Lecture 4

Functional requirements - requirements that are concerned with what the system should do/how it should behave

Quality Requirements - requirements that are concerned with how well the system supports functionalities

Modifiability is a quality attribute that is more concerned with "fitness for future" as opposed to "fitness for purpose" like functionality, performance, reliability and security. Think of modifiability as the foundation for balancing the other qualities. Often **maintainability** is used to describe this as well. A **module implementation** is the secrets hidden in the module.The **axiom of independence** states that a specific module can be adjusted to satisfy its requirement without effecting other requirements.

Modularity is the key to achieving modifiability. A **module** is an unit of the system with a defined purpose and interface. They allow:

- better understanding of system pieces
- parallel development
- evolution by hiding implementation details (limits ripples of changes)
- independent compilation

Interfaces are contracts between modules and their environments. Interfaces should be based on abstractions that are unlikely to change (shit gets crazy when interfaces change). A **provided interface** tells what services the module provides its environment. A **required interface** tells what services the module requires of its environment. A **syntactic interface** specifies how to invoke a modules provided services (usually in class declarations). A **semantic interface** specifies how a module behaves (usually in comments).

Organization Smaller simpler modules are usually preferred wince they will use less dependencies. These are also easier to implement and maintain. Often large systems are modularized into a hierarchy so that we can make larger modules out of smaller ones. Keep your mappings of module names in the hierarchy simple. **Crosscutting** is when a module has dependencies at different levels of the hierarchy which can get confusing. **Tangling** is when a module satisfies requirements across the hierarchy. Try to avoid these.

Analysis of modifiability can be improved by increasing coupling (clearer rules, more abstraction of common services, localize changes), or by decreasing coupling(reduce ripple effects, no globals, dependency injection)

Lecture 5

Cohesion is a measure of the coherence of a module amongst its other pieces (GOOD).

Coupling is the degree of relatedness between modules (BAD).

Good Cohesion means that each module has a clear purpose (responsibility). Modules gather like purposed operations and classes. A good design has clear rules where to put new code during its evolution. Good cohesion happens when there is a clear purpose for each module, function are related by topic or interaction, and modules are abstractions.

Bad Cohesion happens through:

- coincidental cohesion - when code is put in a random module (can happen when design is not clear)
- god classes - when a class has too many responsibilities, often has non communicating behaviors(functions that share no data or results)
- control flow cohesion - when code is merged into a single module to share common flow

- and temporal cohesion.all operation executed at a similar point in time into a module (leads to duplication, and complicated logic)

Coupling happens due to different types of dependencies:

- data - A provides data to B
- control - A controls execution of B
- service - A calls a service in B
- identity - A knows of target module B
- location - A knows the location of target module B
- quality of service - A expects certain quality of service from B
- content - A includes code from B

You can minimize coupling by minimizing the number of dependencies among modules (dependencies within module is fine) or minimizing the strength of the dependencies (data is weakest). Its OK to have more dependencies on stable interfaces, but frequent interface changes should have less dependencies.

High Coupling is bad. Hidden coupling is even worse, everything should be explicit and in interfaces (I'm looking at you JavaScript). This can be caused by mutable globals and reflection. **Implementation inheritance** causes strong coupling and lacks a clear interface, inheriting interfaces is much more desirable since it doesnt suffer the "fragile base class" problem (base class can break everything). **Control coupling** occurs when one module controls execution flow of another usually through flags, should try to use polymorphism instead. **Stamp coupling** occurs when more data is passed to a module than is necessary, it introduces more dependency than is needed.

Lecture 6

Refactoring is improving design without changing functionality.

Technical Debt and Incremental Design **Incremental Design** is looking at refactoring anytime anything new is added or fixed. By frequently refactoring to avoid building up design problems which lead to architectural erosion. **Technical debt** is the accumulation of design problems anytime something new is added without refactoring. Better to floss every night than have to get a root canal.

Test Drive Design is writing tests for a design before you start coding. Refactoring requires a comprehensive test suite to make sure you don't introduce new bugs. Tests should be written based on the requirements so they capture what should happen. Tests can also help flush out a design or interface.

Code Smells indicate possible design problems, usually they trigger a refactor.

Duplicated Code:

- Disadvantages
  - parallel maintenance - must propagate changes to multiple places
  - increases the amount of code to be maintained
- Advantages
  - independent development - evolve code in multiple different directions without breaking other's work
  - simplify things - clone can be specialized for target use making it complicated to refactor
  - increase performance - optimized for specific uses
  - no coupling

- Ways to reduce:
  - extract method - pullout duplicated code into shared method
  - extract class - make new common class to be inherited from
  - template method

Lecture 7

Long Method Methods that are too long should usually be factored out into smaller methods called the compose methods.

Large Class A class that has too many methods likely has too many responsibilities. Break it up into smaller classes or push parts of code into attributes. Some exceptions:

- math utility classes - just a group of functions and not really OOP
- library classes - cater to more usage context
- monitors - needs operations for all controllers
- constructor methods - may need many of these, they don't count to a classes method count

Comments Comments inside methods imply opaque implementations. You should refactor the code to make it more self-evident.

Switch Statement These should usually be replaced with polymorphism. If you are dealing with data type you can used pattern matching. Anytime a new subtype is added you have to go edit every switch statement where it is used and recompile every subtype.

Primitive Obsession This is when a developer tries to manipulate primitives to serve a purpose where a custom class would be better suited. This way we can encapsulate related functions and expand much more easily in the future.

Long Parameter List This makes it difficult for clients to understand. Can be caused by:

- a method having too many responsibilities - break it into subtasks
- method is operating on data that doesn't belong to it - move method to be with data owner
- method takes too many disparate member variables - gather parameters into an object

Feature Envy This is where a mthod is more interested in another class and should be moved into that class.

Data Clumps This is where a set of variables hang out together alot (are frequently passed as parameters together, or accessed at the same time), so they should be encapsulated together into an object.

Shotgun Surgery This is when a single change requires tones of changes to other unrelated things. These changes should be localized into one thing.

Direct Constructor Call This often leads to brittleness. Calls to constructor go wonky when subclasses get involved so you should use a factory method instead.

Speculative Generality This is when code is extended with shit because you might need it someday. This is bad as things will get complex for no reason at all. Instead you should add things when they are needed and evolve the design as required.

Lecture 8

A design patter is a solution to a re-occurring design pattern (used at least 3 different systems). These are ways to package pieces of reusable design knowledge. They usually consist of:

- design context and problem
- design solution
  - structural part made of elements and their roles and relationships/interactions
  - usually a UML
- design consequences
  - impacts on system qualities
  - implementation alternatives
- examples

If a pattern relies of language specific constructs it is called a **idiom**.

Composite Pattern Create a common interface for all elements (no matter their place in the hierarchy) so that all elements and their compositions can be treated the same. Basically we want to treat all individual objects and multiple recursively composed objects exactly the same. You want a component class that has operations used by individuals and collections, then you have a leaf class and composite class that inherit from it.

Decorator Pattern This is used when we want the ability to augment object with new responsibilities without creating a subclass. Basically you have an interface with a subclass for core functionality and a decorator subclass that contains an instance, which your concrete object class inherits from. The functions in the decorator class calls the same function on its instance which will then call the correct function from one of its subclasses. This function will add some functionality then call its super class function (defined in core functionalty class).

Lecture 9

Interpreter Pattern This provides a way to translate a language grammar into an object structure so that operations can be implemented. It has a abstract class for an expression (something composed of more specific parts combined) and from that class you have a subclass for terminal expressions and nonterminal expressions. All of these classes have some function for interpreting them that is called down the tree recursively.

Visitor Pattern This encapsulates operations on the object structure into object so that you can easily add new operations without disturbing the classes implementing the object structure. A visitor is an object encasing an operation. Visitors visit elements of the structure on which they operate through a common interface. Elements have a accept function that calls the visitors visit function for this element (ie the dog class would call visitor.visitDog(this)).

Other Patterns

- Composite
  - Understand UML composition (black diamond), including its semantics (no sharing, no cycles, nested life span), allowed multiplicities on the diamond, and the shape of the corresponding object diagrams.
  - Understand the tradeoff between uniform interface for leaf and composite vs. navigation and editing logic in composite only.
- Decorator
  - Be able to apply decorator to extend design generated by composite.
  - Understand the issue of split identity.
- Adapter

    - Understand the difference between adapter and decorator. Hint: think about what each pattern does to the interface of the wrapee.

- Interpreter
    - Understand that interpreter builds on composite.
- Visitor
    - Understand how double dispatching occurs.
    - Understand the interface of the visit methods.
    - Understand how visitors are unnecessary in functional programming (replaced by pattern matching).
    - Understand when to use visitor and when interpreter.
- Singleton
    - Understand the issue of a singleton being global state and the consequences (potential side effects, concurrent access).
- Factory
    - Understand the difference between abstract factory and factory method.
- Strategy
    - Understand how strategy supports dynamic reconfiguration thanks to object composition.
- Template Method
    - Understand how to vary steps in template method subclass.
    - Understand the fragile base class problem and why object composition and interfaces are often a better alternative.
- Iterator
    - What does iterator encapsulate?
    - What is the difference between iterator vs. visitor? Hint: what is that each pattern encapsulates?
- Observer
    - Understand implementation variants to access subject's data: getting full update vs. selective registration and update.

Lecture 10

An architecure pattern is a solution to a re-occuring architectureal problem (same def as design pattern).

It follows the same pattern as design patterns:

- problem description
- solution
    - structures
        - component types
        - connector types
        - topology - contraints on how things are connected
    - interactions
- consequences

Layered Architecture

Each layer is a different level of abstraction. A layer provides for the layer above it an relies on the layer below it. Can be represented as a stack or onion layers.

Structures:

- components
  - modules
  - interfaces
- connectors
  - function calls and callbacks
  - messages
  - queries
  - etc
- topology
  - hierarchy
  - pure form - communications between adjacent layers only

A **hierarchy** topology allows layers to talk up an down a complicated tree of relationships. A **pure form** allows only adjacent layers to talk to each other.

Advantages of pure style:

- better abstraction - layers can directly build on each other
- portability - you can swap out lower layers easily

Disadvantages of pure style:

- performance - we have to communicate through layers
- limited functionality

We often use a hybrid called **cross-layer** which allows layers to be oddly shaped to give them access to layers above and blow them in different fashions.

Dependencies should really only run down. Having low layers rely on high layers can make things unstable quickly.

Exception handling Exeptions need to bubble up through the layers when they are not properly handled. Ideally an exception should be handled in the same layer it is raised in (that is where the necessary data is, upper layers may not be expecting it, and passing shit increases coupling).

Consequences:

- advantages
  - better understand system by extraction
  - insulation from changes in lower levels
  - levels can be swapped out
- disadvantages
  - performance problems
  - not suitable for all systems

Tiered architectures and interpreters are special forms of layered architecture

Interpreters

These are often used when executing computer programs and are subject to similar consequences as layered architectures. A virtual machine is a special kind of interpreter to convert stuff. These have something called a **hypervisor** that handles the set up and tear down of the VM.

Tiered Architecture

These are layered architectures that are often used in enterprise systems.

Broken up into tier layers:

- data management
- business logic
- presentation

Often data management and business logic are combined into one tier making it a two tier system. We really like to decouple the UI from basically everything else because it has a difference audience and different triggers

Some design patterns used:

- domain logic
  - transaction script
  - table module
  - domain model
  - service layer
- Data source
  - table data gateway
  - row data gateway
  - active record
  - data mapper

You can break things up into as many tier as you want, but shit with similar functionality needs to be coupled and you need a concrete order of tiers.

Lecture 11

Call and Return Architecture

These are the architectures standard in code (main program, abstract classes, OO, functional).

Main program Architecture:

- hierarchy - decomposition into subroutines
- components - procedures
- connectors - procedure calls

Abstract classes Architecture:

- data structures abstracted into set of operations
- components - ADTs
- connectors - function calls and messages

Object Oriented Architecture:

- ADTs with inheritance and polymorphism
- strengths
  - domain modeling
  - abstract problems are easier to understand
  - interfaces and hierarchies
- weaknesses
  - high coupling
  - side-effects
  - complex interactions

Functional Programming Architecture:

- Purity = immutable data

Interacting Processes

Structure:

- components
  - lightweight processes (preemptive threads, cooperative coroutines)
  - heavyweight processes (OS processes)
- connectors
  - shared variables
  - messages
  - events
  - co-routine

OS Processes are much more costly than lightweight processes but can be used to build reliable systems as they provide memory protection and can restart other processes selectively.

Concurrent State Machines

- components - state machines executing concurrently
- connectors
  - message queues
  - shared variables

Implicit Invocation

Instead of the standard invoking a procedure directly, we now call it through events and listeners. Systems built around implicit invocation are called **Publish-Subscribe Systems**.

- components - modules with interfaces providing operations and events
- connectors
  - binding between event broadcast and calls
  - direct operation calls

A component registers its interest in an event by binding a procedure to it. The announcer of events do not know which component will be affected by the event. Components cannot make assumptions about the order they will be processed in or what will happen as a result of their event (treat events like concurrent threads).

Message Broker This is also known as an event bus. It stores messages and forwards them to the correct place allowing persistent and transactional messaging. Often used for enterprise application integration.

Data Distribution Service This has direct forwarding with no storage of events. Due to this data sources have metadata describing the quality of the service. This is usually used for real-time information exchange. strengths

- extensibility - add new features by registering them for events
- loose coupling - add and replace components without affecting each other by registering or unregistering

Weaknesses:

- Loss of control - components have no idea what effect they will have by announcing an event, we cannot rely on order of execution, we don't know when things are done executing
- Difficult to debug - same reasons as above

An example of this is the MVC structure used often in UI as the model uses implicitly invocation to update the view:

- Model - contains core functionality of data
- view - interface for displaying data
- controller - handle user input

Lecture 12

Data Centric Architectures

These are architectures organized around a data repository, so structure of data and data management are very important.

Transactional Database Applications These are the database portion of the three tiered architecture. Concurrency is often used to speed up these systems but this causes some problems. Updates can be lost or reads can get trampled. Liveness is also an issue as things can get deadlocked or starved. First there is a request from the outside world it goes through the session which is basically the long running interaction between the client and server (usually has some form of authentication). From there the transactions are grouped together so that sever similar requests are treated the same for efficiency sake. Transactions can be groups of requests from the user to an application (**business transaction**) or from an application to the database (**system transactions**).

Approaches to Concurrency The problems caused by concurrency are approached in a few ways. One is **isolation** where the data is partitioned in such a way that only one agent should access it at a time. Any data that is immutable can be shared without problems, but not all data is immutable. You can use **pessimistic concurrency control** where you use locks for exclusive access, this reduces liveliness, but is preferred where conflicts are frequent or have sever effects. Finally you can use **optimistic concurrency control** where you attempt to detect conflicts (bu having a version marker to dtect inconsistent reads) to improve liveliness but may result in complex merges or lost data.

When a deadlock occurs you can pick a victim to detect them, introduce timeouts to try an prevent it. A victim is any thread that acquires a locked resource so you can acquire all locks required at the beginning to prevent deadlocks. You could also use lock ordering to prevent deadlocks by forcing an order. Usually we want to use the simplest deadlock recovery/prevention schemes.

Transactions ACID properties:

- Atomicity - each step in transition must complete fully or roll back all the way
- Consistency - a system's resource must be in consistent non-corrupt state at start and end of completion
- Isolation - the result of a transaction must not affect any other transactions until concludes successfully
- Durability - any completed transaction must be permanent (must survive a crash)

Transaction issues:

- long transaction - spans multiple requests
- request transaction - spans one request, declare a method as transaction
- keep transactions as short as possible - no longer than 1 request so use offline concurrency to handle ones that span multiple
- late transactions - all updates in a short system transaction at end of business transaction
- lock escalation - when a query affects many records in table the DB and it locks rows to the table

Online locking Online locking is resource intensive since system transactions are connection oriented and the application needs to keep database connections open to locks. This means that connections are a scarce resource and locking them can make it worse. System transactions are ok to be used as business transactions for up to 10 users, past that you need to keep system transactions short to have higher performance.

Unit of Work Here we maintain a list of objects affected by a business transaction, coordinates of writing out, and the resolution of concurrency problems. We keep track of objects that were created, changed, read, or deleted during a business transaction. The UoW marks the end of a transaction by calling **commit** which opens a system transaction, checks for inconsistent reads, writes changes to database, and commits the system transaction. This has to be kept in session state (one per session).

Implementation alternatives for UoW:

- caller registration - have the caller register that an object was mucked with, this puts burden on the user of objects
- object registration - the receiver object registers itself, so the implementers of persistent classes need to place registration calls (good for OO)
- unit of work controller - takes a copy at read time from database and compares with object at commit time, this allows selective updates on only changed fields

Advantages of UoW:

- improve performance by deferring updates to end of business transaction (store cumulative updates)
- batch update optimization - bundle updates into a single unit, single remote call
- updates are ordered for referential integrity
- same sequence of updates reduces chance of deadlock

Alternatives:

- single system transaction with a save on each update - this has poor performance but may be ok on smaller systems
- defer updates to end by keeping track of changed objects in variables
- keep a dirty flag on each object - basically a ghetto UoW

Optimistic Offline Lock Here we allow conflicts to happen and just roll back one of the transactions that caused it. This works well for systems where business transactions span multiple system transactions (so basically all of them). This also prevents lost updates and inconsistent reads as we just let shit collide and fix it later. It works by checking that, since the last time the session loaded a record no other session has changed it (basically looking for threads trampling each other) for each member of the change set during the system transaction update. One way to implement this is to give each record a version number that we compare with the last time we looked here and increment on success. When a fail happens we roll back and try to resolve it by partial discard or merge.

Pessimistic Offline Lock Here we prevent conflicts between transactions by allowing only one of them to access data at a time. This works by determining what type of lock is needed (read vs write since write requires no others to enter but read allows it). From this we create a lock manager that has a table mapping locks to owners and an api for business transactions that uses locks. The api operations take a session id and object id, acquire the lock, then do some shit.

We also have **coarse-grained locks** that locka set of related objects with a single lock, and **implicit locks** that are locks implemented by the infrastructure for you.

Offline locking is nice and lighweight when compared to online locking. It is also **connection-less** so we dont need to keep a database connection for transactions. Offline locking is much more custimizable since we can add status columns or flags and whatnot. This also makes it much easier to filer and use for different resource types.

Blackboard Architecture

This is an architecture in which agents collaborate to solve a single problem on a blackboard that stores the current solution state and triggers processing on agents.

Benefits:

- cooperative problem solving
- dynamic reconfiguratbility
- extensibility

Drawbacks:

- complex interaction result in unpredictablity
- hard to debug for above reason
- high complexity of blackboard

Lecture 13

These are systems where the availability of data controls the computation (not all data is available immediately), the pattern of data flow is explicit, and data flow is the only form of communication between components.

- Components
  - input ports to read
  - output ports to write
  - computational model to do shit inbetween
- connectors: streams
  - un-directional streams
  - synchronous or async
  - buffered or unbuffered

There are three different topologies of data flow:

- arbitrary graph
- linear data flow
- cyclic patterns (ones with loops)

Batch Control Architectures

Components are independent programs. connectors are some type of storage, and each step runs completely to success before next one can start. Basically data is grouped into batches that go through each component one at a time. This is used when data needs to be analyzed in a batch oriented fashion or when you have discrete transactions that occur at periodic intervals (like a system report every five minutes or so)

Pipe and Filter

The components are filters that process data incrementally as it arrives, the connectors are pipes or conduits (many be sync or not, buffered or not). These systems have outputs that begin before the whole input has been consumed (ex, image processing or any sort of streaming). We can either use pull or push architecture to either write data onto the next pipe or read from the previous pipe. Even better we can make pipes concurrent for processing.

Synchronous data flow Each filter expects a certain amount of input and outputs a certain amount by passing the input through a bunch of filters sequentially (usually a line of a bunch of boxes with cardinalities). This allows for checking for balanced execution for scheduling or deadlock checking. We could also have dynamic data flow by conditional firing or Khan process networks that have actors to execute concurrent read/write tokens.

Strengths of pipes:

- easy to understand - explicit process with simple diagrams
- loose coupling - only data coupling
- reuse - order and type doesnt really matter and filters can easily be shared
- extensibility - super easy to add new filters
- amenable to automated analysis - really easy to check for deadlocks and performance analysis as each input and output has expected attributes
- amenable to parallelization - each filter can be its own thread

Weaknesses of pipes:

- restrictive - not all problems fit, for instance it cannot be repose time oriented or fit interactive applications (data flows one way)
- performance issues - filters that do little analysis still require a full copy of the data (pass by reference)
- implementation may force lowest common denominator data type - more complex data types will get spliced when moving between filters
- error handling - filters crashing can take out the whole pipe

Lecture 14

Process Control Architectures

Process control architecture is used to control physical processes and equipment (usually to maintain certain properties or equilibrium). Feedback is when you pipe the output of a system back into the input to stabilize it (see the entire course you're taking on it).

There are multiple kinds of variables:

- control variable - basically the output we want to control
- manipulated variable - the input variable that we can fuck with
- input/confounding variable - input variables that we cannot control
- set point - the value we want the output to be

Hybrid controllers model both continuous and discrete behavior by toggling between controllers (usually via a state machine).

Multi-variable controllers have multiple outputs and feedback loops (ahhhhh whyyyy).

Model-predicitve controllers attempt to predict the output to try to increase performance (uber complex caclucations going down here).

Lecture 15

Multiple Architectural Views

Each view focuses on a different perspective (for different stakeholders) and allows answering questions. We can use these for code generation or documentation so that people can get what the fuck you're trying to do. We document architecture to provide sequential reading order and make sharing much easier. Software architecture has four componenets

- software requirements - what it has to do
- logical software architecture - how to do it
- physical software architecture
- views - link things for system testing

Uses:

- Communication
- Planning
- Requirements elicitation guidance
- Analysis and design guidance
- Reuse support
- Certification

Lecture 16

No lecture, just one big example.

Lecture 17

Lecture 18

Architectural Notations These can be informal such as simple lines and boxes as a quick sketch requiring no tool, or can be very formal using precise notation using tools based analysis to generate tests and skeletons.

UML This is mostly used for architecture modeling. You need case diagrams to represent the context model and functionalities, class diagrams to represent domain models, and sequence/state/activity diagrams to represent behavior. Case diagrams show actors (the various types of users, human or otherwise) and the actions they will take with cardinalities. We can also break it up into use cases representing the series of actions that happen when an actor does something. A class diagram is the standard UML you are used to. A sequence diagram shows the timing of function calls and returns for specific actions. A state diagram shows the transitions between states, their triggers, and timing. An activity diagram show the workflow of actions

- Class diagrams - component types and their connections, shows architecture logical structure
- State diagrams - component states and transitions, show logical behavior

- Activity diagrams - flow of actions in components, show logical behavior
- Sequence diagrams - interactions among components, shows logical behavior
- Component diagrams - rarely used, shows physical structure
- Deployment diagrams - mapping of software artifacts to hardware nodes

SysML This is an extension of UML for systems and software architecture. We try to represent all aspects of the system including hardware. It adds a requirements diagram that shows what components require other components.

Block Diagrams Block definition diagrams represent the structural elements and their composition/classification. We use these to show the domain model (basically a UML class diagram). Internal block diagrams show interfaces among parts (nested blocks). We use it to show connections among the Components

Parametric diagrams are added in SysML to represent constraints on values to support analysis. Package diagrams are also added in UML ti represent model organization and show name spaces

AADL This stands for architectural analysis and description language. It focuses on modeling software and hardware components and their connections. Components also include behavior components (like state and transitions between them). Lots of quality attributes are included in components like timing and performance information.

Lecture 19

What is performance? The amount of usefull work that can be accomplished by a computer compared to time and resources used.

- Response time - amount of time to process a request from the outside
- responsiveness - how quickly the system acknowledges a request
- Latency - minimum time required to get any form of response
- Throughput - how much stuff you can do in the given time (usually in transactions per second)
- Load - how much stress a system is under
- Load sensistivity - how the response time varies with the load
- Efficiency - performance divided by resources
- Capacity - maximum effective throughput or load
- Scalability - how adding resources affects performance
  - vertical - adding more power to a single server
  - horizontal - adding more servers

464

Lecture 20

Performance can be a part of the requirements, or even part of the design criteria.

You can evaluate by defining section of the system for a specific scenario:

- stimulus
- source
- environment
- artifact

- target

When anaylyzing performance we need a goal for the analysis and a subject to analyze (basically whatever the design is at the time, could be just a sketch). We can also analyze it based on the specifications by getting experts to look at it, model it, or simulate it. Further we could straight up look at the code and run performance tests or run static analysis.

Queuing Network Models Anywhere in the system that queuing happen we abstract it to this model. This allows us to better understand the analysis of concurrent systems. We use **Little's Law** which says that the number of users at a node in the queuing network model is equal to the throughput at that node times the response time at that node. The number of concurrent uses is equal to the number of active users times the system response time divided by response time plush think time.

$\mathrm{N}_i = X_i \times R_i$
$N_{concurrent} = N_{active} \times \frac{R}{R+T}$

Lecture 21

Performance is impacted by resource consumption and blocked time.

Some tricks:

- control your resource demand to reduce latency
- manage resources to reduce response time

Lecture 22

Reliability is the ability of a system or component to do its job. Basically not failing in a specific environment. Since failure is a random phenomenon we view reliability as a probability, using statistics as Mean time to failure, failure rate, and probability of success.

Availability is the probability that the system will be available when needed. It is the mean time to failure over the time to failure plus repair time.

Safety is the condition of being protected from harmful conditions or events (ie the software does not cause the harmful conditions). This is not the same as reliability. Unreliable systems can be safe and unsafe systems can be unreliable.

Reliability engineering focuses on the cost of removing all failures verses the cost of dealing with them. Safety engineering is focues on preventing harmful conditions. Reliability is a property of a component and safety is a system level quality.

Failure is when the system no longer delivers its service correctly, A failure is caused by a **fault** (not all faults cause failures). A fault is in the system and a failure is what you see as a result (sometimes, recovery is possible). An **error** is a incorrect state due to a fault.

1 Lecture 23

We define reliability scenarios such as a function must not be down for more than an hour a week or other aspects such as:

- stimulus: bad value or timing
- source: external system, component, network, etc
- artifact: system or component
- response: notification, recovery, failure
- metric: availability, MTTF, etc

Failure mode and effects analysis is a method of analysis that works by identifying the failure modes per function or component and their effects and probabilities. This is a bottom up method where we start with components and work up to the full system. This is a more systematic method of analyzing failures, but it falls apart when there are multiple failures.

Fault tree analysis is a top down decomposition of system level failure into events that trigger it.

Reliability relies on multipe methods of verification:

- testing - cannot prove there are no faults
- static analysis - checks only for certain classes of bugs
- model checking - exhaustive but limited by state explosion
- deductive program verification - generates proof obligations and proves them

Its impossible to get rid of all bugs so we aim for fault tolerance:

- Error detection
- error handling - prevent propagation
- error recovery - redundancy
- fault prevention and removal - keep things simple and verify everything