/- 2 (Because human mind has trouble dealing with more than 7 things at once.)
- **Sandwiching**: Sometimes uses graphs have cycles, which can be broken using a technique called sandwiching.
    - Decompose one of the cycle's units into two units, such that one of the new units has no dependencies
    - Can be applied more than once to break either mutual dependencies in

tightly coupled units or long dependency chains



Abstraction

A model or representation that omits some details so that it can focus on other details.

Often linked to *decomposition*, the process that spreads the functionality of the system into several modules (achieving modularity).

Generality

Principle of making a software unit as universally applicable as possible, to increase the chance that it will be useful in some future system.

Can make a unit more general by increasing the number of contexts in which it can be used. Several ways of doing this:

- Parameterizing context-specific information
- Removing preconditions
- Simplifying postconditions

9. Design Phase (con't)

Friday, February 6, 2015 12:03 AM

OO Design Process

Process Stages

Stages depend on organization using the process.
But some common activities in these processes include:

- Define the context and modes of use of the system
- Design the system architecture
- Identify the principal system objects
- Develop design models
- Specify object interfaces

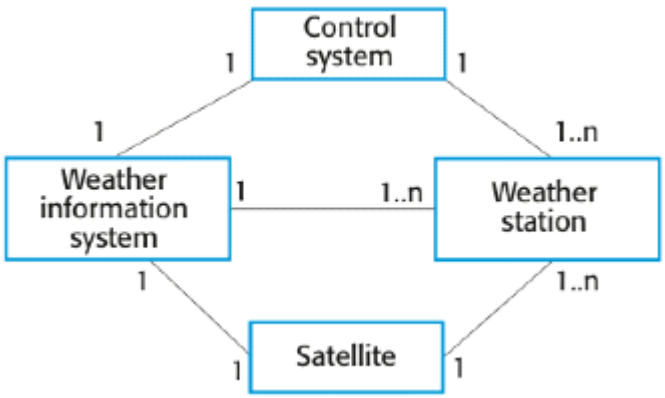
Example provided for a wilderness weather station.

System Context and Interactions

- Understanding the relationships between the software being designed and its external environment is essential for how to design and structure the system.
- Understanding of the context allows you to establish the boundaries of the system
- Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

System Context Model

Structural model that demonstrates the other systems in the environment of the system being developed.



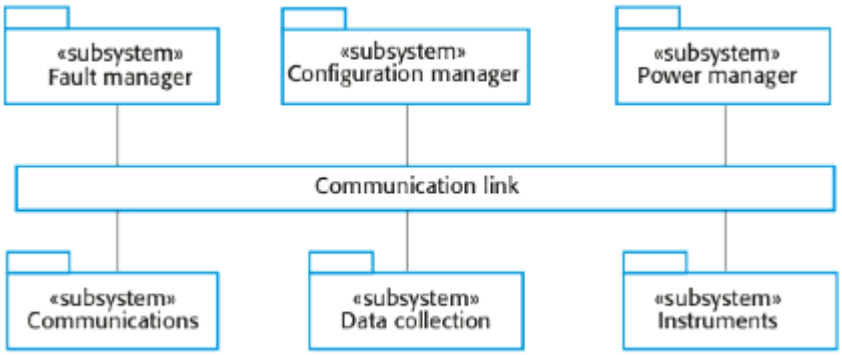
Interaction model

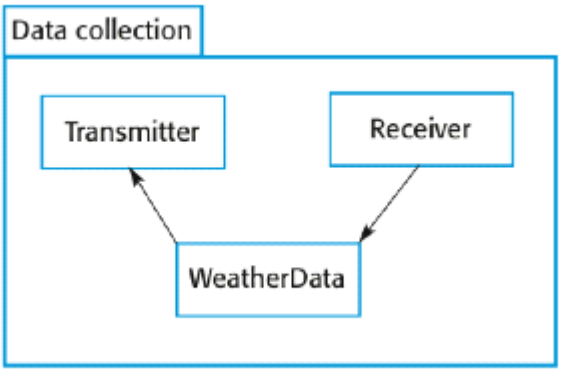
Shows how the system interacts with its environment as it's being used

Architectural Design

- Once interactions between the system and its environment have been understood, use this info for designing the system architecture.
- Identify major components that make up the system and their interactions, and organize them in some way (e.g. layered or client-server model)

e.g. Architecture of weather station:





Object Class Identification

- Difficult part of OO design
- Requires skill, experience, domain knowledge
- An iterative process, unlikely to get it right the first time

Approaches to identification:

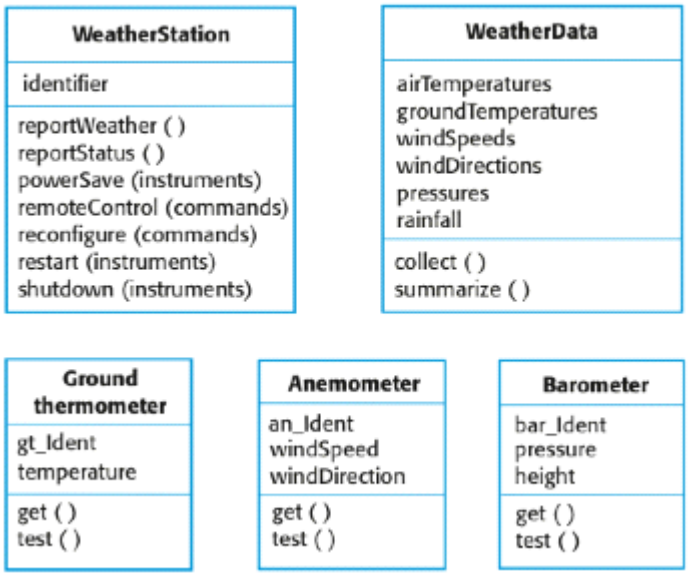
- Use a grammatical approach based on natural language description of the system
- Base the identification on tangible things in the application domain
- Behavioral Approach: Identify objects based on what participates in what behaviour
- Scenario-based analysis: Which objects, attributes, and methods in each scenario need to be identified.

e.g.

Object class identification in the weather station system may be based on the tangible hardware and data in the system:

- Ground thermometer, Anemometer, Barometer
  - These are application domain objects that are 'hardware' objects related to the instruments in the system
- Weather station
  - This is the basic interface of the weather station to its environment, it reflects the interactions identified in the use-case model
- Weather data
  - This encapsulates the summarized data recorded by the instruments

Weather station object classes:



Design Models

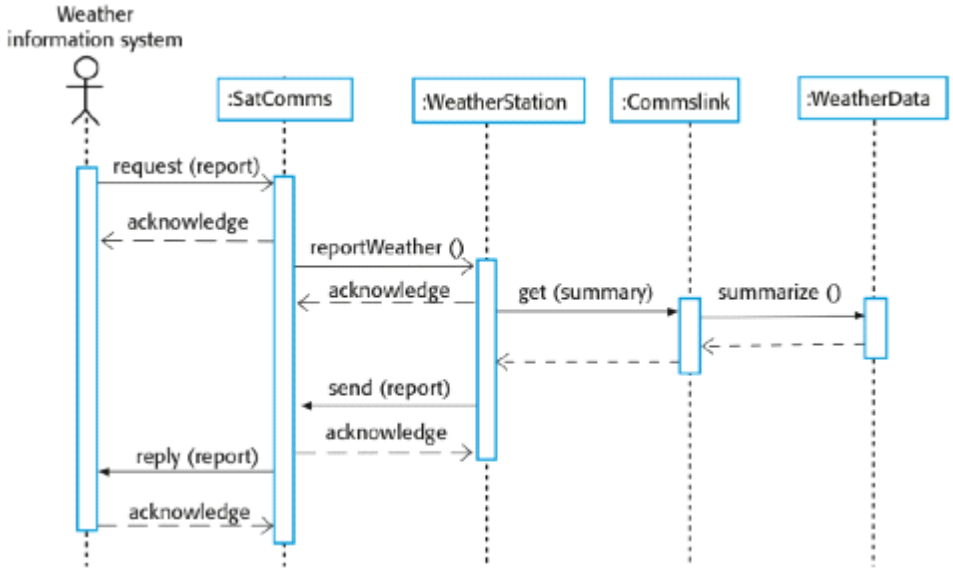
Design models show the objects and objects classes and relationships between these entities.

- Static models describe the static structure of the system in terms of object classes and relationships
- Dynamic models describe the dynamic interactions between objects

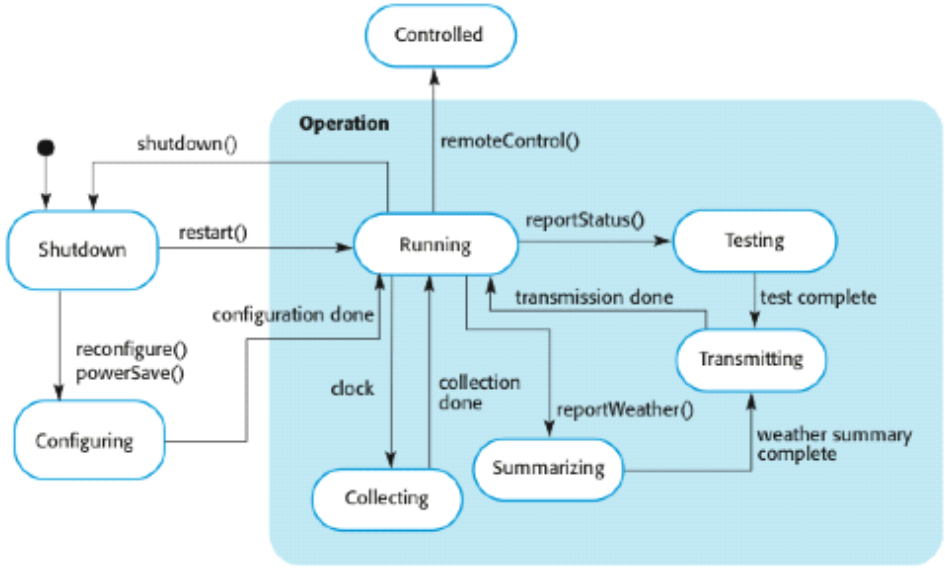
Examples of design models:

- Subsystem models showing logical groupings of objects into coherent subsystems
- Sequence models that show sequence of object interactions
- State machine models that show how objects change their state in response to events
- Other models include: use-case models, aggregation models, generalization models, etc.

e.g. sequence diagram describing data collection:



e.g. Weather station state diagram:



Interface Specification

Object interfaces have to be specified so that the objects and other components can be designed in parallel.

e.g. weather station interfaces:

Build or Buy

- In many domains, you can buy an off-the-shelf system that can be adapted to suit your requirements.
- In this case, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

Key Points

- Software design and implementation are interleaved activities.
- The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- The process of OO design includes activities to:
    - Design the system architecture
    - Identify objects in the system
    - Describe the design using different object models
    - Document the component interfaces
- A range of different models may be produced during an OO design process:
    - Static models (Class/generalization/association models)
    - Dynamic models (sequence/state machine models)
- Component interfaces must be defined precisely so that other objects can use them.