CS189A Capstone
Fall 2023

Lecture 2: Introduction to Software Engineering and Software Process

Course Work

- In 189A, in addition to building a prototype for the project you will also prepare the requirements and design specifications for the project
  - All project artifacts by all teams will be accessible by the class
- There will be no homeworks, midterm or final
  - So that you can devote a lot of time and energy to the project!
- You will be graded on the project deliverables, your performance in presentations and discussions will be evaluated
- There will be reading assignments

Course work

- Team Formation: Forming a team and claiming a project
- Vision Statement: Each team must prepare a 2+ page vision statement about the project describing
  - what the project is about
  - what will be the outcome of the project
  - what will be the implementation platform
- Product Requirements and Design (PRD) document: two deliverables
  - use cases, system architecture
- Presentation and Demo
  - project presentation
  - demo of the prototype

The Structure of the Class

- The class will be organized as follows
  - Monday: Instructor discusses a general software engineering topic followed by meetings with teams to learn about their progress
  - Tuesday discussions: TAs discuss a tool or a topic, followed by discussions/meetings with teams
- We will use agile software development
  - You will do 4 2-week sprints

Course Goals

- To learn the issues and problems involved in large software projects
- To learn phases of software development and evolution: requirements analysis and specification, software design and specification, implementation, testing, and maintenance
- To learn basic software engineering techniques and principles
- To gain experience in large scale software development by working on a team project

CS189A Schedule

Week 1 (Oct. 2)	project pitch meeting, team formation, project selection	
Week 2 (Oct. 9)	contact mentors, select team lead and scribe	sprint 1 starts
Week 3 (Oct. 16)	vision statement due on Oct. 17	
Week 4 (Oct. 23)		sprint 1 ends, sprint 2 starts
Week 5 (Oct. 30)	Requirements and Design PRD version 1 due Oct. 31	
Week 6 (Nov. 6)		sprint 2 ends, sprint 3 starts
Week 7 (Nov. 13)	Requirements and Design PRD version 2 due Nov. 14	
Week 8 (Nov. 20)		sprint 3 ends, sprint 4 starts
Week 9 (Nov. 27)		
Week 10 (Dec. 4)	Final presentations and demos	sprint 4 ends

Software Engineering, Software’s Chronic Crisis, Essential vs. Accidental Difficulties in Software Development

CS189A: Today

- Today:
  - Intro to software engineering and software process
  - vision statement

  - Teams
    - Identify **group leader** and **scribe**
      - **Lead:** motivator, picks up all loose ends, settles debates/makes decisions
      - **Scribe:** records scrums, retrospectives, sprint planning, mentor/TA meetings
    - Contact mentors, set up periodic meetings
    - Work on vision statement: Due Tuesday next week!

Software engineering is 55 years old!

- In 1968 a seminal NATO Conference was held in Germany

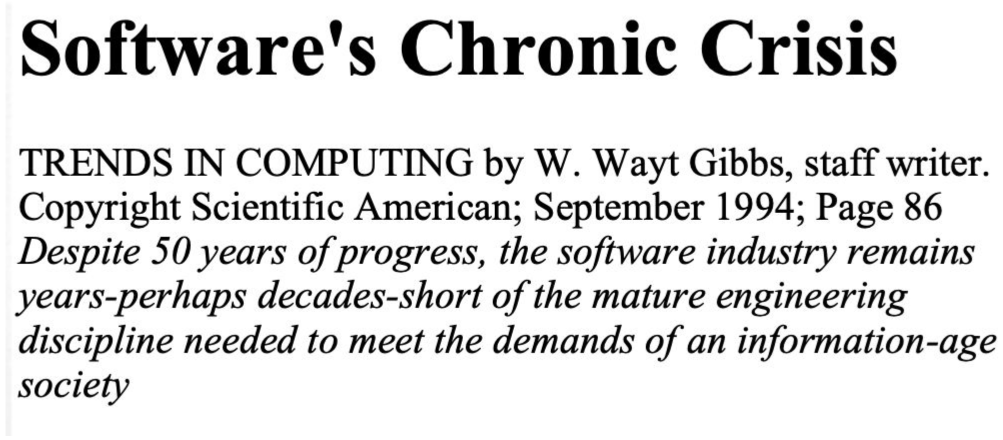
Purpose: to look for a solution to ***software crisis***

–50 top computer scientists, programmers and industry leaders got together to look for a solution to the difficulties in building large software systems

–Considered to be the birth of "***software engineering***" as a research area

Software's chronic crisis

- A quarter century later (1994) an article in Scientific American:



Software's Chronic Crisis

TRENDS IN COMPUTING by W. Wayt Gibbs, staff writer.
Copyright Scientific American; September 1994; Page 86
Despite 50 years of progress, the software industry remains years-perhaps decades-short of the mature engineering discipline needed to meet the demands of an information-age society

Software's chronic crisis

- More than another quarter century later:

- This is a photo of the navigation system of my car
  - ***It crashes and reboots while I am driving!***

Software Engineering & Software's Chronic Crises

Software' Chronic Crisis:

Software systems frequently fail dependability and security requirements

Software Engineering is hard

Both researchers and practitioners have developed various approaches and techniques to address difficulties in software engineering

We will discuss some of them in this course

Software's Chronic Crisis

Large software systems often:

- Do not provide the desired functionality
- Take too long to build
- Cost too much to build
- Require too much resources (time, space) to run
- Cannot evolve to meet changing needs

  - For every 6 large software projects that become operational, 2 of them are canceled
  - On the average software development projects overshoot their schedule by half
  - 3 quarters of the large systems do not provide required functionality

Software Failures

- There is a long list of failed software projects and software failures
- You can find a list of famous software bugs at:
  http://www5.in.tum.de/~huckle/bugse.html
- I will talk about two famous and interesting software bugs

Ariane 5 Failure

- A software bug caused European Space Agency's Ariane 5 rocket to crash 40 seconds into its first flight (***cost: half billion dollars***)

- The bug was caused because of a software component that was being reused from Ariane 4
- A software exception occurred during execution of a data conversion from 64-bit floating point to 16-bit signed integer value
  - The value was larger than 32,767, the largest integer storable in a 16 bit signed integer, and thus the conversion failed and an exception was raised by the program
- When the primary computer system failed due to this problem, the secondary system started running.
  - The secondary system was running the same software, so it failed too!

Ariane 5 Failure

- The programmers for Ariane 4 had decided that this particular velocity figure would never be large enough to raise this exception.
  - Ariane 5 was a faster rocket than Ariane 4!
- The calculation containing the bug actually served no purpose once the rocket was in the air.
  - Engineers chose long ago, in an earlier version of the Ariane rocket, to leave this function running for the first 40 seconds of flight to make it easy to restart the system in the event of a brief hold in the countdown.

- You can read the report of Ariane 5 failure at:

  http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html

Mars Pathfinder

- A few days into its mission, NASA's Mars Pathfinder computer system started rebooting itself
  - Cause: Priority inversion during preemptive priority scheduling of threads

- Priority inversion occurs when
  - a thread that has higher priority is waiting for a resource held by thread with a lower priority
- Pathfinder contained a data bus shared among multiple threads and protected by a mutex lock
- Two threads that accessed the data bus were: a high-priority bus management thread and a low-priority meteorological data gathering thread
- Yet another thread with medium-priority was a long running communications thread (which did not access the data bus)

Mars Pathfinder

- The scenario that caused the reboot was:
  - The meteorological data gathering thread accesses the bus and obtains the mutex lock
  - While the meteorological data gathering thread is accessing the bus, an interrupt causes the high-priority bus management thread to be scheduled
  - Bus management thread tries to access the bus and blocks on the mutex lock
  - Scheduler starts running the meteorological thread again
  - Before the meteorological thread finishes its task yet another interrupt occurs and the medium-priority (and long running) communications thread gets scheduled
  - At this point high-priority bus management thread is waiting for the low-priority meteorological data gathering thread, and the low-priority meteorological data gathering thread is waiting for the medium-priority communications thread
  - Since communications thread had long-running tasks, after a while a watchdog timer would go off and notice that the high-priority bus management thread has not been executed for some time and conclude that something was wrong and reboot the system

Software's Chronic Crisis

- These are not isolated incidents:
  - An IBM survey of 24 companies developing distributed systems:
    - 55% of the projects cost more than expected
    - 68% overran their schedules
    - 88% had to be substantially redesigned

Software's Chronic Crisis

- Software product size is increasing exponentially
  - faster, smaller, cheaper hardware
- Software is everywhere: from TV sets to cell-phones to watches to cars
- Marc Andreessen: "Software is Eating the World"
- Software is in safety-critical systems
  - cars, airplanes, nuclear-power plants
- We are seeing more of
  - distributed systems
  - embedded systems
  - real-time systems
    - These kinds of systems are harder to build
- Software requirements change
  - software evolves rather than being built

Summary

- Software's chronic crisis: Development of large software systems is a challenging task
  - Large software systems often: Do not provide the desired functionality; Take too long to build; Cost too much to build Require too much resources (time, space) to run; Cannot evolve to meet changing needs

- Software engineering focuses on addressing challenges that arise in development of large software systems using a systematic, disciplined, quantifiable approach

No Silver Bullet

- In 1987, in an article titled:
  “No Silver Bullet: Essence and Accidents of Software Engineering”
  Frederick P. Brooks made the argument that there is no silver bullet that can kill the werewolf software projects

- Following Brooks, let’s philosophize about software a little bit

Essence vs. Accident

- Essence vs. accident in software development
  - We can get rid of accidental difficulties in developing software
  - Getting rid of these accidental difficulties will increase productivity

- For example using a high level programming language instead of assembly language programming
  - The difficulty we remove by replacing assembly language with a high-level programming language is not an essential difficulty of software development,
    - It is an accidental difficulty brought by inadequacy of assembly language for programming

Essence vs. Accident

- Essence vs. accident in software development
  - Brooks argues that software development is inherently difficult
    - "*The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms and invocations of functions.* ***This essence is abstract in that such a conceptual construct is the same under many different representations****. ... The hard part of building software is the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation*."
    - Even if we remove all accidental difficulties which arise during the translation of this conceptual construct (design) to a representation (implementation), still at its essence software development is difficult

Inherent Difficulties in Software

- Software has the following properties in its *essence*:
  - Complexity
  - Conformity
  - Changeability
  - Invisibility
- Since these properties are not *accidental* representing software in different forms do not effect them

- The moral of the story:
  - Do not raise your hopes up for a silver bullet, there may never be a single innovation that can transform software development as electronics, transistors, integrated-circuits and VLSI transformed computer hardware

Complexity

- Software systems do not have regular structures, there are no identical parts

- Identical computations or data structures are not repeated in software

- In contrast, there is a lot of regularity in hardware
  - for example, a memory chip repeats the same basic structure millions of times

Complexity

- Software systems have a very high number of discrete states
  - Infinite if the memory is not bounded
- Elements of software interact in a non-linear fashion
- Complexity of the software increases much worse than linearly with its size

Complexity

- Consider a plane that is going into a wind-tunnel for aerodynamics tests
  - During that test it does not matter what is the fabric used for the seats of the plane, it does not even matter if the plane has seats at all!
  - Only thing that matters is the outside shape of the plane
  - This is a great abstraction provided by the physical laws and it helps mechanical engineers a great deal when they are designing planes
- Such abstractions are available in any engineering discipline that deals with real world entities
- Unfortunately, software engineers do not have the luxury of using such abstractions which follow from physical laws
  - Software engineers have to develop the abstractions themselves (without any help from the physical laws)

Conformity

- Software has to conform to its environment
  - Software conforms to hardware interfaces not the other way around
- Most of the time software systems have to interface with an existing system
- Even for a new system, the perception is that, it is easier to make software interfaces conform to other parts of the system

Changeability

- Software is easy to change, unlike hardware
- Once an Intel processor goes to the production line, the cost of replacing it is enormous (the Pentium FDIV bug in 90s cost Intel half billion dollars)
- If a software product has a bug, the cost of replacing it is not significant
  - Just ask users to update their software

Changeability is not an Advantage

- Although it sounds like, finally, software has an advantage over hardware, the effect of changeability is that there is more pressure on changing the software

- Since software is easy to change software gets changed frequently and deviates from the initial design
  - adding new features
  - supporting new hardware

Changeability

- Conformity and Changeability are two of the reasons why reusability is not very successful in software systems

- Conformity and Changeability make it difficult to develop component based software, components keep changing

Invisibility

- Software is invisible and un-visualizable
- Complete views can be incomprehensible
- Partial views can be misleading
- All views can be helpful

- Geometric abstractions are very useful in other engineering disciplines
  - Floor plan of a building helps both the architect and the client to understand and evaluate a building
- Software does not exist in physical space and, hence, does not have an inherent geometric representation

Invisibility

- Visualization tools for computer aided design are very helpful to computer engineers
  - Software tools that show the layout of the circuit (which has a two-dimensional geometric shape) makes it much easier to design a chip

- Visualization tools for software are not as successful
  - There is nothing physical to visualize, it is hard to *see* an abstract concept
  - There is no physical distance among software components that can be used in mapping software to a visual representation

Summary

- According to Brooks, there are essential difficulties in software development which prevents significant improvements in software engineering:
  - Complexity; Conformity; Changeability; Invisibility

- He argues that an order of magnitude improvement in software productivity cannot be achieved using a single technology due to these essential difficulties

Software Development Process

How Do We Build Software?

Let’s look at an example:

- Sometime ago I asked our IT folks if they can do the following:
  - Every year all the PhD students in our department fill out a progress report that is evaluated by the graduate advisors. We want to make this online.
- After I told this to our IT manager, he said “OK, let’s have a meeting so that you can explain us the functionality you want.”
- We scheduled a meeting and at the meeting we went over
  - The questions that should be in the progress report
  - Type of answers for each question (is it a text field, a date, a number, etc?)
  - What type of users will access this system (students, faculty, staff)?
  - What will be the functionality available to each user?

Requirements Analysis and Specification

- This meeting where we discussed the functionality, input and output formats, types of users, etc. is called requirements analysis
  - During requirements analysis software developers try to figure out the functionality required by the client
- After the requirements analysis all these issues can be clarified as a set of **Requirements specifications**
  - Maybe the IT folks who attended the requirements analysis meeting are not the ones who will develop the software, so the software developers will need a specification of what they are supposed to build.
- Writing precise requirements specifications can be very challenging:
  - Formal (mathematical) specifications are precise, but hard to read and write
  - English is easy to read and write, but ambiguous

Design

- After figuring out the requirements specifications, we have to build the software
- In our example, I assume that the IT folks are going to talk about the structure of this application first.
  - There will be a backend database, the users will first login using an authorization module, etc.
- Deciding on how to modularize the software is part of the **Architectural Design.**
  - It is helpful (most of the time necessary, since one may be working in a team) to document the design architecture (i.e., modules and their interfaces) before starting the implementation.
- After figuring out the modules, the next step is to figure out how to build those modules.
- **Detailed Design** involves writing a detailed description of the processing that will be done in each module before implementing it.
  - Generally written in some structured pseudo-code.

Implementation and Testing

- Finally, the IT folks are going to pick an implementation language (PHP, python, Java, etc.) and start writing code.
- This is the **Implementation** phase:
  - Implement the modules defined by the architectural design and the detailed design.

- After the implementation is finished the IT folks will need to check if the software does what it is supposed to do.
- Use a set of inputs to **Test** the program
  - When are they done with testing?
  - Can they test parts of the program in isolation?

Maintenance

- After they finished the implementation, tested it, fixed all the bugs, are they done?

- No, I (client) may say, “I would like to add a new question to the PhD progress report” or “I found a bug when I was using it” or “You know, it would be nice if we can also do the MS progress reports online” etc.
  - The difficulty of changing the program may depend on how we designed and implemented it:
    - Are the module interfaces in the program well defined? Is changing one part of the code effect all the other parts?

- This is called the **Maintenance** phase where the software is continually modified to adopt to the changing needs of the customer and the environment.

Software Process

- Then there is the question of how to organize the activities we mentioned before (requirements analysis, design, implementation, testing).

- There have been significant research on how to organize these activities
  - Waterfall model, spiral model, agile software development, extreme programming, Scrum, etc.

Summary

- Software development involves multiple activities:
  - Requirements analysis and specification
  - Architectural design, detailed design
  - Implementation
  - Testing
  - Maintenance
  - Software development process

- There is active research in all of these areas in the software engineering community

Software Process Models

Software process activities

1. Software specification
   - Customers and engineers define the software that is to be produced and any constraints/requirements on its operation
2. Software design
   - Software spec is designed and prototyped
3. Software implementation, validation, and testing
   - Software is programmed and checked to ensure that it is what the customer requires
4. Software maintenance and evolution
   - Software is maintained (bug fixes, upgrades) and modified to reflect changing customer and market requirements

SW Specification (1): Requirements Analysis and Documentation

- Discussion/debate on the functionality, input and output formats, types of users, etc. is called requirements analysis
  - Product managers and/or software developers **try** to figure out the functionality required by the client
  - Functional and non-functional requirements

SW Specification (1): Requirements Analysis and Documentation

- Discussion/debate on the functionality, input and output formats, types of users, etc. is called requirements analysis
  - Product managers and/or software developers **try** to figure out the functionality required by the client
  - Functional and non-functional requirements

- Writing precise requirements specifications can be challenging:
  - Formal (mathematical) specifications are precise, but hard to read/write
  - English is easy to read and write, but ambiguous
  - Today's solutions employ a combination of
    - **IEEE Software Requirements Specification (SRS), Product Requirements & Design (PRD) – combined with system modeling, user stories, case studies**
    - **Should be a "living document" that evolves over time**
      - **Starts with a vision statement**

(2) Software Design

- Product managers/owners do **not** develop the software
  - Software developers use requirements doc to understand what to build

- Sketch out the functionality in the requirements specification
- Model the system and its components
  - Context, interactions, structural, behavioral
  - User interfaces, user experience
  - Use cases, sequencing, event response, system state, classes/objects
- Define software architecture: **drawings, evolving docs, coding**
  - Components with interfaces (application programming interfaces: APIs)
  - High level and low level
    - Dependencies, modules, alternatives
    - Patterns
  - Prototype components -- ***mock*** out / ***simulate*** missing pieces

(3) Implementation and Testing

- Decide on technologies to incorporate/integrate/reuse
- Implement modules defined by architectural design & detailed design
  - Typically as prototypes that evolve over time into production-quality SW
- As part of prototyping and evolving testing happens **concurrently**
  - That requirements are met, assumptions are held, bugs are minimized
  - Be **defensive**! Prevent cases that you haven't considered from ever executing (assert! exit! return error!)
- Use a set of inputs/actions to **test** the program
  - When are you done with testing?
  - Test parts of the program in isolation
  - Unit tests, functional tests, integration tests

Validation, Verification and Testing

- Reviews, walkthroughs, inspections
- Software testing:
  - black-box vs. white-box; functional vs. structural
  - random testing, exhaustive testing
  - domain testing, boundary conditions
  - coverage criteria: statement, branch & path coverage, condition coverage, multiple condition coverage
  - unit testing, stubs, drivers
  - integration& testing: top-down vs. bottom-up integration and testing
  - regression testing

(4) Maintenance & Evolution

- We finished implementation, tested it, fixed all the bugs, are we done?

- No, we (client) may say, “I would like to add …” or “I found a bug when I was using it” or “You know, it would be nice if we could also …” etc.
  - **Ease of changing depends on how SW is designed and implemented**

- Phase in which the software is continually modified to adapt to the changing needs of the customer and the environment
- **At some point, the software’s lifetime ends**
  - It is decommissioned, deprecated (APIs) and/or no longer supported
  - Typically this is a business decision

Software Process Models

- Stages of software engineering: **requirements specification, design, implementation, testing, maintenance**

- Software process (software life-cycle) models
  - Determine the stages (and their order)
  - Establish the transition criteria for progressing from one stage to the next

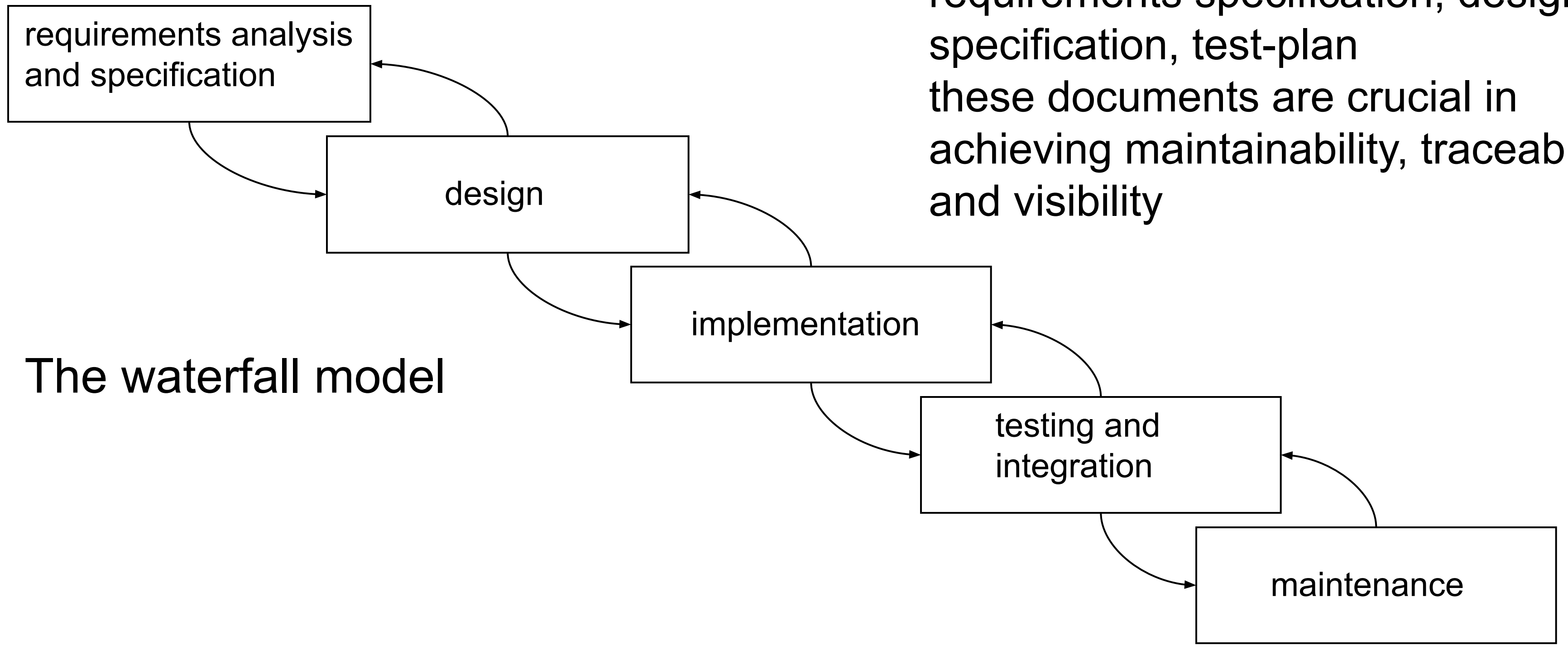
Software Process Models

- Stages of software engineering: **requirements specification, design, implementation, testing, maintenance**

- Software process (software life-cycle) models
  - Determine the stages (and their order)
  - Establish the transition criteria for progressing from one stage to the next
- Software process models answer the questions:
  - What shall we do next?
  - How long shall we continue to do it?
- Models we'll discuss: **waterfall, spiral, evolutionary: agile/extreme**
  - Waterfall (70s, 80s) when all software was "shrink wrapped and shipped"
  - Spiral (late 80s) risk-driven and iterative; Rational Unified Process (UP or RUP)
  - Evolutionary (late 90s, early 00s) as SW becomes increasingly online

Waterfall Model

UC Santa Barbara



Document-driven

requirements specification, design specification, test-plan

these documents are crucial in achieving maintainability, traceability and visibility

The waterfall model

Software product is not only the executable file:
source code, test data, user manual, requirements specification, design specification

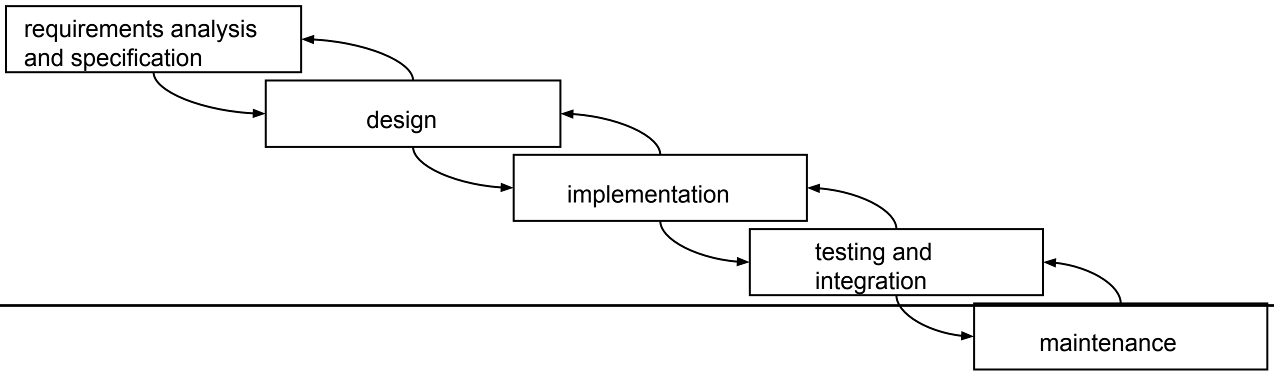
Waterfall Model

Problems with waterfall model

– Because of the restricted feedback loops, waterfall model is essentially sequential
  - for example, requirements must be stated completely before implementation starts
  - it is often difficult for the customer to state all requirements explicitly
  - hard to handle changes in the requirements

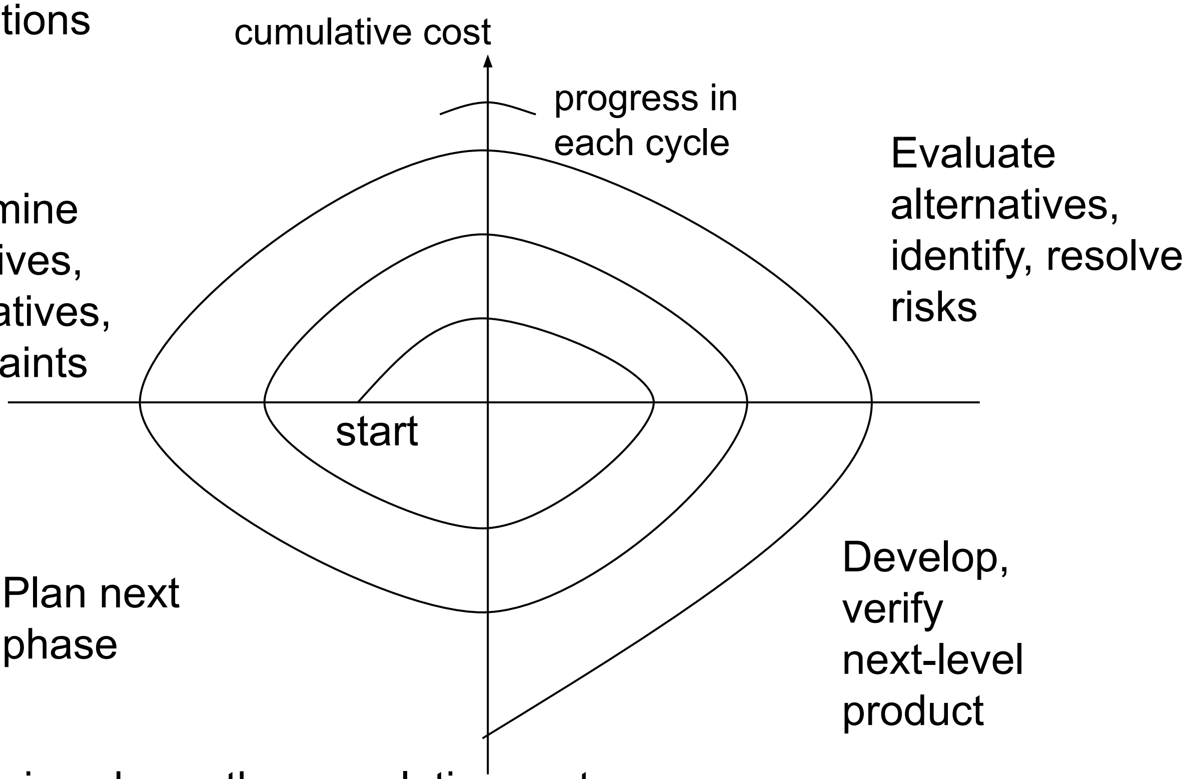
– A working model of the software is not available until late in the project life-span
  - an undetected mistake can be very costly to fix
  - the delivered program may not meet the customer's needs

– For interactive, end-user applications, document-driven approach may not work
  - for example, it is hard to document a GUI



Spiral Model (late 80s origin)

Risk driven, iterative
BUT: software delivered only after many iterations

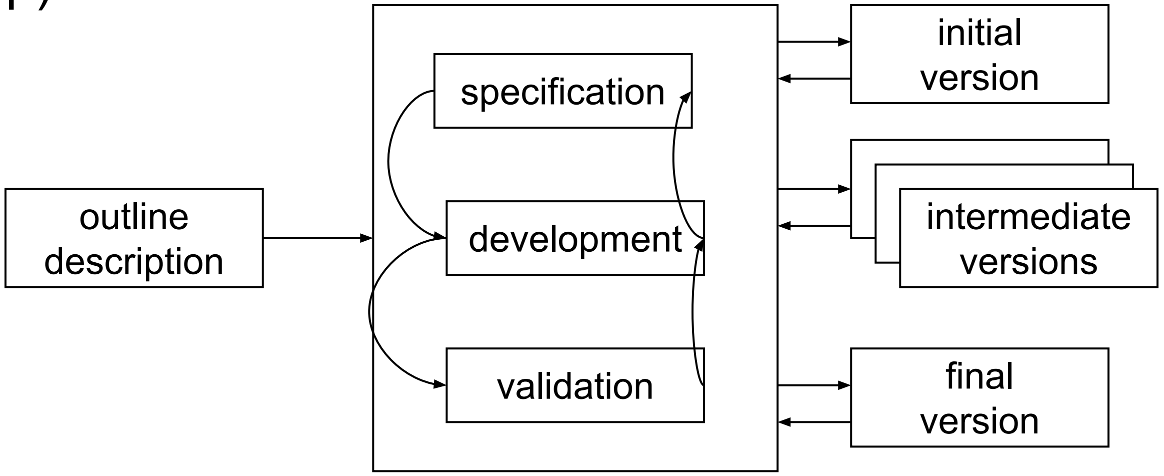

cumulative cost
progress in each cycle
Determine objectives, alternatives, constraints
Evaluate alternatives, identify, resolve risks
start
Plan next phase
Develop, verify next-level product

radial dimension shows the cumulative cost
angular dimension shows the progress in each cycle

Attack the highest risk part (usually obtaining proper user requirements) of the project first, iterate over next highest risk sub-problem

Evolutionary Software Development

- Software is built iteratively and incrementally by first providing an initial version and then improving/extending it based on the user feedback until an adequate system has been developed (**late 90s, early 00s origin**)
  - Agile software development, extreme programming
  - Triggered by change in application type (consumer, phones, web, cloud)
- All activities are executed concurrently with **fast feedback among them**
- Specifics impacted by application domain and deployment strategy (e.g. cloud/SaaS, web app)



Agile Software Development

Manifesto for Agile Software Development (2001)

available at: http://agilemanifesto.org/

“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to 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***

That is, while there is value in the items on the right, we value the items on the left more”

Principles of Agile Software Development

- Our highest priority is to **satisfy the customer** through early and **continuous delivery** of valuable software.
- Welcome changing requirements, even late in development. Agile processes **harness change** for the customer's competitive advantage.
- **Deliver working software frequently**, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- **Business people and developers must work together** daily throughout the project.
- Build projects around **motivated individuals**. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.

Principles of Agile Software Development

- **Working software** is the primary **measure of progress.**
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to **maintain a constant pace indefinitely**.
- Continuous attention to **technical excellence and good design** enhances agility.
- **Simplicity** -- the art of maximizing the amount of work not done -- **is essential**.
- The best architectures, requirements, and designs emerge from **self-organizing teams.**
- At regular intervals, the team **reflects** on how to become more effective, then **tunes** and adjusts its behavior accordingly.

Extreme Programming

- Extreme programming (XP) is a type of agile software development process proposed by Kent Beck (~late 90's)
- XP follows the agile software development principles as follows
  - Software is built ***iteratively***, with ***frequent releases***
  - Each release implements the set of ***most valuable features/use-cases/stories*** that are chosen by the customer
  - Each release is implemented in a ***series of iterations***, each iteration adds more features/use-cases/stories
  - Programmers turn the stories into ***smaller-grained tasks***, which they individually accept responsibility for
  - The programmer turns a task into a set of ***test cases*** that will demonstrate that the task is finished (Test Driven Development TDD)
  - Working as ***pairs***, the programmers make the test cases run, evolving the design in the meantime to maintain the simplest possible design for the system as a whole

Let’s get back to Capstone Projects

We said that this class it about learn by doing approach.

SW Specification (1): Requirements Analysis and Documentation

- Discussion/debate on the functionality, input and output formats, types of users, etc. is called requirements analysis
  - Product managers and/or software developers **try** to figure out the functionality required by the client
  - **Functional and non-functional requirements**

- Writing precise requirements specifications can be challenging:
  - Formal (mathematical) specifications are precise, but hard to read/write
  - English is easy to read and write, but ambiguous
  - Today's solutions employ a combination of
    - IEEE Software Requirements Specification (SRS), Product Requirements & Design (PRD) – combined with system modeling, user stories, case studies
    - Should be a "living document" that evolves over time
      - **Starts with a vision statement**

2-Page Vision Statement

- PDF via email to TA
  - Project Title / Name (can change)
  - Team name, members names/emails
  - Identify the team lead and team scribe
  - Industry Partner (company and the mentor)
  - What the project is about?
    - What problem the project is solving (**what is innovation, the science, and new core technical advance**)?
    - Why the problem is important
    - How the problem is solved today (if it is)
  - Identify the outcome of the project
  - How do you plan to articulate and design a solution
    - List the implementation platform and technologies will plan to use to develop the solution
    - List initial milestones and **how** you plan to achieve them
      - Specification, design, prototyping, testing

Capstone Award Judging Criteria

- 5pt **Science**: Has the project the demonstrated application of important, interesting, or new aspects of Computer Science? (e.g. Use of machine learning, non-trivial algorithms, solid distributed system design techniques)

  5pt **Practice**: Did the project adhere to techniques that represent the state of best practice in industry throughout the development of the system (e.g. repo workflows, test-driven development, issue tracking, or use of static or dynamic analysis tools)

  5pt **Scope**: Has the team attacked a problem of significant (but appropriate) scale and complexity. Does the problem require the development of significant new code and/or the integration of complex exciting parts that are not normally made to interface to on another? Was the project able to complete the goals that it set for itself?

  5pt **Teamwork and Presentation**: Do all the members of the team contribute significantly (in their own ways)? Does the team take the process seriously and communicate effectively with one another and the mentors? Is the project presented both in written and spoken form in a way that is compelling and impressive? Has the team developed an impressive demo?

Teamwork

- Requires people skills. Unless there is some understanding of people, team will be unsuccessful and can fail.

Keys:

- Consistency
  - Team members should all be treated in a comparable way without favourites or discrimination.
- Respect
  - Different team members have different skills and these differences should be respected.
- Inclusion
  - Involve all members and ensure that everyone’s views are considered
- Honesty
  - Be honest about what is going well and what is going badly in a project

Teamwork

- Most software engineering is a group activity
  - The development schedule for most non-trivial software projects is such that they **cannot be completed by one person working alone**.
- A good group is **cohesive** and has a **team spirit**. The people involved are motivated by the success of the group as well as by their own personal goals.
- **Group interaction** is a key determinant of group performance.
- **Flexibility** in group composition is limited
  - Lead must do the best they can with available people.
- **Good communications** across team is essential for success
  - Promotes trust & understanding

CS189A F18 Next Steps

- Today
  - Intro to SWE and vision statement
  - Identify **group leader** and **scribe**
    - **Lead:** motivator, picks up all loose ends, settles debates/makes decisions
    - **Scribe:** records scrums, retrospectives, sprint planning, mentor/TA meetings
  - **Choose a team name**
- Tomorrow: Lead contacts mentors (cc team) as introduction
  - Setup weekly meeting times with team and company mentors
- This week draft/send vision statement
  - Send your draft vision statement to mentors and TAs for feedback
  - Look at the examples from prior years!
- **Vision statement due next Tuesday by end of discussion**