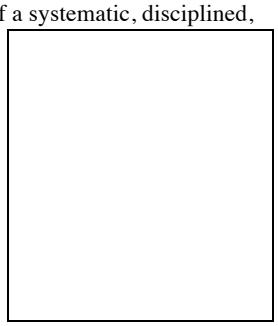
Software Engineering

-IEE Standard 610.12: The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.



-*Software vs. Hardware reliability Curve*

-software (and/or) it's environment frequently changes

-causes what's known as 'bi-trot', aka old software is no longer compatible or useful

Facts:

-The sooner you begin writing code, the longer it will take to finish

-Software and design reviews are often more effective than testing (find 5 times more bugs)

Software Bug→Space Disaster!

-$7B project over 10 years, Rocket exploded after 40 seconds because the software attempted to cram a 64-floating point number to a 16 bit integer and it failed…

180 Degree Bug

-Torpedoes that deviated more than 90 degrees where set to explode to prevent from destroying ship

-Ship fired a torpedo but it got jammed, captain turned more than 90 degrees, ship exploded…

Fixes: Design and Specs before building!

Parallel Development

-allows multiple people to work on different modules, saves time

-issues: more people→more communication, individual tasks must not be too fine-grained (i.e. small), inherent constraints (bottlenecks)

Interfaces

-Specifications that explain how modules work together

-boundaries between components and people

-How to design

-should not change much

-everyone should understand the interfaces

-must decompose a system into separate pieces with boundaries (i.e. what it does, how we build it, who builds it)

Conclusion:

-Specifications: Know what you want to do

-IMPORTANT: define what you want to do, and ensures everyone understands the plan

-Design: Develop an efficient plan for doing it

-Programming: do it

-Validation: Verify you have what you want

-Iterate: repeat…over and over again

Software Engineering Layers

Process: framework of the required tasks (e.g. waterfall, extreme programming)

Methods: technical "how to" (e.g. design and code review, testing)

Tools: automate processes and methods

Waterfall Model:

→Gather Requirements: figure out what it is supposed to do; determine your purpose; talk to customers

→ Specification: written description of what it does; covers all, situations, more compressive than requirements

→Design: system architecture, decompose to modules; specify interfaces between modules; focus on how the system works, rather than what it does

→Implementation: make a plan (i.e. what needs to be done, based on priority and testability);test each module

→Integration: put pieces together, use QA (quality analysis) to test entire system

→Product: ship to consumer, start maintenance and updates

Overall:

-Each stage leads to the next, no iteration

-Testing after each phase (verify the requirements, spec, design and code)

-Outside-In design: requirements, specs, design

-Inside out implementation: implement, integrate, product

Issues:

-Relies heavily on being able to asses requirements at the start; Very little feedback from users until very late (unless they go through the spec docs, unlikely); Spec problems may be found very late (i.e. when you are coding)

-Can take a long time before first version is produced

-Not tuned for speed, environment and other things may change before first product is launched

Rapid Prototyping

-Write a quick prototype, show it to users (used to refine requirements), then proceed as in waterfall model (i.e. throw away prototype and start with spec, design, coding, etc.)

Issues: sometimes hard to ditch the prototype, get attached

Note: Neither are true in life; Specs demand refined requirements; Design can affect specs; Coding problems can affect both; Final product may lead to changes in requirements

i.e. Waterfall model with 'feedback loops'

Iterative Models (MOST COMMON, System is always working, daily builds)

→Gather Requirements: don't spend too much time though

→Specification: it will evolve, identify what is likely to change

→Design: design for expected change, use abstraction to create 'crumple zones'

→Implementation : critical pieces first, can leave some unimplemented

→Iterate: show customer the prototype, update requirements, repeat

Advantages:

-Find problems sooner, i.e. get feedback and if design is feasible

-More quantifiable, when build 3 or 4 is done, product is 75% complete (hard to measure with waterfall)

-Trade off the risk of being slow

Disadvantages:

-Can have major mistakes in requirements, not as much time spent on them before build, may start coding too early

Conclusion*:*

Waterfall: top-down design, bottom-up implementation

-lots of upfront thinking, but slow, hard to iterate

Iterative, or evolutionary process

-build a quick prototype, then evolve, postpone some thinking

Extreme Programming

Waterfall model inspired by civil engineering, i.e. you **grow** software to meet changing requirements (not as concrete as civil engineering)

Control Variables: Time, Quality, Scope

-Scope is control variable for XP, similar to iterative but taken to the extreme

-Dev. cycle = 2-4 weeks

Process:

→Meet with customer to elicit requirements

-**user stores** and acceptance tests

→Planning

-break stories into tasks, estimate cost

-have client prioritize stories

→Implementation

-write unit tests first

-build simplest design to pass test

-Just in time – design and implement what you know right now

-no premature optimization

-refactor code when needed
-make the code easier to read/extend/use for future iterations
-remove duplicated code
-paired programming
-driver and navigator
-disagreements point to design problems
-results in better code, constant code review
-good habits spread
-reduces risk, i.e. collective understanding, keeps bus number of code > 1

→Evaluate and reiterate from step 1

Features:

Customer is part of team! Preferably onsite, always available and actively involved in all stages (i.e. clarifies requirements, writes and runs acceptance tests, negotiates what to do next, always evaluating)

User Stores

Can use index cards, short customer centered description that focus on "what", not "why" or "how"

-Note: use **clients language**, they test if it is complete

-Don't need all of them in the first iteration

Ex. "Create account", "Login", "Logout", "Update account", etc.

-Break stories into **tasks**

Acceptance Tests

Customer Tests: client describes how user stores will be tested (concrete data examples, multiple user stores, e.g. "if I logout, I should…")

Automated Tests: Customer provides XML table, tests run after all releases.

Note: why write tests first?

-Clarifies the task at hand, think in concrete tasks

-forces simplicity, you're only goal is to pass the test

-tests are useful documentation, exposes intent of programmer

-makes you think about testability early

What's different?

Every programmer participates in all critical activities; No complete up-front analysis and design; start quick and build out; decisions made throughout process; Develop infrastructure and frameworks as you develop your app

When/when not to use: Use for: dynamic project done in small teams (2-10 people); requirements prone to changes*;* customer is available **--** Don't use: cost of late changes is very high; ustomer is not available (e.g. space probe)

This class: "Extreme Classical": Waterfall, documentation early; XP: user stores, iteration, test drive development, pair programming

Unified Modeling Language (UML or Union of All M. Lang.)

Designed by committee, many groups participated. Combination of OO programing from Booch, Runbagh (OML), and Jacobsen (OOSE).

Modeling: describing a system at a high level of abstraction, many notations over time (state machines, entity-relationship diagrams, dataflow diagrams)

Class Diagrams – describes class/entity (in the OO Sense), as well as what it interacts with (but not specifics)

Associations – capture n-m relationships (like entity-relationship diagrams (databases)

Aggregation – to show a 'contains a' relationship, denoted by a diamond on the 'whole' side

Generalization/Inheritance - inheritance between classes, denoted by open triangle on super class

Object Diagram – an instantiation of a class diagram, represents static structure of system at **one specific time**

Sequence Diagram – a table, columns are classes or actors, rows are steps in time, entries show control/data flow (method invocations, state changes)

Notes: refine use cases, gives dynamic view of behavior, not orthogonal to other diagrams, overlapping functionality (true for all UML)

Advantages to UML: common language, visual syntax is useful, easier to grasp, forces clarity (precise), commercial tools to support (natural language can't have this)

Disadvantages to UML: a lot of ideas in one, some sublanguages aren't included, sometimes better to pick a subset; visual syntax does not always scale well (details are hard to depict visually, hard to understand a thousand pictures)

Interviewing – sit with clients/user, ask question, focus on what they do say and what they do not say

Master-Apprentice – have them teach you what they do, go to workplace, observe task and repeat yourself

Note: always get copies of reports, logs, emails on process to help support, fill in , or contradict what the user says

Disadvantages: user may not have vocab to explain process, no understanding of what is feasible with an application/pro gram

Straw Men – sketch of product for user that includes storyboard, flowcharts, mockups, major events, etc. (i.e. anything that conveys ideas without writing code)

Specification Views*:*

Developers View: detailed enough to be implementable, unambiguous

Client/users View: comprehensible, not too technical

Legal View: spec can be a contract, should include acceptance criteria

Informal Specs – written in natural langue, inherently suffer from some problems (e.g. missing something, ambiguities, contradictions, etc.). Why do we use it? – it is of common language (fairly understandable), and not as time consuming to create, but not the best.

Semi-Formal – more precision than natural language where desired, usually involves 'boxes and arrows'. More common and useful in real life.

Web App Architecture

Web is client/server architecture, fundamentally request/reply (not so true for 2.0)

SaaS – Software as a Service

Quiz Answers – backend is architectural component of web apps; HTTP is a communication protocol; status code 500 in HTTP means internal server error; Status code 304 means page has not modified, can used cached page; web server sends event notifications to clients in response to client requests; Apache is a web server; Load balancer is used at the server; a cache is used at the browser; JavaScript is a language unrelated to Java;

HTTP – web transport protocol specifying requests and responses (built on TCP); by default uses port 80 on the server (most reliable through proxies and firewalls), also HTTPS which ensures authenticity; requires paid certificate; only the client can initiate communication, server sends data as a response, client pulls

periodically
URI – unique resource Identifier, ex.
http://server.com:8888/path/file?p1=v2&p2=v2#x
-protocol (scheme): http, https
-host: server.com
-port: 8888, remember default is 80
-path: /path/file
-Query: p1=v1?p2=v2
-Anchor: # x (optional)
Elements are URL encoded, ex. %245%20for%20me ($5 for me)
State Example – web servers often do not keep the state of users as it requires it to keep connection open for all users (lots of resources used up), hard to distinguishing reliabl
y different clients and different requests; backend might store state, use cookies!
Cookies – key-value pars associated with a web site, stored at the browser for a specified time (every subsequent visit to the same site, by the same browser will send the cookies)
Note: many security issues if used incorrectly
Summary: web apps involved many moving parts: HTTP servers, HTTP protocol, caches, HTTP clients, databases, etc.; convenient to use frameworks that bring these pieces together (Rails, Django, etc.)

Software Architecture

MVC Architecture – a method for separating the user interface application from its domain login
Main goal – separate the model from the UI (view/controller);
Model - functional core of the application, aka business /domain logic;
encapsulates state of application (e.g. database); if model is correct, data will continue to make sense; INDEPENDENT OF VIEW (cannot hold direct references to the view or controller); independence gives it flexibility, robustness, testability, reusability, etc.; active – model defines a change notification mechanism that can notify views/controllers (if they are listening for it), common for desktop GUI apps; passive – model is called by view/controller to get data (common in web UI), puts burden on controller.
View – what to show and how to show it (based on state of model); informs controller of user actions; has access to read model, but should not be able to change it; for web app view is HTML, JS
Controller – view deals with "output", controller deals with "input"; controller decides what model actions are needed, selects next view, allows the user to drive the app; in web app, controller is handler for incoming HTTP requests; in other apps, controller may be minimal or generic
MVC Advantages – separates and modularize concerns; separations between data layer and interface is key (view is replaced/expanded; model data changes are reflected in all views; better scalability; distribution over network is simplified)
MVC Disadvantages – view and controller depend heavily on API's; close coupling between view and controller (can be hard to separate); more complex (but may payoff later)
Architectural components – computation elements; where the 'work' happens; (model, view, and controller); NOTE; boxes in above picture
Architectural connectors – communication elements (e.g. API's), enable communication between components; may be explicit (procedure call) or implicit (event); NOTE: arrows in above picture
Architectural Configurations – arrangements of components and connectors to form an architecture (e.g. MVC, Pipe and filer, layered systems, event based systems)

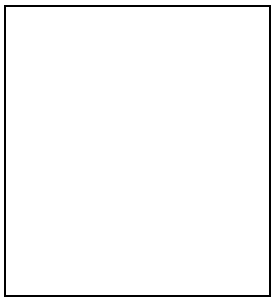
Pipe-and-Filter – consistent of components (filters) that transform/filter data, before passing it on via connectors (pipes) to other components.
Examples: Unix programs, Compilers.
Advantages : overall behavior can be understood as a simple composition of the behaviors of individual filters; support reuse; modular testing; easy maintenance and enhancement (new filters can replace old)
Disadvantages: not good for interactive applications (e.g. incremental update of output); for performance we need tighter coupling of filters (e.g. the code generator of a compiler might need some info from the type checker); extra overhead to parse/unparsed data to external formal(unless you pass data structures)
Layered Architectures – each layer provides a higher level of abstraction to the layer above (e.g. Operating Systems!);
Advantages - support increasing levels of abstraction during design; support enhancement (change in one layer affects at most two layers); reusable (can define layer interfaces, interchange implementations of same interface);
Disadvantages - may be hard to identify layers; what if one layer wants to communicate with a non-adjacent layer
Event based (implicit invocation) – independent reactive objects/processe s handle events (e.g. DBMS, GUI's)
Advantages – announcers of events do not need to know who will handle the event; supports re-use, supports evolution of systems (easy to add new agents)
Disadvantages – components have no control over ordering of computations, or when events are finished; hard to understand the control flow;
Summary – architecture is what enables us to 'scale up' our software to handle larger applications; matching the right architecture to your applications is crucial

Design Patterns

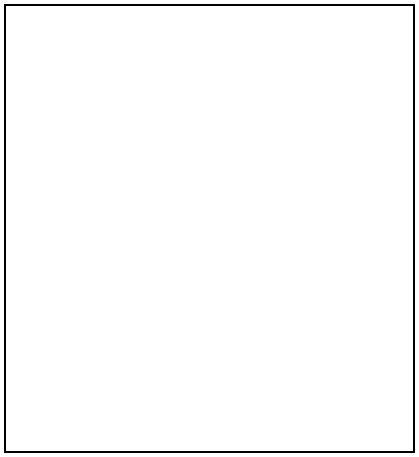
Descriptions of communication objects and classes that are customized to solve a general design problem in a particular context;
Importance: they name common, successful ways of building software; much easer to learn the architecture of a system from descriptions of design patterns over reading code; members of a team have a ay to name and discuss elements of their design
Elements – 1. Pattern name (useful); 2. Problem it solves (when to apply the pattern) 3. Solution (participants and their relationships) 4. Consequences (costs of applying the pattern, space/time tradeoffs);
Composite pattern (right) – easy to add new object types; clean interface, elegant code; BUT have to fight corner cases; hard to enforce certain characteristics;

Decorators – allows use to add decorations (e.g. Borders, scroll bars, menu's) that we can mix independently; write n decorates and can created 2^n combinations; allows us to add features to an object
Factories – allows use to create look and feel dependent objects;
Bridge pattern – allows us to use multiple windows;

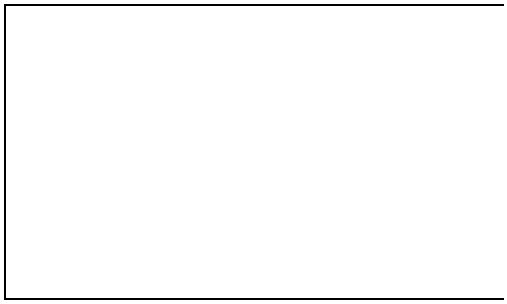
Iterators – work well for traversing over data if we do not need to know the type of elements being iterated over (e.g. send kill message to all processes); not good for spell- checking;
Visitor pattern – allow traversal and type-specific actions on elements; have a do it method for each element type; very powerful; whenever you have a hierarchical structure of heterogeneous elements, write support for visitors (do nothing visitor);



Source Configuration Management (SCM)
Allows us to recreate project history; revert to an older version (discard changes, investigate bugs); compare versions; find edits;
Repository – stores all version of project
Checkout (svn co) – copies current version of project from Repo to a local copy (aka 'working copy') (not
e: editing is always done on LC)
Commit – stores snapshot of LC in repo (stores date, author comments, creates a new revision)
Inspect changes: svn diff –r 12:13, svn diff –r 13 (or svn diff)
Merging – always down in LC; syntactic; conflicts mean two users changed the same lines, doesn't know what do; note lack of conflicts does not mean changes work together;
Branches – allow different users to work on different tasks; maintains an always working copy in master; custom releases
XML (eXtensible Markup Language)
Adds lightweight semantic info to plain text; can describe complex structured data; really just data
DTD: Document Type Definition
XPath – compact syntax for grabbing fragments of XML data; similar to regular expressions
XSLT (eXtensible Style sheet language transformations) – programming language for transforming XML
Markup languages – give structure and meaning to plain text; lightweight overlay (erase and you have plain text); vocab agreed upon by users;
HTML is a markup language, good for webpages; but you can always make your own!
Notes; -'<' and '>' become '<' and '&gt', '&' becomes '&amp'; elements are strictly nested and explicitly closed!
Testing
Errors are inherit in any engineering discipline, thus ALWAYS TEST.
Testing: exercising software to try and generate failures
Static Verification; identify problems by looking at code, resolve issues *Approaches to verification* – inspection walkthrough, i.e. manual review; formal proof, i.e. proving program does what it is meant to do
Note: Today QA is mostly testing
Levels of testing – unit testing is verification of single modules; *integration testing* is verification of interactions among modules; *system testing* – testing the whole system; *acceptance testing* – validation of the software against user requirements; *regression testing* – testing of new versions of software against old tests (finding bugs, creating test to check for bug and always running new versions against), should be automatic, and ensures forward progress, but it is $$$$
Test Driven Development
A strategy for interleaving code development with automated test development (i.e. try out the code while writing it, but automate; write test while writing features, or tests first)
Steps – 1. Write one or more test that do not work 2. Write code to make tests compile and run 3. Debug and fix current test (Quickest first) 4. Check all other tests still work 5. Refactor code and repeat
Tests first – helps achieve use-driven-api; faster debugging
Summary – TDD results in fewer bugs and more maintainable code; fits well with agile process; mix coding with testing
-*Design/implementation driven*: One or more tests for each method; careful with corner cases
-*Bug driven*: Write a system test that reproduces the bug } Repeat until all tests pass 1. Investigate the bug and narrow down the cause 2. Write "smaller" test that reproduces the bug 3. Fix code to pass the smallest test 4. Try to run the larger tests
Regression Testing
Recall: Always run all tests when code changes; Regression testing ensures forward progress; We never go back to old bugs ; Easy to diagnose the test failure when we know which revision breaks the tests ; Continuous integration systems; Attempt to run the regression tests on every commit (or at least with high frequency)
Software Dilemmas
Nearly half of all software development projects are late, and one-third are over budget.
Code Coverage
Structure Coverage Testing – code that has never been executed likely has bugs; at least the test suite is not complete; this leads to the idea of ***statement code coverage***; divide program into elements (e.g. statements) and the cover rate of a test suite = (# of elements executed by the suite/ # elements in program)
-Microsoft reports 80-90% code coverage; safety critical software must get 100% statement c
overage; average is about 85%
Branch Coverage = # execute branches / # branches in program
(left has two branches (4-7); branch is executed when ***both*** outcomes have been executed, aka multiple condition coverage



Data Flow Coverage – an untested flow of data from an assignment to a use of the assigned value; could hide erroneous computation; even if we have 100% statement
and branch coverage, their can still exist faults; = # of executed def-use paris/ #of def-use pairs in program
Path Coverage – programs can have an exponential number of paths, many are not reachable



Code Coverage Limitations – 100% code coverage does not mean no bugs, many bugs lurk in corner cases, also 100% is almost never achieved (products have shipped with < 60% coverage), and high coverage may not even be economically desirable.
Benefits – helps identify weak test suites and where more tests are needed, in practice statement and condition coverage is useful; control flow heuristics may be useful
Budget Coverage – testing is done when money is gone or time is up; test selection is important
Summary – you can never test completely, important to use a principled way to know when you have tested enough and what parts are more important for testing