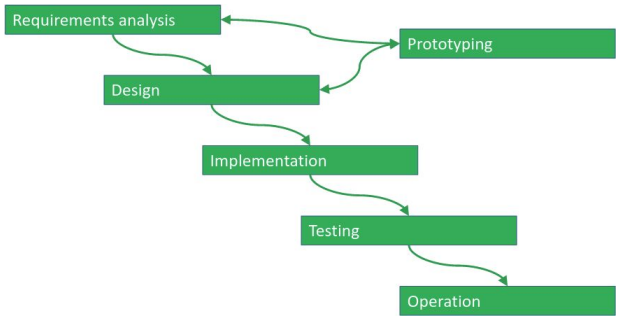
Software (System) Development Lifecycle (SDLC):
requirements>design>implementation>testing>maintenance
(Most time spent on maintenance - features, performance…)
***Quality attributes* of great software:**
- Reliability: will it perform properly in assumed conditions?
- Efficiency(+performance): can sys respond & work fast, scale to high loads?
- Integrity(+security): possible to put sys in bad state?
- Usability: can real users complete goals w/ the sys?
- Maintainability(+modifiability): how hard to make changes?
- Testability: can semi-auto test if sys is right?
- Flexibility(+robustness): easy to adapt to unusual conditions?
- Portability: could sys be ported to new platform?
- Reusability: what parts could be used in new sys?
- Interoperability: sys talks to other relevant systs?

Use cases: activities a system supports
Entities: kinds of objects involved in use cases
Attributes: properties of entities
Sys boundary: part of the sys that will be built
Process: set of ordered tasks

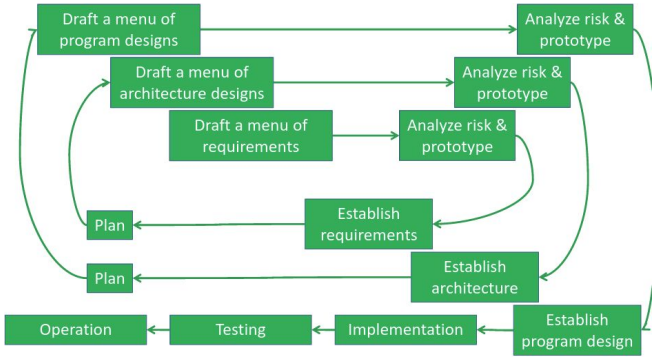
- Requirements: What should the system do?
  - Functional requirements: describe *what* sys should do
    - Unstructured text:
      - Req. definition (external viewpoint): sys is a black box w/ some interface -- emphasis on *sys's role*
      - Req. specification (sys viewpoint): env is accessed via in/outputs -- emphasis on *how sys works*
    - Structured use cases:
      - Use case name: *succinct & meaningful*
      - Actor: *who 'does' the activity?*
      - Preconds: *what is true before activity?*
      - Postconds: *what is true after activity?*
      - Flow of events: *what steps do actor & sys perform during the scenario?*
  - Non-fcnal reqs: describe *how well* sys should do stuff
    - Can be written as unstructured text
    - Often written in terms of *fit criteria* (how good does sys need to be; tightly related to important qual attributes; shouldn't be 'imagined' but driven by cust needs
  - **Prototyping:** *depict* what sys should look like, *test* prots w/ custs or (pref) users, *fix up* prots & use learned to impl. real sys
    - Throwaway prots: paper, low-fi (impl. w/ tool like photoshop)
    - Evolutionary: hi-fi (impl. on target platform - not fully functional, but will be incorp'd into final product)
    - Testing: *let UI speak for itself*; if misunderstood, fix on spot if poss (user is always right)
  - Stakeholder review: engrs present understanding of reqs, SHs correct, everyone discusses, engrs revise reqs, repeat if necessary
  - Manual analysis: syst'ly check general consistency (if unst. text says sys should support X, is this cons. w/ what UCs are saying; are necessary ents shown in UML and/or MSD)
  - Formal analysis: check if reqs are *provably* consistent
  - ***Paper/low/hi prots, SH rev, man. analys. all especially good for validation (is goal correct), all those + formal analysis good for verification (is solution correct)**
- Design: How should it do it?
  - Architectural design (overall *structure* of the system):
    - What components should be in the system?
    - How should the components be connected?
  - Program design (how *code* should be organized):
    - How should each component's code be distributed among classes and/or functions?
- Implementation: Writing code
  - May include writing comments, writing other documentation, helping other engrs w/ coding, answering questions, reading colleagues' code/docs, messing around w/ code
- Testing: Making sure code is right
  - Unit testing (automatically checking indiv. components)
  - Sys integration testing (components work well together)
  - Usability testing (checking user interfaces)
  - Acceptance testing (checking that cust/user is happy)
- Operation: Using the system
  - Dist. code to custs/users
  - Providing docs & support
  - Debugging after users try out the system
  - Studying how well sys works in practice
  - Adapting sys for new markets

Waterfall: Good for small systems whose requirements can be fully understood before any design & coding



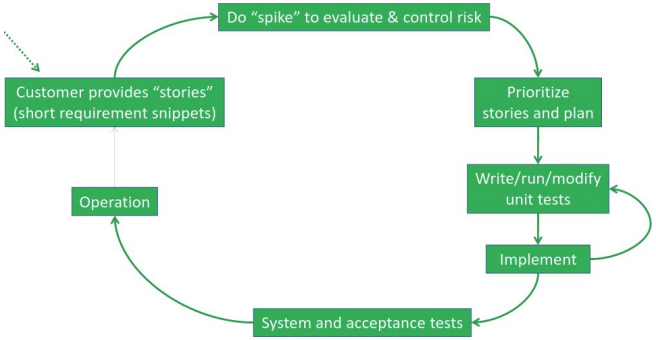
- No prototyping in pure waterfall
- Goes through SDLC sequentially (not realistic for large systems)
- Lengthy documentation of requirements
- Drawbacks:
  - Non-iterative: hard to handle changes during development
    - Views sw dev as manufacturing rather than creative
    - Long wait for final product

Spiral: Good for larger systems w/ vague requirements & many alternatives for designing & coding



- Developed for more structured feedback

Agile: Good for systems where you can rapidly create something very small but useful & then expand from there, *coding focused*



- Not usually that tidy
- Iterations:
  - Grouping: not all itrs result in a new product release
  - Sub-dividing: each itr has 'micro-itrs' inside
- Examples:
  - Extreme Programming (XP): 1-2 wk itr
  - Scrum: 30-day itr; mult. Self-organizing teams; daily 'scrum' coordination
  - Chrystal: collection of approaches based on notion that every project has unique needs
- **XP:**
  - Communication: good to talk w/ cust & btw devs
  - Simplicity: keep simple and grow sys and models when needed
  - Feedback: let users give feedback early and often
  - Courage: speak the truth, with respect
  - Practices: *customer*: cust is part of team, participates in testing; *realism*: realistic about meeting cust needs, meet needs in small increments, sustainable pace (no all-nighters); *design:* simple design, design improvement, around coherent idea, continuous integration to see if sys is on track; *teamwork*: pair programming, test-driven development, collective code ownership, coding standards
  - User stories: flexible, fuzzy, minimalist, backed by acceptance tests
    - Systematically break down each into tasks required
    - Estimate effort for each task (use spike (experimental implementation) if needed
    - Cust prioritizes (chooses which story for next itr)
  - Acceptance tests: may be mult for each story, ea validates a part of a story by exercising the sys how a user would, tell what cust will use to judge success, let you know when you can stop dev'ing
    - Ideally written by cust, should be precise & unambiguous
    - Run ≥ 1 per week, preferably more
    - Useful to have integ machine where running ATs (representative of what cust would have)
  - Pair prog'ing: *driver* controls keyboard, is talking to copilot, explaining intent of code and where s/he is going; *copilot* watches for mistakes, offers ideas, can request to drive
    - Must say yes if someone asks to pair and you can, pick standard as team, take turns with diff pair combos, code written alone *must* be rewritten
  - Models:
    - **NOT true:** mod=docs, mod'ing implies heavyweight process, mod'ing freezes reqs, mods never change, mods are complete, mods must be created w/ a tool, all devs know how to mod right, mod'ing is a waste of time, data mod is only one that matters
    - Don't let mods distract from hunt, mod itr'ly and incr'ly, mod w/ other ppl, make simple content, use simplest tools
  - **Cost, schedule, quality:** *pick two*
    - Activity graph: shows dependencies of proj's acts (filled circs for start and fin, 1 circ per milestone, labeled arrs for acts)
    - Effort: PSP one way to record effort (size of compo, times taken, refer when making future predictions), can be done @ team lvl
      - *Estimation models*: algorithmic (eg COCOMO) - inp=desc of proj&team, outp=est of effort; machine learning (eg CBR) - gather descs of old projs&time taken, run prog that creates model (now have cust algorithmic model)
    - Schedule: sort milestones in topo order, compute soonest each can be reached from imm dependencies
    - Risk: impact=loss if risk->prob, likelihood=probability risk->prob, control=degree to which can *reduce exposure*
    - Slack time (latest-earliest poss start time): good for risky acts; acts on crit path never have slack time
  - **Testing:** focused on functional correctness (ever right?) and reliability (usually right?)
    - *Test (case)*: one exec of prog that may expose a bug
    - *Test suite*: set of test cases
    - *Tester*: program that generates tests
    - Black box testing: doesn't look at code or intern. structure, sends inputs and observes outputs; abstracts away internals
    - White box: opens box, use src code to design test cases
    - Unit: 1st phase by devs of modules

    - Integration: combines unit-tested modules
    - System: tests whole program (focuses on breaking the sys)
    - Acceptance: by users to see if sys meets use requirements
    - Regression(good at all times): changes can break code, bring back old bugs

Contrasting processes:

	Waterfall	Spiral	Agile
Emphasizes:	-Simplicity -Traceability	-Risk management -Exploring alternatives	-Flexibility -Immediacy
Weakness:	Requirement/design mistakes can be costly	Exploring alternatives can be costly	Continual rework can be costly
Style:	-Highly controlled -High ceremony	-Moderately controlled -Moderate ceremony	-Rapid & organic -Low ceremony

Some definitions
-"traceability": relationships between requirements and system elements are documented
-"immediacy": getting some sort of working system to the customer as fast as possible
-"rework": redesigning the architecture and/or refactoring the program code
-"controlled": conformance to process is highly valued, even if it slows a project down
-"ceremony": how much analysis, documentation, and planning is involved

Improving the system later: for use w/ any process
- Iterative: Get whole system working *pretty* well then add features throughout system
    - Good if you need to implement most of a sys before you can get much value
- Incremental: Get part of the system working *really* well then add more parts to the system
    - Good if most of a sys's value is concentrated in a small number of components
- Agile rules of the simplest design (with precedence):
    - Sys (code+tests) must communicate everything you want to
    - Sys must contain no duplicate code
    - Sys should have fewest possible classes
    - Sys should have fewest possible methods
- Refactoring: program transformation that *improves code's organization, not function*
    - Split long methods (≥1 screen) into smaller ones; move often duplicated code (≥3 dups) into a method; break large classes (≥7 member vars and/or ≥50 methods) into pieces, using an appropriate design pattern; etc (rename, delete unused methods, move, introduce factory, change signature)
    - Must have a working unit test suite, then talk with pair prog'er about it, try it, run unit test again, itr until code is better and unit tests passed

Diagrams:
- Use case: stick man for user, oval for UCs (*italicize* abstract UCs), -➤ when a UC calls another, -▷ for specialization
- UML class: one box per kind of entity, listing attributes (*italicize* abstract), lines w/o arrows show references (labeled w/ cardinality), -▷ for specialization (points towards more general), -➤ indicate dependencies (usually omitted in reqs' class diagrams)
- ERD: one box per kind of entity, list ents on branches, lines w/ ◇ show relationships (◇ label indicates role) #s or vars on lines show cardinality
- Dataflow: ovals are 'functions' provided by sys (in arrow=param(labeled), out arrow=output(labeled)), rectangles are actors, 'half-rectangles' (二) =datastore
    - Often clearer w/ separate dataflow for each UC
- Message seq: 1 box/entity involved, ea. box has --- showing lifetime (which can end if an obj is destroyed), arrows show messages, draw arrow back if return val, conditionals are written in [ ] (loops can be in shaded box)
- State charts: 1 box/state, arrows show poss. state transition (annotated for what conds the trans occurs), filled circle shows start, nested filled circle shows stop

Architecture: High level block diagram of system components
- Architecture: shows pieces of a sys & their relationships
- Component: self-contained piece of sys w/ defined interfaces
- Connector: linkage btw components via an interface
- Certain archs occur a lot (common kinds of comps & conns, typical arrangements): client-serv, p2p, pub-sub, pipe&filter, repo, layering, etc.
- Server: component that provides services
- Client: component that interacts w/ user and calls server
- Peer: compo that provides services and may signal other peers
- Publish: a compo advertises that it can send certain events
- Subscribe: compo registers to receive certain events
- Classic repo: cli-serv design w/ services for storing/accessing data
- Blackboard repo: pub-sub; compos wait for data to arrive on repo, then compute and store more data
- *All repo systs are cli-serv systs that can also store data*
- Filter: compo that transforms data
- Pipe: conn that passes data btw filters
- Layering: compo that provides services to next layer; compos 'hide' lower layers
- **Decomposition:** providing detailed view of a component
    - Functional: break reqs into fcns and fcns into sub-fcns (each fcn *computationally combines* output of sub-fcns)
    - Data-oriented: identify structures in reqs, break structs down (one compo/data structure; each struct *contains part of data*)
    - OO: ident structs aligned w/ fcns in reqs, break down (1 class compo/data+fcn pkg; each compo contains *part of data+fcns*)
    - Process-oriented: break reqs into steps into substeps (1 compo/sub-step; ea. substep compl. *1 part of a task*)
        - Definitely involved in pipe+filter architecture
    - Feature-oriented: break reqs into services into feats (1 compo/srvc or feat; ea. feat *makes srvc 'a little better'*)
    - Event-oriented: break reqs into systs of events into subevents & state changes (ea. compo gets and sends certain events and mngs certain state changes; ea. compo like a *stateful agent*)
        - Definitely involved in pub-sub architecture
- Eval'ing arch designs: compare against desired *qual attribs*, check for problematic *failure modes* or walk through *use cases*
    - Goal to ident room for improvement, not prove it's perfect

OO design: all code should have a purpose
- Module (class, pkg, compo, etc): should be put with related data/code
- Coupling (reduces maintainability): If A&B coupled, modding A may require modding B
    - Content coupling (**worst**): A mods B
    - Common: A&B both r/w same data
    - Control: A calls B
    - Stamp: A provides structured data to B
    - Data: A provides unstructured data to B
    - Uncoupled (**best**): none of the above
- Cohesion (increases maintainability): If A highly cohesive, easy to find code for a concern
    - Functional/informational (**best**): A&B work together for 1 purp
    - Communicational: A&B use same data
    - Procedural: A execs, then B execs, & A&B are vaguely related
    - Temporal: A execs, then B execs, but not related
    - Logical: Either A or B might be executed
    - Coincidental (**worst**): none of the above
- **Tips:**
    - Law of Demeter: code in one module shouldn't talk to children of another module
    - Move code to where it's used (change module boundaries so fewer lines cross boundaries)
    - Split modules to reduce module cycles
    - Prefer composition over inheritance (combining submodules instead of 1 compo/module/class specializing another)
    - Communicate through interfaces when feasible (polymorphism)
- **Design patterns:** help main, flex, other qual attrs; help sys desrs
    - Builder: knows how to create complex obj; use when instantiating an obj requires filling it w/ parts or lengthy config
    - Adapter: converts one interface to another by wrapping; use to overcome incompatibility
    - Facade: provides unified, hi-lvl interface to subsys; use when calling subsys needs frequent series of complex code
    - Memento: encapsulate state in an obj; use if might want to return to certain state later
    - Interpreter: parses and acts on instrs written in certain syntax; use to add scriptability
    - Observer: watches for other obj to change state; use in any event-driven design
    - Template method:reduce amount of duplicate code among subclasses of same parent class; use when mult subclasses have similar (not identical) impls of the same method
    - Factory method: encapsulates code that creates objs
    - Strategy: allows algs to be selected at runtime; use when best alg is not known until app is running
    - Decorator: extend obj's functionality at runtime; flexible alt to inheritance @ design time to create subclasses that support new features
    - Composite: collection of objs that represents a composite entity; client modules deal only w/ new interface, don't need to know how compobj's data nodes are structured
    - Visitor: collects and encapsulates oper frags into own classes; use when prog needs to itr over existing data struct and do some operation on each item in the structure

Professionalism:
- Key assumptions: future will be like the past, you are reflective enough to pay attention to self and actions, disciplined enough to take time to *record and use data*
- Personal SW process: PSP is one way of collecting and using data, but agile doesn't prescribe any particular method (should be simple and not time-consuming)
- Discussions for ends of itrs: interesting coding techns discovered, new IDE features, key problems encountered, one teammate really good/bad at something (for future pairings)
- Traits of a professional: varied activities req'ing special skills, personal standards of excellence, society-centric motivation, giving back to society
- About being *ethical*: breaking the *law* can earn *fine or jail time*, breaking a *moral* can ruin your *reputation*, breaking an *ethic* can ruin your *conscience*
- 8 principles of IEEE Code of Ethics: act in public interest, act in interest of clients and employers, produce quality products, maintain independent judgement, manage ethically, protect integrity of profession, support colleagues, pursue lifelong learning