CS 130 Notes

Lecture Notes

4/1 - Week 1

- Shellshock bug is where someone can specify an environment variable that can be referenced from outside to run the code on the RHS of that variable.
  - Shellshock could enable an attacker to cause Bash to execute arbitrary commands and gain unauthorized access to many Internet-facing services, such as web servers, that use Bash to process requests.
  - Shellshock is a privilege escalation vulnerability which offers a way for users of a system to execute commands that should be unavailable to them. This happens through Bash's "function export" feature, whereby command scripts created in one running instance of Bash can be shared with subordinate instances.
- Meltdown and Spectre represented attacks on speculative execution. CPUs go faster by guessing with branch prediction and hoping that their predictions are right. The attacker times a piece of code to access a piece of cache that is shared with the victim.
  - Since browsers allow for multiple threads for each tab, one thread/tab can access the information that is contained in another.
- Essays due 2 days before the debate and the commentary is due 2 days after the debate.
- Hardware and firmware are hard to change while software is the part that is easy to change and is automated.
  - “Easy” if you know how.
- Software is not manufactured, and we ignore the cost that normally is incurred with mechanical or electrical projects.
- It also does not wear out and is not really affected by physical forces. The hardware may die but the software won’t.
- For hardware failure, bathtub curve shows you that initially you have a high failure rate when the initial costs and problems associated with using a device for the first time. Then that rate will go down to a local minimum and then will steadily increase as the device wears down.
- For software failure, you have the same bathtub at the beginning (since there will be quite a few bugs) with lots of spikes with bugs coming up after updates are made and software patches being released.
- In software, there are no spare parts and most of it is custom built. You cannot really make one website that you can sell and that works for every single person.
- Definition of software engineering is the establishment and use of software engineering principles in order to get economic (low cost of developing the software), sound (well tested and solid support that it will work), reliable (you can rely on it), and efficient (low

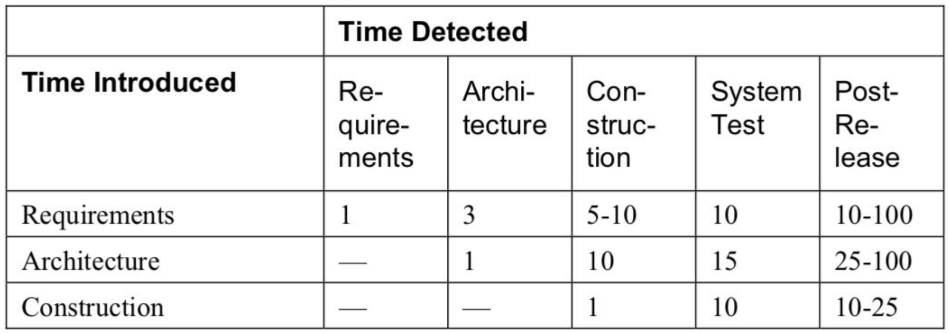
operational cost to the user) software on real machines (not talking about PDAs and turing machines, etc).
    - It's about creating and establishing an institution that is focused on making software have the aforementioned qualities.
- Software engineering is about the practical problems of producing real software while CS is about the theory and methods that underlie programs and system engineering includes software, hardware, people, processes, and networks.
- Software engineering notions
    - Software processes
    - Requirements
    - Reuse of modules
    - Security
    - Dependability.
- Software engineering goals
    - Understand the problem because often the users won't be able to give you everything.
    - Quality and Maintainability
    - Work across a wide set of domains.

4/3 - Week 1

- Requirements is figuring out what you want to do and construction is actually doing it.
- For construction, in addition to coding/debugging, we need to have:
    - Detailed design is the part where you're planning at a detailed level about what code you're going to write. You're writing the documentation for the methods.
    - Integration which is bringing the different parts of the program together.
    - Integration testing is where you run that integrated program.
    - Unit testing is where you test very small components of the program, like certain functions and methods.
    - Construction planning is where you are figuring out how is going to do what and when.
- Other components that are on the periphery of the construction are:
    - Corrective Maintenance is where you've shipped the code, people are using it, and there are still bugs in the product. This is different from debugging because the code is supposed to already be finished.
        - It is defined as any maintenance performed to return equipment to proper working order.
    - Problem definition - Figure out the problem
    - Requirements definition - Elaboration of the problem
    - Software architecture - How the software hooks together. Doesn't say what the software will do, but talks about how they will go together. Glue, not the pieces.
        - Sometimes depends on the hardware environment you're in.

    - System testing - Testing the software + hardware components. Basically the whole thing rather than just software or just certain functions in the software.
    - Upgrading deployment - Thinking about how to respond to other people upgrading components that you use and how you will push out updates.
- The prereqs before you start construction are problem/requirements definition and software architecture.
    - If you mess up or omit any of those, then you have the following costs

Table 3-1. Average Cost of Fixing Defects Based on When They're Introduced and When They're Detected



	Time Detected				
Time Introduced	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture	—	1	10	15	25-100
Construction	—	—	1	10	10-25

    - Basically, if you make a mistake with those 3 prereqs, then the cost will increase as you get further along in the project.
- When it comes to ethics, we need to define responsibilities to the public, client, employer, colleagues, the profession, and yourself.
    - And we need to balance these competing responsibilities. It's up to our independent judgement to determine when to push on one or the other.
- Hooker's Software Engineering principles
    - Provide value to users.
    - Keep It Simple Stupid
    - Architectural vision
    - Managed and understood development process
    - Be ready to change
    - Plan for reuse
    - Think before doing
- Somerville has #1,4,6 and he also has requirements of dependability and security.
- There's no one right answer in software engineering. There are lots of tradeoffs and competing situations.
- Pressman practice principles
    - Separation of concerns
    - Understand limits of the abstractions that you used.
    - Look for patterns and keep things consistent
    - Look at program/system from multiple perspectives.
    - Think about the information transfer between components of your system as well as between different people on the team.
    - Agility and adaptability

- When doing requirements engineering, we need to talk to the stakeholders and see what they want to make sure our requirements fit. All the design elements should also reflect the requirements.
- Also need to create a system model which is your model of what the problem looks like.
- Good requirements are:
  - Testable
  - Feasible
  - Don't conflict with each other
  - Able to be attributed to a particular source since it will help you resolve conflicting requirements
  - Bounded (i.e "Do X as best you can" is not bounded)
  - Essential
  - Specified in the user's terminology. Don't complicate it.
  - Match the system's vision
  - Prioritized - Important for being able to schedule the order and urgency for which things should get done.
  - Validated - Checking all of the above items
- Types of requirements. The distinction between all them are a little fuzzy.
  - User vs System
    - User requirement comes from users and is visible to the users. The audience is the users and the managers. These describe user visible services and constraints imposed by users. It refers to their needs and the types of activities they are allowed to do. User requirements talk about the problem domain, the world of the user. They describe what effects need to be achieved.
    - System requirements are where the audience is developers and operation staff. More detailed info about functions and operational constraints (which are imposed by system not by users), but end users don't need to understand. System requirements talk about the solution domain, the world of the software logic. They describe what the software must do (as opposed to the effects in the user's world that this may or may not achieve).
  - Functional vs Nonfunctional
    - Functional is just what the system does. It specifies something that a user needs to perform their work.
      - Ex) a system may be required to enter and print cost estimates.
    - Nonfunctional (quality of service requirements) are more of the big picture constraints like performance, reliability, accessibility, and security issues. Hardest to write and hardest to keep track of.
- Developing the requirements involves
  - Identification of the stakeholders - Figure out who cares about the software and determine their viewpoints and figure out where they're coming from. Find the agreements and disagreement between the different stakeholders. Ask the dumb

questions about the benefits, the overall goal, and about what the communication will look like between you and the stakeholders.

    - Discovery Elicitation - Elicitation is researching and discovering the requirements of a system from users, customers, and other stakeholders. Use a well defined procedure (can vary depending on organization) that involves interviews, meetings so that you can define the problems and elements of the solution.
        - After this stage, you have to produce a scope (basically everything you're planning to do), feasibility analysis, stakeholder list, use cases, explanation of the need and why the project is necessary.
    - Requirements Negotiation - Resolving any conflicts and remove any requirements that are too ambitious or don't fit with the rest of the requirements. Also discover and assign the priorities of the requirements.
    - Requirements Validation - Check the consistency of the requirements and make sure they are unambiguous and consistent. For this step, you can build a prototype and check to see that it is possible.
    - Requirements Management - Requirements change with time and you need to manage this. Need to set up a change-control procedure for possible changes to be brought up, and for the right stakeholders to be notified and for the right changes in the design to be made.

<u>4/5 - Week 1</u>

- Example of SRS
    - Login
        - Appearance
            - Blue
            - Giant Logo on top
    - Main Page
        - Appears after login
        - Feed
            - Status
            - Comments
- Write down what the client really wants. As an engineer you need to clarify the client's needs since they aren't as technically knowledgeable. Need to clear up the things that are technologically feasible.
- Also need to be aware of any requirements that could lead to security or data vulnerabilities.
- Not everything that the client says will be viable or realistically possible.
- SRS needs to be modular and covers the functionalities that the client wants.
- Characteristics of the requirements
    - Testable
    - Feasible
    - Don't conflict

    - Attributable

4/8 - Week 2

- In collaborative development, we want to focus on cost effective methods for defect-detection.
    - This is important because a lot of your software development process will involve finding bugs.
    - Want to catch defects that are less likely to be caught by individuals who are using the end product.
- Methods for doing collaborative development
    - Formal inspections: This is the gold standard. The goal is focus on defect detection. Don't worry about fixing the bugs, just be able to identify it for the developers to then address later. The important thing is to be formal about the process. Use a checklist that all the reviewers have to go through to make sure they take a look at how the application deals with the most likely problem areas.
        - The reviewers and the developers should engage in a meeting (with a moderator + scribe) where the review is discussed. This is a chance for the authors/developers to explain why they made certain choices. Key distinction is that management and the actual users are not involved.
        - The general plan should be to have the authors give an overview, and then do an in depth discussion of specific segments of the code. A report should be made by the moderator to summarize results.
        - Want to do these formal inspections with stable code that shouldn't really be changing that much.
    - Pair programming: One developer has the keyboard and the other watches to think strategically about where the code is going and if it is achieving the purpose. Basically have two heads instead of one to fight the bottleneck of thinking power and planning in development.
        - The code quality of the piece of code will inevitably be of the better programmer of the two.
        - The pair of members will keep each other honest and will make sure there aren't any corners being cut.
        - Also helps to educate the junior programmers (by pairing them with more senior people)
        - Don't let the observer relax, don't let the person at the keyboard take total control, avoid petty disputes, and rotate partners.

      Overall goal here should be to be able to develop higher quality code more efficiently.
- Dynamic perspective of software processes
    - Along the y axis are the different stages of the process (communication -> planning -> modeling -> construction -> deployment)

    - Along the x axis are the things you should always do at some point or at all points (metrics, estimation scheduling, risk, change, management of quality).
    - The dots on the graph refer to the particular action and the stage at which the action will happen.
- Traditional software process
    - Waterfall model: Focus of the model is on the y-axis and on order/consistency of the different stages. Helpful for when you have a new project that is similar to one in the past.
        - Process: Requirements reading -> architecture+design -> component design -> coding -> unit testing -> integration testing -> system testing -> acceptance testing
        - We want to finish each stage before we start the next one.
        - Parts of the waterfall (second half) are checks on the parts in the first half
        - Problems with this approach are that:
            - It is slow since we lose parallelism. One stage can't be started until the previous one is complete.
            - Inflexible to problems that come up later in the process.
            - Doesn’t work in environments where you discover requirements while you’re coding/testing.
    - Repeated Waterfalls: To fix the parallelism, you can do different stages in parallel for different versions of the product. Also, try to make the dependencies more fine grained. A whole stage might not need to wait for the whole previous stage to be complete.
        - Prototyping is a way to accelerate the waterfall by creating a mock solution to the whole problem. This makes the time spent of each stage decrease when you actually start the waterfall.
    - Spiral Model: Similar to repeated waterfall, but you think of each iteration of the waterfall (which could refer to as the different versions in repeated waterfall) as a different product. Each iteration represents a major change in technology, system, or user interface.
    - Rational Unified Process: Helpful for object oriented design projects. Below are the phases for RUP.
        - Inception: Vision document - what app should be doing, brief use cases, business model, risk cases
        - Elaboration: Use cases, analysis model, architecture, in depth risk analysis, plan for doing the rest of the project.
        - Construction: Design model overviewing the design classes and interfaces. Actual coding. Test cases, test plan/procedure
        - Transition: Development -> User actually using the stuff. Include feedback from beta testers
        - Production: Bug fixes

      The phases are listed in sequential order but you are allowed to jump between the stages, unlike in waterfall. RUP encourages iterative development, manage

    the changes to requirements and the effects the changes will have on other components.
  - Agile Methods: Want to promote adaptability because needs and requirements change quickly, promote self organization, collaboration/communication, and working software every 2 weeks. In order to get those pros, we have to give up some processes, tools, documentation, and planning.
    - The pros are that design and construction are interleaved, more cost effective (you spend more in the beginning with Agile, but the cost will plateau later on).
  - Extreme Programming is a type of Agile method. The phases are:
    - Planning: Set of stories that describe the product and improvements needed to the product. The customer sets the value and the dev team sets the cost (if > 3 weeks, then split the story).
    - Design: Create CRC (class, responsibility, collaborator) cards. You also create spike solutions which are design prototypes for the risky parts of the service.
    - Coding: The overall code, unit tests, refactoring to make future development easier.
    - Testing: Just acceptance tests specific by the customer. No integration or system tests really.
- Timeline for software processes: Waterfall -> Rational Unified Process -> Agile
- Scrum sprints are 2-4 weeks and they involve a product backlog.
  - First step is to assess the priorities. Then, select the features you want to focus on. Then develop. Finally, you need to review
    - All those stages should involve the customer, except for the develop part.
  - There is a lot in common with XP.
- There are a lot of Agile methods in general, and they differ in the types of communication, scheduling, etc.
- Problems with Agile methods
  - Mismatch with formal organizations like the government or military. This is because you have lots of rules developed over many years. They want specs that are formal, while Agile doesn’t focus as heavily on that part.
  - There is inertia in large organizations that are used to planning. It is hard to switch over to such a different development process.
  - Tempting to skip refactoring
  - Prioritization can be hard in large organizations. Hard to determine priorities and that can cause trouble in the Scrum meetings.
  - Developers and customers are sometimes not good at collaborating.

<u>4/10 - Week 2</u>

- In complex systems, we have technical and sociotechnical components.
  - Technical system refers to the browsers, JS, FB servers, etc.

    - Sociotechnical system is talking about the fact that FB is a social network that includes rules provided by FB and the ones society has.
        - This system is built on top of the technical system
- There are emergent system properties that come about when the size of the system grows. The kinds of things that you have to worry about change when you get to a certain scale.
    - Building a zoo for 100 monkeys is totally different than building one for 10 monkeys.
- Some common emergent system properties are security, reliability (definition of reliability changes as well), etc.
- As a system gets more complex, there are more things that you might not know about and more nondeterminism.
    - A wicked problem is an example of something you don't really know about until you've solved it. You didn't know the existence of it beforehand.
- Emacs 27 Class Example
    - Want to create a function called dumper that saves state. It wants to generate an executable, but that is getting harder with ASLR.
    - Since that won't work, we also have pdumper, which writes data structures in a binary format and it is machine dependent. However, one problem is that you could have a bad load of the format given differences in compiler flags and architecture styles.
        - I think the solution is to compile emacs twice, and after the first time, take a checksum and make sure it is the same as what's expected?
        - You can make a choice to checksum all the files (only have to link emacs once) or checksum the executable (have to link emacs twice).
- Idea of reproducible builds is to start with the same sources and finish with the exact same executables. They could be different because of timestamps included during the compilation causing a technically different .o file.
- 4 Levels of system modeling
    - World View - Enterprise strategy. Looking at the whole society and thinking about what will make the whole org succeed. Requirements engineering is involved.
    - Domain View - Not looking at entire enterprise, but rather just a business area design. (e.g Registrar instead of the whole UCLA organization). Component engineering is involved.
    - Element View - Business system design. (e.g process of enrolling rather than just the registrar). Analysis and design modeling is involved.
    - Detail View - Construction and integration (e.g one singular function in the enrollment website)
- UML is a language that helps with system modeling. It has deployment diagrams, activity diagrams for procedures that people should do, class diagrams, and use-case diagrams. Good way to visually communicate between engineers.
- Systems development process
    - First create a conceptual design that leads to your system vision or proposal.

    - Then, procurement which leads to regulations and competition.
    - Then, development with a waterfall-like model
    - Finally, operation which is where flexibility and adaptability.
- Modeling is the bridge between requirements and design.
    - Modeling needs to done in a written way and needs to be able to change. A picture taken of a whiteboard isn't really able to be changed.
    - Models needs to be decompostable in that they should be able to be broken apart.
    - Focus on what you want done, not how you're going to do it. Save that for a later stage.
    - Modeling can help you adjust both the requirements and design.
- Classic models
    - Data modeling concepts
    - Data flow modeling
    - Scenario based modeling
        - Sequence modeling that looks at the relationships between different objects or views running concurrently.
        - Control flow modeling that looks at how program moves between states.

4/19 - Week 3

Practice Debate Notes

- Topic: Developers should strongly prefer inheritance to delegation.
    - Inheritance is built into all OOP languages. https://www.geeksforgeeks.org/delegation-vs-inheritance-java/
- Topic: Pair programming more cost effective than formal inspection
    - Pair programming lets you catch mistakes earlier. It is more costly in terms of time, but reduces the amount of future development work there may be. It also encourages devs to keep the code quality high since there is somebody else checking your work. Also helpful for onboarding new employees. Junior devs can have an easier time adapting to the codebase and getting an understanding of the codebase. However, pair programming doesn't scale well, devs expend more energy in communication and understanding other people's coding style. Also holds back senior devs from working on larger and more important issues.
    - Formal inspection is a common review process. Goal is to identify defects after the dev has finished coding. Saves a lot of time when people don't have to work together at the same time. Allows devs to review code on their own time, which is more time efficient.

4/22 - Week 4

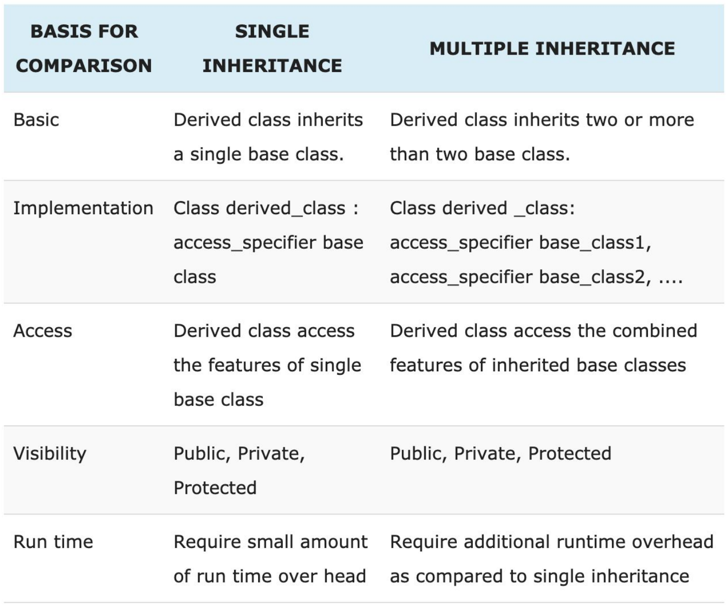
Debate Notes (In class)

- Topic: Developers should strongly prefer inheritance to delegation.

    - Pro: Inheritance lets you reuse code. Will lead to less of a need to change code in lots of different locations. More useful in real world situations where people and time are finite resources. Is-a relationship is a lot clearer than the has-a relationship. Lots of developers already have experience with inheritance patterns, meaning that it's easier for companies and organizations to train employees. Inheritance is more intuitive and we can make use of parent child relationships. Find good use cases for it though.
    - Con: Delegation (has-a) and inheritance (is-a) solve problems in different ways. Both can be useful in certain times and contexts. Code reuse is also there in delegation since instead of inheriting from a different class, we create a variable instance?
- Topic: Multiple inheritance traits are more cost effective than just single inheritance.
    - Pro: Both inheritance types encourage code reuse. Multiple allows more flexibility in terms of classes that share features. Can be used in incorrect contexts though and can be used when it's not really needed. There is also ambiguity for when the parents implement a certain function in different ways. What version should the child use?
    - Con: Simplicity, avoids ambiguity, and performance gains are the big pros to single inheritance. Java uses single inheritance and it is very popular.

Debate Notes (Review)

- Inheritance in Java programming is the process by which one class takes the property of another other class. Delegation is simply passing a duty off to someone/something else. It means that you use an object of another class as an instance variable, and forward messages to the instance.
- Delegation
    - (+) It is better than inheritance for many cases because it makes you to think about each message you forward, because the instance is of a known class, rather than a new class, and because it doesn't force you to accept all the methods of the super class: you can provide only the methods that really make sense.
    - (+) The primary advantage of delegation is run-time flexibility – the delegate can easily be changed at run-time.
    - (-) Delegation is not directly supported by most popular object-oriented languages
- In single inheritance, we do have only one base class which is inherited by only one derived class. In multiple inheritance, we have more than two base classes which are combinedly inherited by only one derived class.



BASIS FOR COMPARISON	SINGLE INHERITANCE	MULTIPLE INHERITANCE
Basic	Derived class inherits a single base class.	Derived class inherits two or more than two base class.
Implementation	Class derived_class : access_specifier base class	Class derived _class: access_specifier base_class1, access_specifier base_class2, ...
Access	Derived class access the features of single base class	Derived class access the combined features of inherited base classes
Visibility	Public, Private, Protected	Public, Private, Protected
Run time	Require small amount of run time over head	Require additional runtime overhead as compared to single inheritance

- Multiple inheritance
    - (-) Multiple inheritance is quite confusing as here a single derived class inherit two or more base class. If the base classes have an attribute or function with the same name then for derived class, it becomes difficult to identify that which base class's attribute or function it is it has to derive.
    - (-) Overhead issues are lesser in case of single inheritance. But in case of multiple inheritance the object construction or destruction invokes the constructors and destructor of the parent class in the class hierarchy which increases the overhead.
    - Single inheritance is more towards specialization. Whereas multiple inheritance is more towards generalization.
- Single inheritance
    - (+) Single inheritance is somehow more simpler and easy to implement than multiple inheritance.

Lecture Notes

- Architecture and Design are things that are kind of independent from the choice of programming language.
- Architecture deals more about the hooking together of components, while design is more about knowing your app and what components you want to create.
    - Architecture is the glue and design is the parts.
- The type of glue you use depends on your program and data organization.
- Standard architectures
    - Repository architecture: Apps sit on top of the database. They don't communicate directly to one another, but rather they do it indirectly through the DB. The DB contains the state.

  - Dataflow architecture: No central DB, and you hook apps together through pipes. Data flows through the apps and the pipes. The state isn't known to one particular individual. There is just data, and functions with inputs and outputs.
    - Ex) Shell scripts have this architecture.
  - Call and return architecture: One function calls and it returns. This allows you to hook different parts of your code together.
  - Message passing architecture: No call and return, but rather clients and servers can send messages to each other. Can implement call/return if you want, but this approach gives you more flexibility for what you can return for each message.
- Layered architectures:
  - Wedding cake diagram where you have different levels describing different parts of the system and what component depends on what.
- Other architectures:
  - Master slave architecture is where you have one master that schedules the work for the slaves/workers to do. The master shouldn't be the bottleneck and the workers shouldn't be waiting around too much.
  - Fat vs Thin client is where server has to do much of the work and the client is very thin. The server will take care of how to simply stuff, do computation, etc. Thin client advantage is that you don't store much state there and thus it is more portable and it's more secure in that outside people trying to access it won't get too much.
  - Multi-tier client server is where you have different levels of servers: cache servers for the accessing the webpage, web server for the webpage itself, and then DB server for getting the actual info.
    - Cache server is a client for the web server and web server is a client for the DB server.

    Need this layered approach since you have a lot of people trying to access the DB server.
  - Distributed components are where you rely on brokers to hook things together?
  - P2P architecture where there isn't a central DB and the peers are the ones that exchange info and data. The risks with this system are security and trust related.
  - Event processing architecture is popular with IoT and realtime services. The key is that you handle events very quickly.
  - Distributed architecture is where you get high availability, scalability, resource sharing, etc.
- Difference between SOA and RESTful. Both are concerned with accessing Web services.
  - REST is basically an architectural style of the web services that work as a channel of communication between different computers or systems on the internet.
  - SOAP is a standard communication protocol system that permits processes using different operating systems like Linux and Windows to communicate via HTTP and its XML.

    - SOAP is a more rigid set of messaging patterns than REST.
    - Instead of using XML to make a request, REST relies on a simple URL in many cases.
    - Soap +
        - Language, platform, and transport independent (REST requires use of HTTP)
        - Works well in distributed enterprise environments (REST assumes direct point-to-point communication)
    - Rest +
        - No expensive tools require to interact with the Web service
        - Efficient (SOAP uses XML for all messages, REST can use smaller message formats)

4/24 - Week 4

Debate Notes (In class)

- Topic: To improve software quality, pair programming is more cost effective than formal inspection.
    - Pro: Helps developers reduce the amount of time spent on bugs because you have two minds trying to figure it out. Finding errors at an earlier stage is a lot better than finding it later. There is also always a second opinion so there is always a tight feedback loop (where in formal inspection, the bugs are sitting there for a lot longer). It's easier to write good code in the beginning instead of writing the code, finding the bug, and then having to to restructure. It also helps more people get familiar with the codebase. Developers will be more focused since there will be someone there keeping them in check.
    - Con: Formal inspections are better at finding bugs and asks code writer to look at the bugs found. There is also a feedback loop to improve the checklist. Formal inspection also allows people to see codebase. There are a couple problems with pair programming where the programming skill of one person is vastly greater than the skill of another, and thus the more skilled person is wasting their time. Also, there could be some issues socially with pair interactions. On the other hand, just because you're friends doesn't mean that you'll work better together in coding. Also, there is a possibility there could be an echo chamber in the pair and they might be afraid to disagree. A formal inspection may be more effective.
- Topic: Every file or similar component in a software should contain a version string that is updated when the component changes.
    - Pro: Can see what files have been changed recently and thus helps out with bug finding. You can also revert back to a previous functional version of the code where the bug is not there. Helpful for new developers that aren't as familiar with Git. Also adds extra information about how old the file is, and context for why it was changed.

    - Con: There are already lots of workflows like Git that handles bug tracking and version control, so there is no need for a version string. Also, version strings may not be able to be put into files of all types (like Yaml files).

Lecture Notes

- Design Patterns: They attempt to overcome the failure of object oriented systems. The problem was that the classes would have to be slightly modified depending on where the class is used. The ideal of having plug and play modules doesn’t really work. Design patterns wanted to work at a higher level of abstraction to let the plug and play characteristic come about. You’ll will creating those slightly modified classes, but abstraction will kind of hide them.
    - Specifically, these patterns are informal way of telling you how to go about dealing with object oriented systems.
    - It can also be a relationship between context, problem, and solution.
- Factory pattern
    - In this case, you have a lot of related classes, but the objects created are not the same types as the classes. The problem is that we want to allow classes and subclasses to figure out how to instantiate themselves. The solution is to have a static method of a class and returns C but the object might be be generated by the new() command.
    - Aka trying to build objects in ways that aren’t the normal way (which is invoking new())
    - Factory method is a creational design pattern, i.e., related to object creation. In Factory pattern, we create object without exposing the creation logic to client and the client use the same common interface to create new type of object. The idea is to use a static member-function (static factory method) which creates & returns instances, hiding the details of class modules from user.
        - Client doesn't explicitly create objects but passes type to factory method "Create()"
    - Creating an object directly within the class that requires (uses) the object is inflexible because it commits the class to a particular object and makes it impossible to change the instantiation independently from (without having to change) the class. That’s why factory methods are better.
- Categories of patterns
    - Creational: Factory, Builder, Prototypes
    - Structural: Emphasizes integration, Pipes
    - Behavioral: Emphasizes communication and responsibility, Visitor
- Something about aspects and cross cutting concerns?
- Separation of concerns is where you have everybody work on a different class and have them just concentrated on that. Splitting the work involves looking at the main design from the front. However, there is also a problem when you start to look from the side view to see if there are any overall problems like too many people using too much memory allocation.

- Liskou Substitution Principle says that members of a subclass should be usable anywhere a member of the superclass could be used.
    - Subclass contains all the operations a superclass has + some of its own.
- The common reuse principle is that you group together classes only if they are used together.
- System building
    - You normally have different platforms (dev, build, target) and you'll have makefiles that are self building code.
- There are also forms of automation in shell scripts, make, automake, autoconf, configure, etc.
    - GNU Automake is a programming tool to automate parts of the compilation process. GNU Autoconf is a tool for producing configure scripts for building, installing and packaging software on computer systems where a Bourne shell is available.
    - Configure generates a unique Makefile based on the specifics of your system. Basically it is software configuration for the build process.
        - The configure script is responsible for getting ready to build the software on your specific system. It makes sure all of the dependencies for the rest of the build and install process are available, and finds out whatever it needs to know to use those dependencies.
    - Once configure has done its job, we can invoke make to build the software. This runs a series of tasks defined in a Makefile to build the finished program from its source code.
    - The make install command will copy the built program, and its libraries and documentation, to the correct locations.
- Something about Meyer's Dilemma
    - Tradeoff between the time you release and quality of the product.

4/26 - Week 4

- Roles for the project
    - Team leader (team contact)
    - Scribe to take notes of meetings and design
    - Programmers for
- Someone needs to email Davis
- Make sure TA has access to the Github repo
- Might need to do an SRS for the project
- Midterm
    - Look at textbook notes
    - Study the debate pros/cons

4/29 - Week 5

Debate Notes (In class)

- Topic: To improve program structure and quality, test driven development is more cost efficient than code driven development.
    - Pro: Writing test cases helps you stay on track and write code to satisfy requirements. You can do incremental unit testing where you can figure out if there are bugs associated with certain subroutines. However, test driven is not feasible because some tests have very long build times which means that your debug cycles are slower. You can also make the case that it encourages devs to write hacky code that simply passes test cases and doesn't get to the root of the issue. However, you can say that TDD helps you figure out what the important important test cases there are.
    - Con: Code driven development allows you to write code and test when you need to rather than all the time. However, you can make the case that you still have to deal with long build times depending on how many tests you create and how frequently you run them. Also, as the clients' requirements change, tests will have to change as well, and so it's easier to do that in CDD. A lot of companies use it as well.
- Topic: Software development and software testing should be separate career paths.
    - Pro: Dev and testing require different skills. Dev requires more broad knowledge while testing requires more specific knowledge about certain parts of the system. Companies can save money by hiring testers and allowing more experienced devs to work on more important parts of the project. Separating paths means that there is a clearer division of responsibility for devs and for testers.
    - Con: Devs know their code the best so they will do well if they are involved in that testing process. If they know that they have to do the tests, then they will get a new perspective and have better code quality. If testing is eliminated from a particular role, devs are more responsible and able to check themselves when they have to think as a tester.

Debate Notes (Review)

- Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only. This is opposed to software development that allows software to be added that is not proven to meet requirements.

Lecture Notes

- There are different ways to go about defect detection. Finding bugs is hard, but fixing them is easy.
- Different forms of testing will have different detection rates.
    - Testing, beta testing, desk checking, modeling/prototyping, pair programming, informal code review, formal inspection, informal design review, formal design inspection

- It’s not as simple as just choosing the form of testing that has the highest detection rates. You have to take into account costs and the fact that certain forms of testing are better or worse for some situations.
    - There is no best method because you have pros/cons depending on what stage you’re at and what resources you have.
- Defect amplification model
    - Let’s say you have the different stages of your software dev project (reqts, modeling, design, code). If you have errors entering each phase, then each phase acts as an amplifier where the errors will get magnified. The goal in the design is to make the amplifiers into attenuators.
- (External) Software quality aspects are:
    - Functionality - Does the program have the feature that you want
    - Correctness - Does the program match the specifications that describe what is expected.
    - Reliability - Does the program have a long mean time to failure. Basically how long it takes for the program to crash. Highly reliable pieces of software will have larger MTTFs.
    - Robustness - Does the program react well to curveballs.
    - Integrity - Does the program resist malicious intent from outside people and does it resist screwups on the devs end.
    - Efficiency - Does the program minimize resource usage.
    - Usability - Is the program easy to use and work with.
    - Adaptability - Is the program useful in a lot of different domains and scenarios without having to refactor or reconfigure it.
- (Internal) Software quality aspects are:
    - Flexibility - Easy to modify the software (for devs) for other users or for other features.
    - Maintainability - Easy to fix the software
    - Reusability - Easy to use the software in other places. Easy to cut/paste from.
    - Testability - Easy to test the correctness of the software. How hard is it to write the test cases?
    - Portability - Easy to move software to other platforms and systems.
    - Readability - Easy to put code on the screen and understand it (more low level)
    - Understandability - Easy to navigate the codebase and answer questions (more high level)
- If it is correct, then it should be reliable and it should be robust if the specs are written correctly.
- ISO 9000 is defined as a set of international standards on quality management and quality assurance developed to help companies effectively document the quality system elements needed to maintain an efficient quality system.
    - ISO 9000 says that it’s important to develop a quality management system which is a policy that emphasizes its importance. This policy should be documented and there should be a QA plan.

- Testing
    - It’s sort of unnatural for devs since you’re trying to find bugs instead of writing code to solve problems. Succeeding as a tester means that you’re trying to find bugs.
    - The job is also never really done. You can't really prove the absence of errors, only the presence of them.
    - Testing doesn’t really improve software. Customers won't really recognize it.
- Verification and validation
    - Verification asks whether we are building the product right. It is the bridge between the code and the specs.
    - Validation looks at if we are building the right product. It is the bridge between the user requirements and the specs.
- System testing is different from unit testing and integration. It’s more focused on incorporating the non-software components to see if the whole system works. Unit testing just focuses on certain parts of the software.
    - Recovery testing, security testing, performance testing, and deployment testing are subtests you would do as part of this overall system test.

5/1 - Week 5

Lecture Notes

- Software configuration management involves changing the code as well as changing the way you run the code (configuration).
- Questions to ask when doing software configuration management
    - What are the software configuration items? What stuff needs to be managed and what is managed automatically.
    - How do we record different versions?
        - Normally we use Git
    - How do we control the changes and how is responsible for making and approving the changes?
        - There is change control software like git hooks that basically run whenever code gets committed. It makes sure code will be able to be compiled and stuff.
        - Also there is manual control in that we only let certain devs make commits to master.
    - How do we ensure that the changes are done correctly?
    - How do we notify others about the changes?
        - Need some way of reporting. This could be emails or news file for users or change log file for devs.
    - How do you generate a system from the source code? This refers to the building process.
        - Files like Makefiles help automate this process.
    - How do you get software into people’s hands?

        - Release management is important to think about.
- Baseline can be defined as a place of known departure. In software terms, you have a collection of these software configuration items,
- There are version control design decisions to make.
    - Should it be decentralized where there is no central repo. Every developer has their own history and this can cause conflicts. But if it is centralized, then access to the database can be a bottleneck.
    - Should there be optimistic version control which is where you allow people to work on the same file at the same time. This can lead to merge conflicts. But if you have pessimistic VC, then people need to notify others when you start working on a module.
- Release management
    - You can try to do upgrades over a network, but you can't assume that everyone has the latest versions and thus you have to be able to support earlier code.
    - You also need to keep track of dependencies.
    - Need to think about what can go wrong with an update. There could be a partial install.
- Debugging (error = mistake by devs, fault = latent problem in SW, failure = actual occurrence of the problem)
    - Assume you have some sort of a bug report. We have to go from that to defect removal. This involves
        - Stabilizing the error: Figure out when the error happens and make it reproducible - disable ASLR, go single threaded, simplify the test case.
        - Locate the fault: Form a hypothesis about the fault and then test it and then determine the fix.
    - Try to understand the problem first and then install a fix.
- 33% speedup and then 50% speedup means that you get the program twice as fast because 4/3 * 3/2 = 2.
- Tuning vs refactoring
    - Tuning is about optimizing performance.
    - Refactoring is about making code cleaner.
    - In both, the functionality shouldn't change. Tuning will change the performance and refactoring will change the organization.

5/3 - Week 5

- Why don't software failure rates follow a bathtub curve?
    - For normal non-software things, failure rates are high at the beginning and at the end. Software doesn't wear out which means that you won't have a high failure rate at the end. However, you'll still have the high rate at the beginning. As you make updates, the error rate jumps up but then you make patches and it goes back down but you can't get to the same low level as before. The average error rate progressively increases.

- Cannot use hashing for fault isolation since it doesn't use indices to search.
- Difference between simulation and a prototype?
    - Simulation is realistic testing of components and their interactions to find bugs.
    - Prototyping is more about creating an initial version of a product. You're building enough of the system to show a scenario/problem.
- Sources of inefficiency in modern software projects?
    - Dependency management - Don't have people waiting on each other and blocked on work.
    - Poor documentation leading to wasted time
    - Vague requirements - Don't know what you're doing and thus could spend time on something that isn't actually important.
    - Poorly moderated and unorganized meetings
    - In terms of inefficiency of code while it is running, you have viruses, network issues, and centralized databases.
- Coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules.
    - Usually less coupling is better since we don't want our application to rely on outside things too much.
- 80% of the bugs are fixed by changing 20% of the reported bugs.
    - A good portion of the bugs will be easy to find and you can get a lot of them out of the way in a short amount of time, while there will be a small percentage of bugs that will be incredibly hard to find.
- IDE -> no naming conventions needed?
    - For: Faster to learn to write code, removes mistake of misnaming a variable
    - Against: Not everyone uses an IDE, easier to read code with proper naming, IDEs vary from each other, naming conventions help with code style.

5/8 - Week 6

Debate Notes (In class)

- Topic: Project team should have a single technical leader who manages the project.
    - Pro: Process becomes efficient since one leader can take control of the room, bring back focus of a discussion, and arguments will tend to occur more frequently. Would help reduce the amount of work and refactoring needed. Can be difficult for team members to take initiative since everyone assumes that other people will take charge, so if there is a leader at the beginning, they are aware and the dev process will proceed more quickly. Since all students have about the same amount of knowledge, students can push back on certain directions from the leader.
    - Con: Not all projects will benefit from one technical leader. Formal structure inhibits info exchange. One person might not have all the skills and knowledge necessary to be in charge of understanding all the technical components in a

        project. It is an unfair amount of work and responsibility put on a single person especially if it's a student.
- Topic: Agile dev is typically more cost effective than plan driven management in enterprise applications. Plan-driven software development is a more formal specific approach to creating an application. Plan-driven methodologies all incorporate: repeatability and predictability, a defined incremental process, extensive documentation, up-front system architecture, detailed plans, process monitoring, controlling and education, risk management, verification and validation.
    - Pro: Agile allows for adaptable requirements. Cost of making changes is not very high. More effective communication. Good at unknown unknown risks since you get feedback and make tests along the way. Can adapt the budget as you go. Since it is impossible to identify all the risks, important to have a software process that deals well with it.
    - Con: Does especially well with known known and known unknown risks which allows devs to plan for them early on. Better with focusing on reliability in the applications. Might not be feasible to have a variable budget.

Lecture Notes

- In project management, you have to think about the connections/communication between different people. There are N choose 2 interactions where N is the number of people in the group.
- As the size of the project grows (lines of code), then defects/kLOC also increases. You'll have almost exponentially more bugs as your code grows. Also, your productivity (number of lines written) will decrease since there's less large work than needs to be done, and more smaller features used instead.
- Thus, we need to have project management techniques that work as the project grows in the number of people and the size of the codebase.
- Danger signs of project management
    - Managers and devs avoid the best practices.
    - Sponsorship was lost or was never there.
    - Project lacks people with the right skills
    - Users don't want the product.
    - Business needs change or are unknown.
    - Unrealistic deadlines.
    - Chosen technology changes.
    - Changes are managed poorly.
    - Product scope (boundaries) isn't defined since there is always more work to do
    - Developers don't understand the users' needs. Happens when you don't have communication between the people who care about the system and the dev.
- Components of a project management plan
    - Team organization so that people have roles and duties.
        - Have to manage the people, product, process, and the project
    - Risk analysis. Have to know where the obstacles are in your plan.

    - Resource requirements in order to understand the budget. People, hardware, and software cost money.
    - Work breakdown so that people know what tasks have to be completed, the deadlines, and the milestones.
    - Schedule to organize the above ^
    - Reporting mechanism for when things go wrong, people don't show up, etc. There has to be a way of keeping track of how things are going.
- Match roles to personalities and skill sets.
- Project Management steps
    - Understand the problem by getting the right team and setting objectives.
    - Minimize turnover
    - Measure progress
    - Keep it simple stupid. Avoid risk.
    - Review what went wrong and what went right.
- PM techniques
    - Waterfall: This technique is also considered traditional, but it takes the simple classic approach to the new level. As its name suggests, the technique is based on the sequential performance of tasks. It is traditionally used for complex projects where detailed phasing is required and successful delivery depends on rigid work structuring.
    - Agile: Agile project management method is a set of principles based on the value-centered approach. It prescribes dividing project work into short sprints, using adaptive planning and continual improvement, and fostering teams' self-organization and collaboration targeted to producing maximum value. Agile is used in software development projects that involve frequent iterations and are performed by small and highly collaborative teams.
    - RUP: Rational Unified Process (RUP) is a framework designed for software development teams and projects. It prescribes implementing an iterative development process, where feedback from product users is taken into account for planning future development phases. RUP technique is applied in software development projects, where end user satisfaction is the key requirement.
    - PERT: Program Evaluation and Review Technique (PERT) is one of widely used approaches in various areas. It involves complex and detailed planning, and visual tracking of work results on PERT charts. Its core part is the analysis of tasks performed within the project. This technique suits best for large and long-term projects with non-routine tasks and challenging requirements.
    - XP: Extreme project management technique (XPM) emphasizes elasticity in planning, open approach, and reduction of formalism and deterministic management. Deriving from extreme programming methods, it is focused on human factor in project management rather than on formal methods and rigid phases. XPM is used for large, complex and uncertain projects where managing uncertain and unpredictable factors is required.

    - Critical Path: Actually, this technique is an algorithm for scheduling and planning project works that is often used in conjunction with the PERT method discussed above. This technique involves detecting the longest path (sequence of tasks) from the beginning to the end of a project, and defining the critical tasks. Critical are tasks that influence the deadlines of the entire project, and require closer attention and thorough control. Critical Path technique is used for complex projects where delivery terms and deadlines are critical, in such areas as construction, defense, software development, and others
    - Critical Chain: Critical Chain is a more innovative technique that derives from PERT and Critical Path methods. It is less focused on rigid task order and scheduling, and prescribes more flexibility in resource allocation and more attention to how time is used. This technique emphasizes prioritization, dependencies analysis, and optimization of time expenses. It is used in complex projects. As it is focused on time optimization and wise resource allocation, it suits best for projects where resources are limited.

<u>5/13 - Week 7</u>

Debate Notes (In class)

- Topic: Experienced cost modeling is better choice than algorithmic cost modeling for factory floor software applications.
    - Pro: There is flexibility in taking advantage of past human experience. Need leaders with deep experience and can help with calls that formulas can't model. Algorithms require analysis of many projects and this does not work in cases where there are unique projects. We won’t need as much data.
    - Con: Models are holistic and are less error prone. Can get accurate cost estimates given accurate attributes. Allows you to do the cost estimations at a higher frequency since it is a fixed machine cost rather than valuable human hours. Algorithms also can provide high levels of precision and will stick around even when employees leave. Scales with the amount of code, so might take a lot longer for bigger projects.
- Topic: COCOMO II is a good choice for modeling cost of software development for the telescope project
    - Pro: More flexible, well known, and well documented.
    - Con: Empirical estimation models are reliant on past projects and might not be good indicators for specific future projects.

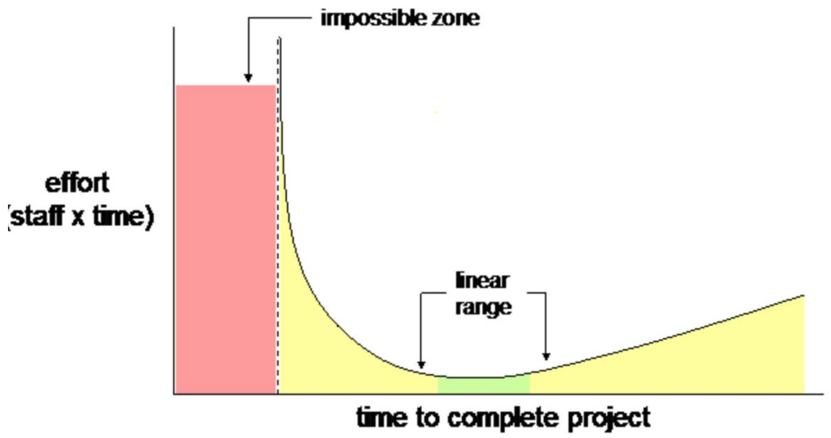
Debate Notes (General Notes)

- The accurate prediction of software development costs is a critical issue to make the good management decisions and accurately determining how much effort and time a project required for both project managers as well as system analysts and developers. Cost estimation models are mathematical algorithms or parametric equations used to estimate the costs of a product or project.

Lecture Notes

- Estimation is the part of planning that is tough. You want to be able to predict the cost and the variance of the cost.
    - You ideally want some probability distribution of the cost but that’s pretty hard to obtain.
- Risk estimation is one subtype. This can be done early in the project. Start by writing down what can go wrong.
    - Examples of things that can go wrong: Lead dev leaves, hardware isn't good enough, etc. Each of them has a probability of happening. They also have an impact measure of the cost if that particular thing happens.
        - Multiply probability and cost to figure out expected loss, which will help you prioritize which risk to worry about the most.
- 3 times to do risk assessments
    - Preliminary - During requirements
    - Life cycle - During system development. At this point, you’ll know the implementation, but you won't really know the edge cases or the high use cases you only see during production.
    - Operational - During production. At this stage, you want to respond to known issues.
- Types of risk
    - Business risks: Business risks revolve around the possible impact to the organisation as a whole if individual projects suffer some form of disaster.
        - Losing a sponsorship
        - Loss of confidence in the company or corporate embarrassment resulting from a failed product or service may contribute to a falling market share and a smaller customer base which in turn leads to a reduced share price and loss of turnover or profit.
    - Product risks: Product risk is the risk associated with the software or system, the possibility that software or system may fail to satisfy end user/customers expectations is known as product risk. It is the set of things that could go wrong with the service, software or whatever is being produced by the project.
        - Unsatisfactory software has risk and has the possibility of failure which can cause major functional damage.
        - Low quality software can have many problems like functionality, reliability, usability or performance.
        - Product risks can be classified in many ways but they fall into one of two main categories, functional and nonfunctional. Functional risks relate to how the product may not achieve the activities it is designed to do, such as receiving data, performing calculations, producing reports and interfacing with other systems. Non-functional product risks relate to such possible problems as not performing a (correct) calculation quickly enough or being unstable with a high number of concurrent users.

    - Project risks: As we know that testing is an activity and so it is subject to risk which may endanger the project, so we can say that the risks associated with the testing activity which can endanger the test project cycle is known as project risk.
        - Examples: Delay in the test build to test team.
        - Unavailability of test environment.
        - Delay in fixing test environment due to lack of system admin.
        - Delay in fixing defects by development team.
        - Going over-budget, missing key milestones or deadlines, issues of resource availability, scope-creep, lack of support from higher management
    - General similarities and differences
        - In the same way that project and business risks are quantified (using likelihood and impact) product risks should also be categorised and measured. The main difference with a product risk is that it is normally, but not always, mitigated by appropriate testing (in contrast, a project risk cannot be mitigated by testing).
- Risks can be categorized across the boundaries.
- Planning has to be done to manage these risks.
- Agile planning involves iterations where you choose the things you want to implement, break them down into tasks, and devs will sign up to do particular tasks.
    - You also want to review how people are doing frequently.
    - If you're late on getting the tasks done, try to reduce the scope.
    - What the tasks are will depend a lot on the project and product type. The tasks can also be divided into subtasks.
    - The tasks also likely will have dependencies and thus you need to figure out which tasks need to be scheduled before others.
        - The width of your task schedule should match the team size. Aka at any point in time, everyone should be working on something.
        - However, this isn't possible because dev competencies and skills place constraints on this.
        - So this means that the schedule is not going to be ideal. It is a way to track progress tough and you can revise the schedule as you go.
- Agile wants to avoid large planning overhead.
- You also have situations where you have to decide whether to build/buy, in house/outsource, etc. Since there is a lot of combinations of options, you need to prune the options that aren't good fits.
- PNR curve shows the relationship between dev time and the total project effort. For a minimum amount of effort, you have a dev time of t_0. However, most times we don't want to minimize effort, but we're good with increasing the effort if that decreases the dev time.



5/15 - Week 7

Debate Notes (In class)

- Topic: N-version programming (running n versions of the program and voting) more effective that spending the same amount improving the reliability of a 1 version program.
    - Pro: Helpful for when cost of failure is high. It will help reliability and lower the probability of fatalities. Reducing failure rate of the software will reduce the number of deaths. There will also be lower integration costs as well as higher reliability. Lots of efficiency with the small sizes of teams working on different versions of the program. Single point of failure in the 1-version program as well.
    - Con: Extra time to produce N possibly flawed versions could be spent on creating 1 robust and fully tolerant version. Assuming that the N versions is independent is wrong. The failures aren't independent and thus we can't conclude the reliability is higher (Or if you do want to ensure independence, then it will be more costly). Coming up with novel solutions means that people will follow the simple approach and thus the N programs can be relatively similar. There is also an inefficient development process for the N programs. There is also a single point of failure with the comparator that looks at the results of the N programs. There is also not a clear choice for what the best value for N will be, and there is only one chance to choose N. Splitting a group into N subgroups will mean that you could have some groups that don't have the right people with the right skill set.

Lecture Notes

- Dependability focuses on making sure that software works even when real word accidents happen.
    - This mostly handles the random malfunctions and bugs in the program.
- Security focuses on making sure that software works even when an adversary is attacking you.
    - This means that the attackers will know where the bugs are and they are trying to exploit them.

    - This tends to be a harder problem since the problems don't come up randomly but rather another party is actively trying to figure out how to being your system down.
- The above two aspects are more important than functionality since you have to take into account the cost of failure in projects. The cost of the failures (reputation, unintended harmful effects, etc) can exceed the cost of the software itself.
- The two aspects are hard to modularize. They need to apply to the whole system, not necessarily to different components. Security aspects will end up leaking throughout the module. Hard to isolate these two things into different modules. You need to consider the effect of other modules as well as hardware and operational and network failures.
- Often times, increase in dependability requires an increase in dev costs, and there are diminishing returns. This applies to a lot of security and dependability concerns. This is because it becomes harder and harder to make your system more secure and dependable as you go.
- Dependability is composed of (first 5 are the ones that Sommerville talks about)
    - Security
    - Safety - How likely is the SW failure going to damage the users
    - Reliability - How likely is it that the SW will match expectations. Will the SW match the specs?
    - Availability - How likely is it that the SW is up and running? Will it have to go down from time to time?
    - Resilience - How well does the system resist and recover from failures. This refers to failures from the infrastructure.
    - User error tolerance - How well does the SW tolerate errors from the users? Is there an undo button?
    - Repairability - Are the bugs easy to fix?
    - Maintainability - Is it easy to change the SW to adapt it to different circumstances?
- These aspects are not independent of each other.
- There are different types of problems and bugs that a program can have.
    - Human error - Mistake by the designer
    - Human fault - Latent problem in the system
    - System error - Bad internal state
    - System failure - Observable bad behavior.
- Terminology around safety
    - Accident - Unplanned event causing injury. This is similar to a system failure.
        - Risk probability is the chance that the accident occurs.
    - Hazard - Condition in the system that might lead to an accident. This is very similar to a system error and/or a human fault.
    - Damage - Cost measure.
    - Hazard severity - Looks at the what the worst case damage will be for a given hazard.
        - Hazard probability is the chance that the hazard occurs.

- Terminology around security
    - Exploit - Unplanned event causing the loss of secret information.
    - Vulnerability - Condition that might lead to an exploit.
    - Exposure - Time when the system is available to be attacked.
    - Attack - A successful attempt to exploit.
    - Threat - Vulnerability that might be attacked in a particular situation.
    - Control - Mechanism to make a vulnerability harmless.
- Accidents and exploits are dynamic while hazards and vulnerabilities are static.
- Dependability requirements are hard to come up with. It's easy to think that you're done when you're not. Also easy to put in too much and over specify what you want.
    - Need to identify the risks, prioritize the risks, and decompose them into root causes.
- Reducing the risk of faults involves:
    - Testing programs
    - Avoiding subscript errors
    - Tolerate them, toss exceptions, and figure out contingency plans.

<u>5/20 - Week 8</u>

Lecture Notes

- In order to make sure that software quality is high, we want to reward high quality code, blame people for bugs, and make it the QA team's responsibility.
- MTBF is the mean time between failure and looks at the time in between start of consecutive down times. It is a measure of how reliable a hardware product or component is. The MTBF figure can be developed as the result of intensive testing, based on actual product experience, or predicted by analyzing known factors.
    - You also have MTTR and MTTF.
- POFOD = Probability of failure on demand is about MTTR/MTBF
- ROCOF = Rate of occurrence of failures = 1 / MTTF
- AVAIL = Availability. Want the number of 9's to increase to at least 4.
- Software safety revolves around modeling the risk in the production runs, focus on the errors that users care about.
    - This analysis can come through a fault tree analysis which is a tree where the immediate children is a bunch of situations connected by ORs. These describe all the situations that can cause the error at the top. We also want to model the probabilities of the leaf events.
- 3 levels of acceptability with hazards - acceptable, unacceptable, and as low as reasonably possible.
- Dependable programming means that you should do information hiding, exception handling, and input validity checks. Also helpful to have restart capabilities, checkpoints where you save the entire state of the program, and timeouts. Can also help to avoid floating point, new keyword, and pointers/subscripts.

- O.S architectures should have a kernel protection system (making sure that the programs that can access the kernel aren't doing something malicious) and self monitoring architectures to identify processes that have gone wrong.
- Creating dependable SW processes has the goal of convincing the customer that your SW is reliable. This includes test plan creation/management, static analysis of code/design, formal inspections, etc.
- Security risk assessment involves identifying
  - The assets, the value of the assets, and exposures. (stuff on your side)
  - The possible threats and attacks (attackers' stuff)
- It also involves writing security requirements.
- Security design guidelines
  - Use an explicit security policy.
  - Defense in depth
  - Fail securely. Have the failed state be more secure than the normal state.
  - Balance security and usability for the users.
  - Log user actions
  - Specify the input formats
  - Design the deployment procedure.
- The goal is to have system survivability where we repel attacks, detect successful attacks, and recover while the system is running.
- AWS advantages of using formal methods
  - Eliminate hand waving.
  - Automatic checking within design
  - Documentation
- Need a champion of the approach

5/22 - Week 8

Lecture Notes

- State transitions are described via logical predicates. There are preconditions and postconditions.
- There are also safety properties in the form of invariants.
  - Want to make sure that every state transition keeps the invariants.
- Downsides of formal methods
  - Hard and expensive. Need to have an automated theorem prover since the state transitions and preconditions and postconditions are hard to keep track of.
  - Notation is crucial. Might need to come up with a new notation that fits with your application.
  - Easy to overformulize.
  - Not a substitute for knowing what you're doing.
  - They still need comments.
  - No guarantee freedom from bugs.

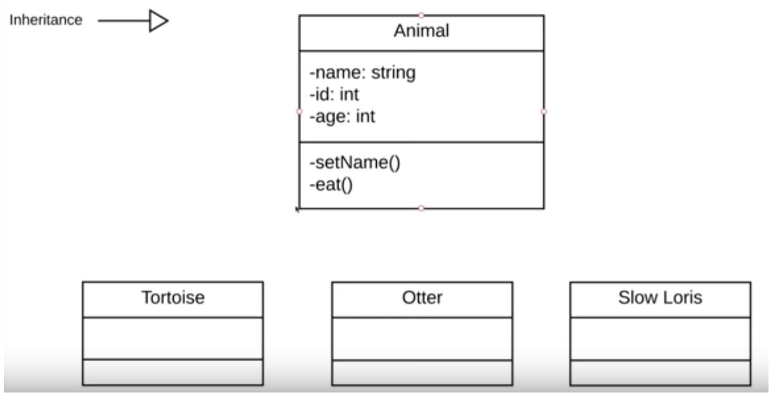
    - They need engineering like everything else. Need a design phase, documentation phase, need to do version control on your theorems, etc.
- Proof carrying code means that you have source code and along with it, you have a proof of a certain characteristic.
- Tools that help software devs
    - Dev environments like Eclipse and Emacs. Within this, you have refactoring tools, diff/patch/merge, warnings about unused code, automatic imports, styling, GDB or debuggers more generally, package managers
    - You can also have project specific tools
- UI/UX and documentation are linked because if one is strong, then it can help the other out.
    - These can referred to as marketing or education. They are both aimed at getting your good software idea into people's hands.
- UI/UX design
    - Reduce the user memory load
        - Have a consistent UI in terms of how a user does a particular operation that is replicated across the site. Make the operations consistent across the different components.
        - Be compatible with the existing practice.
        - Use real world metaphors
        - Disclose information progressively.
        - Have reasonable defaults because users don't like to click around a whole lot. If they don't like the defaults, then it should be easy to do shortcuts.
- UI Design Patterns
    - Breadcrumbs - Path to go back
    - Progress indicators
- Accessibility
    - Provide text alternatives for any significant non text content.
    - Synchronize media alternative with screen readers and closed captions.
    - With any alternatives that you come up with, don't lose any info from the original.
    - Make all the functionality useful from a keyboard.
    - Don't create content that is known to cause seizures.
- UI Design Principles for web apps
    - Mask latency issues of sending messages to server. Make the UI feel like it's doing something and/or try to anticipate the next request (preloading).
    - Be sure to track the status of the user and save work.
    - Same UI shared among many apps
- UI Design Principles for mobile apps
    - Typing is more of a pain
    - Direct manipulation
    - Metaphors matter more
    - Immediate feedback and aesthetics are more important.

- Usability can be defined as an application being more effective, more efficient, easier to learn, fewer/less important errors, and/or more satisfying.

5/29 - Week 9

Lecture Notes

- Software evolution can occur because of a change in the business needs, teams, or technologies used.
- Agile cycle: Change requests, change analysis, release planning, code changes, and final code release.
- The changes required will often come with maintenance of functionality, bug fixing, and porting. ⅓ of the changes will come in the development of new features.
- Software reengineering involves turning a legacy program into a new program. This involves the following steps
    - Replace individual components separately.
    - Rewrite from scratch.
    - Replace individual functions.
- One problem with reengineering is understanding legacy code. That code can also be obfuscated which can make things even harder.
- Reuse design: Standardize the interface protocols, data, and the architecture.
- UML



    - s

5/31 - Week 9

Discussion Notes

- PM who wants to expend 300K on a project whose minimum is only 200K
    - In this curve, the cost is parabolic to the development time.
    - You can move the curve to the left and try to get the project done faster if you have that 100K to spend.
- Example of a software error that doesn't lead to a fault, and a fault that doesn't lead to a failure.
    - A fault is a failure that hasn't necessarily happened yet, while a failure is something that actually did go wrong. The fault is the discrepancy in code that

has the potential to cause the fault. An error is a mistake from the human in creating the fault.
        - So error -> fault -> failure
    - Error that doesn't lead to a fault: This is where the human messes up but the program doesn't. An example would be accidently adding instead of subtracting, but the second number is always zero so it doesn't really matter.
    - Fault that doesn't lead to a failure: Example is where you have a for loop where there is a line of code that accesses an element outside the error, but based on the loop structure, the program will never run that code. The potential for the failure is there, but it doesn't happen. Another example is an overflow possibility existing (aka you never check for arithmetic overflow) but it never actually happens.
- How to know whether you've devoted enough dev resources to software testing?
    - Cost benefit analysis where you see if you get more benefits than the costs after doing the software testing.
- Are verification and validation wicked problems (don't know the solution/problem until you've actually solved it aka you don't know what you don't know)?
    - Yes because it's hard to verify and validate something because you're not guaranteed to know if you've found all the errors. Can't verify and validate the problems until you see them. You can't predict all the errors just from thinking about the code.
    - No because the wicked problem is actually designing of the application. Verification and validation aren't wicked if the design is good.
- Relation between Software Maturity Index and Mean time between failure?
    - Should increase the MTBF if SMI increases.
- Loop testing tests the loop entrance and running of the code. Test the first and last values rather than the ones in the middle.
    - Often finds the errors faster because you're just looking at the edge cases (beginning and ending conditions) instead of running through all the test cases.
- OO principles that can also apply in other contexts
    - Open-closed principle says that a class is open for subclassing but closed for modification. You don't want one class to be modified, so you modify the subclass that inherits from it. This is mainly used in object oriented programming.
    - Liskov Substitution Principle says that subclasses should be suitable for their base classes, and this is also only used in OO programming.
    - Dependency Inversion Principle says that you want to have dependencies on abstractions not concrete classes. This can be used outside OO programming in the context of pointers that you can make void.
    - Interface Segregation Principle can be used outside in context of multiple sub functions.
    - Number of classes only useful in OO
    - Coupling between object classes can be used outside in context of linking one .o file to another.

- Earned value analysis is looking at how valuable software was at a particular time and how much products were worth at past points.

6/3 - Week 10

Lecture Notes

- Realtime characteristic refers to predictable time performance.
  - We care about fast operations but we want them to be predictable. We don't want situations where something takes one second one time and then 20 seconds another time.
- The user is in charge, not the program.
- Embedded systems characteristics
  - In embedded systems, there are a wide variety of devices, real time response is key.
    - In normal applications, correctness and performance are independent of one another, but with embedded systems, they influence each other since it's not always possible to have both.
  - In addition to latency limits, there are also size, power, and energy limits.
  - Programs in ES also never stop.
  - May not need portability because you might not need them to run on a wide variety of devices, since there is normally a single type of device you are running the program on.
- Hard real time is where deadlines in terms of latency must be met. Often, you see this in safety critical applications like car/train braking systems. Soft real time are applications where missing a deadline has a cost, but it's not the end of the world.
- In real time modeling, you have a state machine and you get inputs from the outside world (understood through sensors) and input from a clock, and then the outputs to actuators that perform some action.
  - A good design approach is to have a stimulus response event loop. This will response to periodic events (from the clock) and aperiodic ones (from async devices). These are combined in an event queue which can suffer from the convoy effect that can cause starvation of certain jobs. Solution is to break up the problem into multiple logical threads.
- Patterns for embedded systems
  - Observe and React: Wait for an input from sensors, respond quickly, and just wait otherwise. This approach is good when events are rare.
  - Environmental Control: Monitoring the outside environment at all times. This approach is good when events are common or in a continuous stream.
  - Process Pipeline: This approach is good when there could be a queue overflow. A router would have this. In this, you can hopefully break the event handlers into different pieces.
- A real time operating system needs to have

    - A scheduler that takes in a list of tasks and interrupts, and there will be a schedule as an output. The scheduler has to decide what to do next given a bunch of different inputs and factors.
    - Resource management is the component that takes the list of things that need to be run as well as info about the number of resources available and if there are prerequisites.
    - Dispatcher will be the one that gets info from ^ and actually runs the processes.
- Doing timing analysis involves looking at specific deadlines that should be met from the hardware and the user POV, as well as looking at the execution time of the code (worst case - hard real time and best case - soft real time).
- IP Law
    - Copyright refers to the expression, not the idea. It is long term: life + 75 years
    - Patent is shorter term, about 20 years. Hopefully it encourages people to publish their ideas.
    - Trademark used to make sure people can tell things apart from imposters.
    - Trade secrets refer to the info protected by NDAs.
    - Personal data refers to the rights to the data collected by companies about you.

<u>6/5 - Week 10</u>

Lecture Notes

- A software license is not an express contract, and it only grants someone permission to do something. Even though it's not a contract, there are some strings attached on you following certain terms.
    - Often part of a contract but doesn't have to be.

McConnell Notes

<u>Chapter 1 (Construction)</u>

- Construction is mostly coding and debugging but also involves elements of detailed design, unit testing, integration, integration testing.
    - Important non construction activities include management, requirements development, software architecture, user-interface design, system testing, and maintenance.
- Construction is the central activity in software development. It is after the requirements and architecture specification, and before the system testing.
- The product of construction is the source code and that is the most accurate description of the software.

Chapter 2 (Metaphors)

- By comparing a topic you understand poorly to something similar you understand better, you can come up with insights that result in a better understanding of the less-familiar topic. This use of metaphor is called “modeling.”
    - The power of models is that they’re vivid and can be grasped as conceptual wholes.
    - Metaphors help you understand the software-development process by relating it to other activities you already know about.
- An algorithm is a set of well-defined instructions for carrying out a particular task. A heuristic is a technique that helps you look for an answer. It tells you how to look, not what to find.
- The writing and software analogy doesn’t work because software is normally written with a bunch of people, can't be done in one sitting, undergoes a lot of changes over time, and has a lot of reuse.
- Software and farming analogy isn't great because you don’t have any direct control over how the software develops (like you don’t have control over whether the seeds grow, you can only plant them).
- Best analogy is construction. Treating software construction as similar to building construction suggests that careful preparation is needed and illuminates the difference between large and small projects.

Chapter 3 - 3.5 (Overall SWE Process)

- “Measure twice, cut once”. Basically spend a lot of time in project requirements and planning.
- When talking about the programming process and the requirements for a particular project, need to appeal to logic, analogy, data.
- The cost to fix a defect rises dramatically as the time from when it’s introduced to when it’s detected increases.
- Overall Process: Problem Definition -> Requirements -> Architecture -> Construction -> System Testing -> Future Improvements.
    - A problem definition defines what the problem is without any reference to possible solutions.
    - Requirements describe in detail what a software system is supposed to do, and they are the first step toward a solution.
    - The goal of preparing for construction is risk reduction.

Chapter 4 (Construction Decisions)

- Key construction decisions
    - Choice of language: Want to use something familiar to programmers, think about compiled vs interpreted. Every programming language has strengths and weaknesses.

    - Programming conventions: The implementation must be consistent with the architecture that guides it and consistent internally. That's the point of construction guidelines for variable names, class names, routine names, formatting conventions, and commenting conventions.
    - Location on the tech wave: Determines what approaches will be effective—or even possible.
    - Selection of construction practices: Defined coding conventions for names, comments, and formatting?, identified your location on the technology wave and adjusted your approach to match?, defined an integration procedure (for checking code into master), programmers write test cases for their code before writing the code itself?, selected a revision control tool?, etc.
        - Choose the practices that are best suited for your project.

Chapter 5 (Design)

- The phrase "software design" means the conception, invention, or contrivance of a scheme for turning a specification for a computer program into an operational program.
    - Design is the activity that links requirements to coding and debugging.
    - You have to "solve" the problem once in order to clearly define it and then solve it again to create a solution that works.
- Design process is sloppy because it's hard to know when your design is "good enough.", how much detail to include, formal design notation or just comments, when to stop, etc.
- Also, design is about trade-offs, restrictions, it is non-deterministic (even with the same problem space, people can come up with different designs), heuristic process rather than repeatable one with predictable results, and emergent in that they evolve and improve through design reviews, informal discussions, experience writing the code itself, and experience revising the code itself.
- Design helps to manage complexity. Projects fail most often because of poor requirements, poor planning, or poor management. The software is allowed to grow so complex that no one really knows what it does.
- Ineffective designs come from
    - A complex solution to a simple problem
    - A simple, incorrect solution to a complex problem
    - An inappropriate, complex solution to a complex problem
- Solutions are to
    - Minimize the amount of essential complexity that anyone's brain has to deal with at any one time.
    - Keep accidental complexity from needlessly proliferating.
- Internal design characteristics
    - Minimal complexity, ease of maintenance, minimal connectedness, extensibility, reusability, high fan in (high number of classes that use a given class), low-to-medium fan-out, portability, leanness, stratification.

- Levels of design: software system, divide into subsystems (DB, UI, business logic, communication between the subsystems, etc), divide into classes, divide into data and routines, and internal routine design.
- Design heuristics: find real world objects and know their attributes and relations, form consistent abstractions, encapsulate implementation details, use inheritance, hide information (A good class interface is like the tip of an iceberg, leaving most of the class unexposed.)
- Coupling describes how tightly a class or routine is related to other classes or routines. The goal is to create classes and routines with small, direct, visible, and flexible relations to other classes and routines (loose coupling).
    - Good coupling between modules is loose enough that one module can easily be used by other modules.
    - Try to create modules that depend little on other modules.
    - Ex) A routine like sin() is loosely coupled because everything it needs to know is passed in to it with one value representing an angle in degrees. A routine such as InitVars(var 1, var2, var3, ..., varN) is more tightly coupled because, with all the variables it must pass, the calling module practically knows what is happening inside InitVars(). Two classes that depend on each other's use of the same global data are even more tightly coupled.
- Evaluation of coupling:
    - Size refers to the number of connections between modules. A routine that takes one parameter is more loosely coupled to modules that call it than a routine that takes six parameters.
    - Visibility refers to the prominence of the connection between two modules. Make the connections as obvious as possible. Passing data in a parameter list is making an obvious connection and is therefore good. Modifying global data so that another module can use that data is a sneaky connection and is therefore bad.
    - Flexibility refers to how easily you can change the connections between modules.
- Kinds of coupling
    - Simple-data-parameter coupling
        - Two modules are simple-data-parameter coupled if all the data passed between them are of primitive data types and all the data is passed through parameter lists.
    - Simple-object coupling
        - A module is simple-object coupled to an object if it instantiates that object.
    - Object-parameter coupling
        - Two modules are object-parameter coupled to each other if Object1 requires Object2 to pass it an Object3. This kind of coupling is tighter than Object1 requiring Object2 to pass it only primitive data types.
    - Semantic coupling

        - The most insidious kind of coupling occurs when one module makes use, not of some syntactic element of another module, but of some semantic knowledge of another module's inner workings.
        - Examples
            - Module1 passes a control flag to Module2 that tells Module2 what to do.
            - Module2 uses global data after the global data has been modified by Module1.
- Design patterns reduce complexity by providing ready-made abstractions (You can say let's use Factory method and devs know what you're talking about).

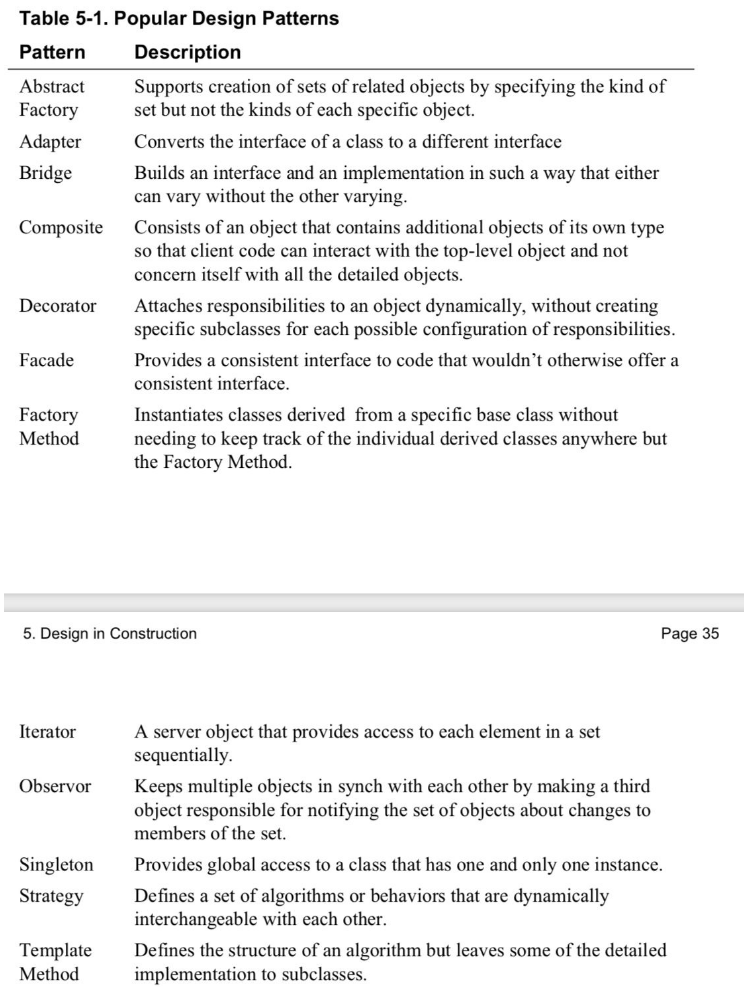


Table 5-1. Popular Design Patterns

Pattern	Description
Abstract Factory	Supports creation of sets of related objects by specifying the kind of set but not the kinds of each specific object.
Adapter	Converts the interface of a class to a different interface
Bridge	Builds an interface and an implementation in such a way that either can vary without the other varying.
Composite	Consists of an object that contains additional objects of its own type so that client code can interact with the top-level object and not concern itself with all the detailed objects.
Decorator	Attaches responsibilities to an object dynamically, without creating specific subclasses for each possible configuration of responsibilities.
Facade	Provides a consistent interface to code that wouldn't otherwise offer a consistent interface.
Factory Method	Instantiates classes derived from a specific base class without needing to keep track of the individual derived classes anywhere but the Factory Method.
Iterator	A server object that provides access to each element in a set sequentially.
Observor	Keeps multiple objects in synch with each other by making a third object responsible for notifying the set of objects about changes to members of the set.
Singleton	Provides global access to a class that has one and only one instance.
Strategy	Defines a set of algorithms or behaviors that are dynamically interchangeable with each other.
Template Method	Defines the structure of an algorithm but leaves some of the detailed implementation to subclasses.

- Top down vs bottom up design approaches
    - The guiding principle behind the top-down approach is the idea that the human brain can concentrate on only a certain amount of detail at a time. If you start with general classes and decompose them into more specialized classes step by step, your brain isn't forced to deal with too many details at once.
        - The strength of top-down design is that it's easy. People are good at breaking something big into smaller components.
        - You can defer construction details.

    - Sometimes the top-down approach is so abstract that it's hard to get started. If you need to work with something more tangible, try the bottom-up design approach. Ask yourself, "What do I know this system needs to do?" Undoubtedly, you can answer that question. You might identify a few low-level responsibilities that you can assign to concrete classes. Then you can look from the top.
        - Results in early identification of needed utility functionality, which results in a compact, well- factored design.
        - But sometimes you find that you can't build a program from the pieces you've started with.
- The key difference between top-down and bottom-up strategies is that one is a decomposition strategy and the other is a composition strategy. Both approaches have strengths and weaknesses.
- Experimental prototyping is writing the absolute minimum amount of throwaway code that's needed to answer a specific design question.
    - Can work poorly when developers aren't disciplined about writing the absolute minimum of code needed to answer a question, the design question is not specific enough, and devs do not treat the code as throwaway code (they think it might end up in production).

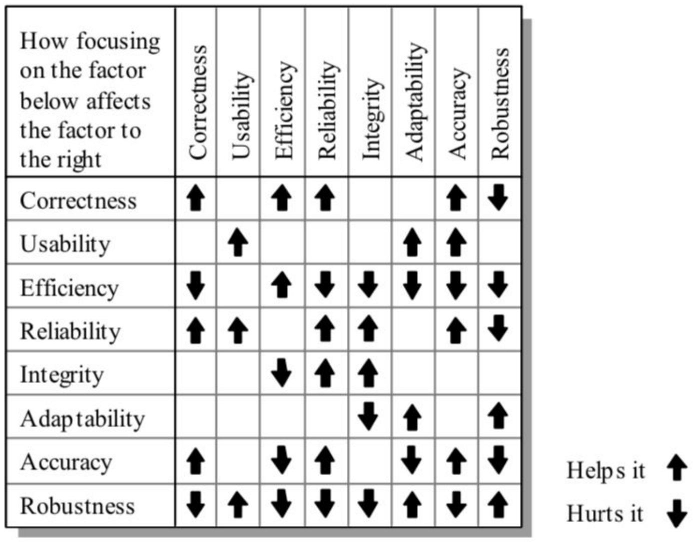
Chapter 6 (Classes)

- A class is a collection of data and routines that share a cohesive, well-defined responsibility.
- Benefits of abstract data types: hide implementation details, changes don't affect the whole program, make the interface more informative, easier to improve performance, don't have to pass data all over your program
- Good class interfaces.
    - Basically creating a good abstraction for the interface to represent and ensuring the details remain hidden behind the abstraction.
    - A class interface should hide something—a system interface, a design decision, or an implementation detail.
- Abstraction helps to manage complexity by providing models that allow you to ignore implementation details. Encapsulation is the enforcer that prevents you from looking at the details even if you want to. Without encapsulation, abstraction tends to break down.
- Containment is the simple idea that a class contains a primitive data element or object (has a relationship). Inheritance is the complex idea that one class is a specialization of another class (is a relationship). It adds a lot of complexity though.
    - Containment is usually preferable to inheritance unless you're modeling an "is a" relationship.
- Barbara Liskov argued (Liskov Substitution Principle) that you shouldn't inherit from a base class unless the derived class truly "is a" more specific version of the base class.

- Reasons to create a class: Model real-world objects, model abstract objects, reduce complexity, isolate complexity, hide implementation details, limit effects of changes, hide global data, facilitate reusable code,

<u>Chapter 20 (Software Quality)</u>

- External quality characteristics: correctness, usability, efficiency, reliability, integrity, adaptability, accuracy, robustness.
- Internal quality characteristics: maintainability, flexibility, portability, reusability, readability, testability, understandability.
- Not all quality-assurance goals are simultaneously achievable. Explicitly decide which goals you want to achieve, and communicate the goals to other people on your team.



How focusing on the factor below affects the factor to the right	Correctness	Usability	Efficiency	Reliability	Integrity	Adaptability	Accuracy	Robustness
Correctness	↑		↑	↑			↑	↓
Usability		↑				↑	↑	
Efficiency	↓		↑	↓	↓	↓	↓	↓
Reliability	↑	↑		↑	↑		↑	↓
Integrity			↓	↑	↑			
Adaptability					↓	↑		↑
Accuracy	↑		↓	↑		↓	↑	↓
Robustness	↓	↑	↓	↓	↓	↑	↓	↑

Helps it ↑
Hurts it ↓

- To improve software quality, software-quality objectives, explicit quality-assurance activity, testing strategy, software-engineering guidelines, informal and formal technical reviews, change-control procedures,

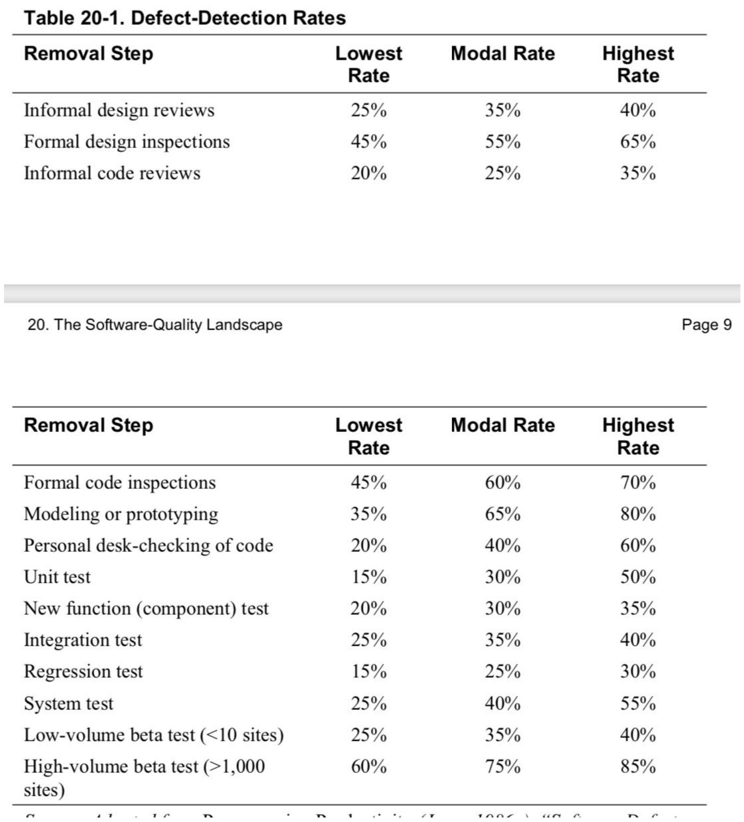


Table 20-1. Defect-Detection Rates

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

20. The Software-Quality Landscape Page 9

- Defect-detection methods work better in combination than they do singly.
    - No single defect-detection technique is effective by itself. Testing by itself is not effective at removing errors. Successful quality-assurance programs use several different techniques to detect different kinds of errors.
- You can apply effective techniques during construction and many equally powerful techniques before construction. The earlier you find a defect, the less damage it will cause.
- There are costs associated with trying to find defects as well as costs with trying to fix them.
- The best way to improve productivity and quality is to reduce the time spent reworking code, whether the rework is from changes in requirements, changes in design, or debugging.

<u>Chapter 21 (Collaboration)</u>

- All collaborative construction techniques are attempts to formalize the process of showing your work to someone else for the purpose of flushing out errors.
- Pair programming one programmer types in code at the keyboard, and another programmer watches for mistakes and thinks strategically about whether the code is being written right and whether the right code is being written.
- A formal code inspection is a specific kind of review that has been shown to be extremely effective in detecting defects and to be relatively economical compared to testing.

Chapter 22 (Testing)

- Types of testing
  - Unit testing is the execution of a complete class, routine, or small program. Tested in isolation from the complete system.
  - Component testing is the execution of a class, package, small program, or other program element
  - Integration testing is the combined execution of two or more classes, packages, components, subsystems
  - Regression testing is the repetition of previously executed test cases for the purpose of finding defects in software that previously passed the same set of tests.
  - System testing is the execution of the software in its final configuration, including integration with other software and hardware systems. It tests for security, performance, resource loss, timing problems.
- Testing is usually broken into two broad categories: black box testing and white box (or glass box) testing. “Black box testing” refers to tests in which the tester cannot see the inner workings of the item being tested. “White box testing” refers to tests in which the tester is aware of the inner workings of the item being tested.
- Testing is a means of detecting errors. Debugging is a means of diagnosing and correcting the root causes of errors that have already been detected.
- Testing is hard/annoying because goal runs counter to the goals of other development activities, can never completely prove the absence of errors, does not improve software quality, have to assume that you’ll find errors in your code.
- Approach to testing
  - Test for each relevant requirement to make sure that the requirements have been implemented.
  - Test for each relevant design concern to make sure that the design has been implemented.
  - Use “basis testing” to add detailed test cases to those that test the requirements and the design.
- We should test first because
  - Detect defects earlier and you can correct them more easily.
  - First think at least a little bit about the requirements and design before writing code
  - Exposes requirements problems sooner
- Also need to do data flow testing. Data can be in 3 forms
  - Defined - The data has been initialized, but it hasn’t been used yet.
  - Used
  - Killed - The data was once defined, but it has been undefined in some way.

- Also, you can describe the variable as entered when control flow enters the routine immediately before the variable is acted upon and exited when control flow leaves the routine immediately after the variable is acted upon.
- Testing tools: diff tools, random data generators, coverage monitors, debuggers, perturbers, error databases, etc.
- Errors tend to cluster in a few error-prone classes and routines. Find that error-prone code, redesign it, and rewrite it.

Chapter 23 (Debugging)

- Debugging is identifying root cause of error and correcting it.
- Debugging process
    - Stabilize the error - Stabilizing an error usually requires more than finding a test case that produces the error. It includes narrowing the test case to the simplest one that still produces the error.
    - Locate the source
    - Understand the problem
    - Fix the defect
    - Test the fix
    - Look for similar errors

Chapter 24 (Refactoring)

- Reasons to refactor: routine is too long, code is duplicated, class has poor cohesion, changes require parallel modifications to multiple classes, global variables are used
- Data level refactoring: replace a magic number with a named constant, rename a variable to be more clear, replace an expression with a routine
- Statement level refactoring: decompose a boolean expression, return as soon as you know the answer, replace conditionals with polymorphism
- Routine level refactoring: extract a routine, move a routine's code inline, add a parameter.
- Class implementation refactoring: change value objects to reference objects, replace virtual routines with data initialization.
- Class interface refactoring: move a routine to another class, convert one class to two
- System level refactoring: create a definitive reference source for data you can't control, provide a factory method rather than a simple constructor

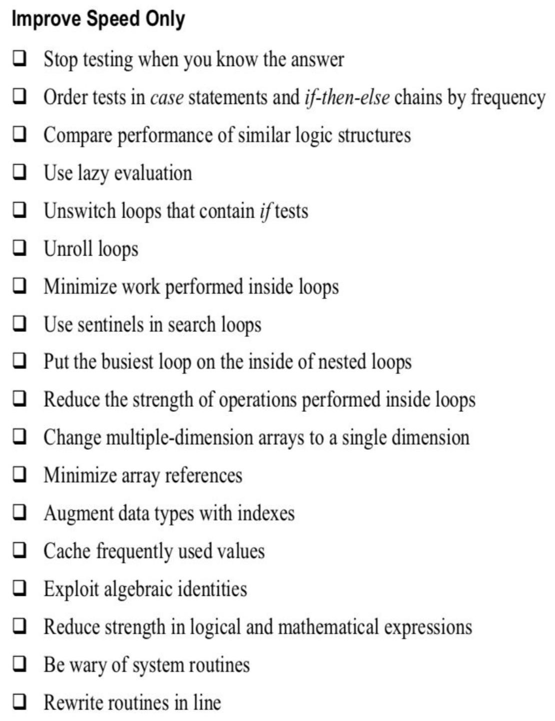
Chapter 25 (Tuning)

- Code tuning is the practice of modifying correct code in ways that make it run more efficiently.
- Sources of inefficiency are IO operations, swapping pages of memory, system calls, interpreted languages, errors.

- Quantitative measurement is a key to maximizing performance. It's needed to find the areas in which performance improvements will really count, and it's needed again to verify that optimizations improve rather than degrade the software.

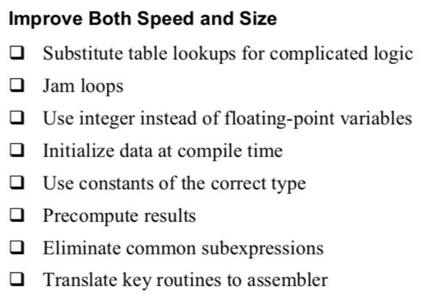
Chapter 26 (Code Tuning Techniques)

- Code tuning refers to small-scale changes rather than changes in larger-scale designs.
    - These changes degrade the internal structure in exchange for gains in performance.
- Code tuning has the following categories
    - Logic
        - Stop Testing When You Know the Answer - Add breaks, look at conditionals carefully, etc
        - Order Tests by Frequency - Arrange tests so that the one that's fastest and most likely to be true is performed first. It should be easy to drop through the normal case.
        - Compare Performance of Similar Logic Structures
        - Substitute Table Lookups for Complicated Expressions
    - Loops
        - Unswitching - Switching refers to making a decision inside a loop every time it's executed. If the decision doesn't change while the loop is executing, you can unswitch the loop by making the decision outside the loop.
        - Jamming, or "fusion," is the result of combining two loops that operate on the same set of elements.
        - Unrolling
        - Putting the Busiest Loop on the Inside
    - Data Transformation
        - Use Integers Rather Than Floating-Point Numbers
        - Use the Fewest Array Dimensions Possible
        - Caching



Improve Speed Only

- ❑ Stop testing when you know the answer
- ❑ Order tests in *case* statements and *if-then-else* chains by frequency
- ❑ Compare performance of similar logic structures
- ❑ Use lazy evaluation
- ❑ Unswitch loops that contain *if* tests
- ❑ Unroll loops
- ❑ Minimize work performed inside loops
- ❑ Use sentinels in search loops
- ❑ Put the busiest loop on the inside of nested loops
- ❑ Reduce the strength of operations performed inside loops
- ❑ Change multiple-dimension arrays to a single dimension
- ❑ Minimize array references
- ❑ Augment data types with indexes
- ❑ Cache frequently used values
- ❑ Exploit algebraic identities
- ❑ Reduce strength in logical and mathematical expressions
- ❑ Be wary of system routines
- ❑ Rewrite routines in line



Improve Both Speed and Size

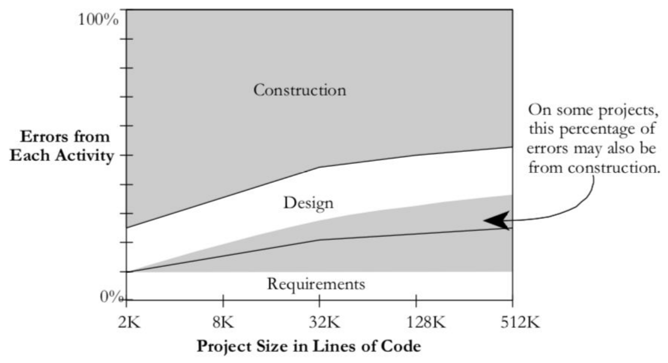
- ❑ Substitute table lookups for complicated logic
- ❑ Jam loops
- ❑ Use integer instead of floating-point variables
- ❑ Initialize data at compile time
- ❑ Use constants of the correct type
- ❑ Precompute results
- ❑ Eliminate common subexpressions
- ❑ Translate key routines to assembler

- Results of optimizations vary widely with different languages, compilers, and environments. Without measuring each specific optimization, you'll have no idea whether it will help or hurt your program.

Chapter 27 (Program Size and Construction)

- As the number of people on a project increases, the number of communication paths increases too. The more communication paths you have, the more time you spend communicating and the more opportunities are created for communication mistakes.

Both the quantity and the kinds of errors are affected by project size. You might not think that the kinds of errors would be affected, but as project size increases, a larger percentage of errors can usually be attributed to mistakes in requirements and design. Here's an illustration:



- It indicates that the number of errors increases dramatically as project size increases, with very large projects having up to four times as many errors per line of code as small

projects. The data also implies that up to a certain size, it’s possible to write error-free code; above that size, errors creep in regardless of the measures you take to prevent them.
- At small sizes (2000 lines of code or smaller), the single biggest influence on productivity is the skill of the individual programmer (Jones 1998). As project size increases, team size and organization become greater influences on productivity.
- Construction activities dominate small projects. Larger projects require more architecture, more integration work, and more system testing to succeed.
    - Construction becomes less predominant because as project size increases, the construction activities—detailed design, coding, debugging, and unit testing—scale up proportionally but many other activities (communication, planning, management, requirements development, architecture, integration, etc) scale up faster.
- Development of a system is more complicated than the development of a simple program because of the complexity of developing interfaces among the pieces and the care needed to integrate the pieces.
- All other things being equal, productivity will be lower on a large project than on a small one and a large project will have more errors per line of code than a small one.

Chapter 28 (Managing Construction)

- Good coding practices can be achieved either through enforced standards or through more light-handed approaches.
- Configuration management is the practice of identifying project artifacts and handling changes systematically so that a system can maintain its integrity over time. Another name for it is “change control.”
    - It includes techniques for evaluating proposed changes, tracking changes, and keeping copies of the system as it existed at various points in time.
    - Another configuration-management issue is controlling source code. If you change the code and a new error surfaces that seems unrelated to the change you made, you’ll probably want to compare the new version of the code to the old in your search for the source of the error.
- Estimating a Construction Schedule
    - Establish objectives, Allow time for the estimate, and plan it, Re-estimate periodically
- Measurement

Chapter 30 (Programming Tools)

- Design Tools
    - Current design tools consist mainly of graphical tools that create design diagrams.

    - Graphical design tools generally allow you to express a design in common graphical notations—UML, architecture block diagrams, hierarchy charts, entity relationship diagrams, or class diagrams.
- Source Code Tools
    - Integrated Development Environments (IDEs)
    - Diif, merge, prettier tools
    - Syntax, semantics checkers, metrics reporters
    - Version control
- Executable Code Tools
    - Compilers convert source code to executable code. A standard linker links one or more object files, which the compiler has generated from your source files, with the standard code needed to make an executable program.
    - Make - The purpose of make is to minimize the time needed to create current versions of all your object files. For each object file in your project, you specify the files that the object file depends on and how to make it.
    - Code libraries
- Stuff for debugging and testing
- Code Tuning Tools
    - An execution profiler watches your code while it runs and tells you how many times each statement is executed or how much time the program spends on each statement.
    - Some day you might want to look at the assembler code generated by your high-level language. Some high-level–language compilers generate assembler listings.

<u>Chapter 31 (Layout and Style)</u>

- Code layout affects how easy it is to understand the code, review it, and revise it months after you write it. They also affect how easy it is for others to read, understand, and modify once you're out of the picture.
- The Fundamental Theorem of Formatting is that good visual layout shows the logical structure of a program.
    - If one technique shows the structure better and another looks better, use the one that shows the structure better.
- The smaller part of the job of programming is writing a program so that the computer can read it; the larger part is writing it so that other humans can read it.
- Good layout objectives
    - Accurately and consistently represent the logical structure of the code
    - Improve readability
    - The best layout schemes hold up well under code modification. Modifying one line of code shouldn't require modifying several others.
- Layout techniques
    - Use whitespace, grouping, blank lines, indentation, one statement on each line.

Chapter 32 (Self Documenting Code)

- Documentation on a software project consists of information both inside the source-code listings and outside them—usually in the form of separate documents or unit development folders. On large, formal projects, most of the documentation is outside the source code.
    - In contrast to external documentation, internal documentation is found within the program listing itself. It’s the most detailed kind of documentation, at the source-statement level. Because it’s most closely associated with the code, internal documentation is also the kind of documentation most likely to remain correct as the code is modified.
- Types of comments
    - Repeat of the Code, Explanation of the Code, Marker in the Code (TODOs), Summary of the Code, Description of the Code’s Intent,

Chapter 33 (Personal Character)

- The characteristics that matter most are humility, curiosity, intellectual honesty, creativity and discipline, and enlightened laziness.
- The characteristics of a superior programmer have almost nothing to do with talent and everything to do with a commitment to personal development.