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.