

Lecture 1

1.1 laws of SA

First Law: Everything is a trade-off

Second Law: Why is more important than How

1.2 what's software architecture?

important design decisions about software structure and the relationship between these structures.

1.3 why are design decisions important?

it supports a set of qualities that a successful system should have. it provides a conceptual basis for system development, support, and maintenance.

1.4 why is software architecture important?

1. Inhibits or enables a system's Quality Attributes: positions the design along different trade-offs
2. Restrict design alternatives 2.1 channels the creativity of developer
3. Focuses attention on the assembly of components 3.1 rather than the components themselves
4. Enhances communication among stakeholders 4.1 customer, user, project manager, coder, tester, etc

1.5 what's design quality?

the efforts required to meet the needs of the customer

1.6 how to evaluate the design quality?

if that effort is low and remains so, it is a good design. Otherwise, it grows with each new release, the design is bad.

UML Diagram

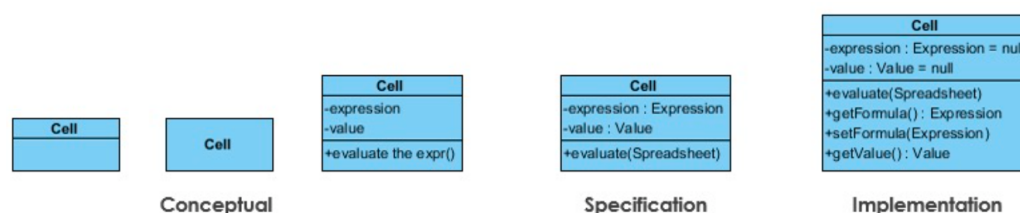
1. Uml class notation

1.1 three partitions (only class name is compulsory) class name, class attributes, class methods

1.2 class visibility: +(public) -(private) #(protected)

1.3 abstract class names need to be italicized

1.4 interface needs to be highlighted by "<<interface>>"



2. relationships between classes

2.1 Association: A has an attribute whose type is B

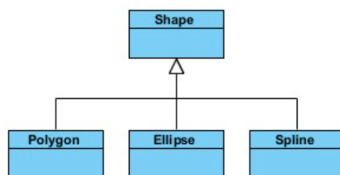




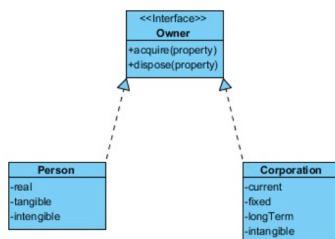
2.2 Dependency: an object of one class might use an object of another class in the code of method. If the object is not stored in any field, then this is modeled as dependency relationship



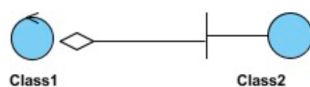
2.3 Inheritance (generalization): a generalization is a relationship between a more general classifier and a more specific classifier. an abstract class name is shown in italics.



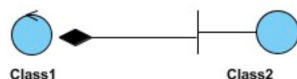
2.4 Realization: a relationship between an interface and the object which implements methods of interface



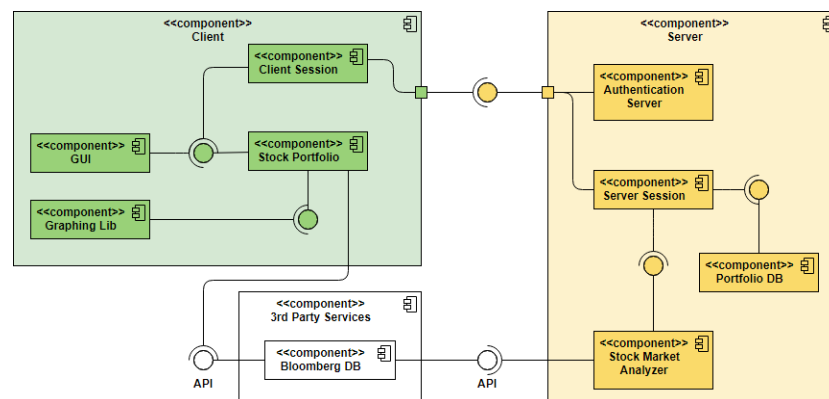
2.5 Aggregation: a special type of association, it represents a "part of" relationship. Class2 is part of Class1. Class1 and Class2 have separate lifecycles (if Class1 is destroyed, Class2 remains unchanged)



2.6 Composition: a special type of aggregation where parts are destroyed when the whole is destroyed. Objects of Class2 live and die with Class1.



2. Component Diagram



Lecture 2

Key points: review object-oriented concepts

1.1 what is a system? systems are collections of objects that collaborate according to rules.

1.2 what is object collaboration?

objects interact with each other by passing messages. a message can be simple query or a request to perform an operation required by the system. this communication is implemented in software by having one object invoke another object's methods

1.3 objects and classes

Objects are things in the world; they have distinct identities, attributes and behaviour.

Objects with similar attributes and behaviour can be grouped into one category, which we call a class.

To put it another way: a class is a way of describing the similarities exhibited by a set of objects

Objects are instances of a particular class.

Two instances of the same class will be described by the same attributes but can still look different because those attributes can have different values.

1.4 dynamic binding: ensures that the appropriate version of method is called in each case.

1.5 abstract class: a class with ≥ 1 abstract method is an abstract class, and cannot be instantiated

1.6 interfaces: Like classes, but they have no fields and their methods are implicitly abstract. Sole purpose is to provide a polymorphic interface through which to manipulate instances of classes that implement the interface. Not subject to inheritance restrictions: a Java or C# class can have only a single superclass but it can implement any number of interfaces.

Lecture 3

Review of UML class diagram and component diagram.

Lecture 4,5,6

1.1 what is unit test?

Testing of individual units of code, in isolation from the rest of the system

1.2 why write unit tests?

- (1). As changes are made over time to realize short-term goals, code loses its structure
- (2). System accumulates technical debt – which, if not addressed, can eventually make it impossible to modify
- (3). 'Paying off' the technical debt is much easier if you do so by 'small instalments' – i.e., frequent small changes

1.3 what is refactor and refactoring

refactor(verb): To restructure software, via a series of refactorings, without changing its observable behaviour.

refactoring(noun): Refactoring is the process of changing a software system in a way that does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence, when you refactor, you are improving the design of the code after it has been written.

1.4 how to do a unit-test?

- (1). sets up some fixture—data and objects that are needed for the test
- (2). write with a placeholder for the expected value, replace the placeholder with the code's actual value
- (3). run tests; make sure they all pass
- (4). inject a fault, revert the fault (always make sure a test will fail when it should.)
- (5). Run tests frequently. Run those exercising the code you're working on at least every few minutes; run all tests at least daily.

1.5 when should we refactor code?

- (1). Before adding a feature: Once a feature is added, put in tests and tidy-up before adding the next feature.
- (2). As we do code review: As well as finding errors, code reviews may advise on refactoring.
- (3). When there's a '**bad smell**' to the code. Typical Code Smells: Duplicate Code, Long Method, Long Parameter List, Temporary Field within object, Switch Statement, Excessive Comments.

1.6 when do we not refactor?

- (1). when structure is so bad that it would be easier to throw the code away and start.
- (2). when the code doesn't function correctly.
- (3). when a hard deadline is very close.

Bad Code Smell

(1). Duplicate Code: Two code fragments look almost identical.

payoff: Merging duplicate code simplifies the structure of your code and makes it shorter. Simplification + shortness = code that's easier to simplify and cheaper to support.

(2). Long Method: A method contains too many lines.

payoff: Among all types of object-oriented code, classes with short methods live longest. The longer a method or function is, the harder it becomes to understand and maintain it. In addition, long methods offer the perfect hiding place for unwanted duplicate code.

(3). Long Parameter List: More than three or four parameters for a method.

payoff: More readable, shorter code. Refactoring may reveal previously unnoticed duplicate code.

(4). Temporary Field: Temporary fields get their values (and thus are needed by objects) only under certain circumstances. Outside of these circumstances, they're empty.

payoff: Better code clarity and organization.

(5). Switch Statements: You have a complex switch operator or sequence of if statements.

payoff: Improved code organization.

(6). Excessive Comments: A method is filled with explanatory comments.

payoff: Code becomes more intuitive and obvious.

Refactorings

(1). Extract Class: Create a new class and move the relevant fields and/or methods from the old class to the new class.

(2). Inline Class: Move all A features to B class, then delete it.

(3). Pull Up Method: You have methods with identical results on subclasses. Move them to the superclass.

(4). Push Down Method: Behaviour on a superclass is relevant only for some of its subclasses. Move it to those subclasses.

Lecture 7,8

1.1 what is design pattern?

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

Design Pattern

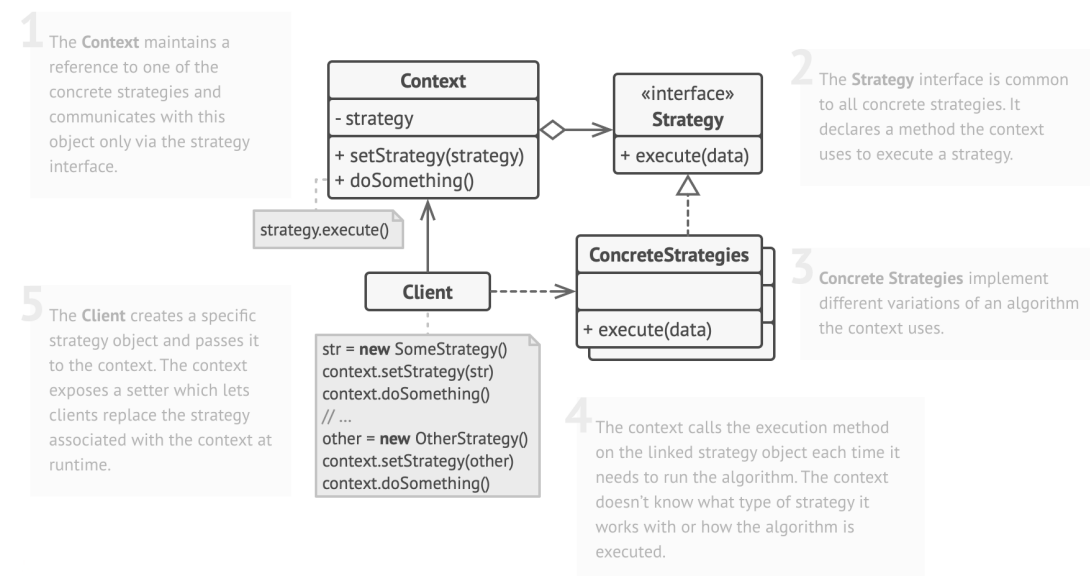
1. The Strategy Pattern

problem: A class can benefit from different business rules or algorithms, depending on the context in which it is used. Particular algorithm or set of rules that are chosen depends upon the class user.

Solution: Abstract the algorithm into an interface, the strategy. Provide algorithms as classes implementing this interface. Have the context class maintain a strategy reference. Have the user's code supply a concrete strategy object to the context class.

Pros: You can swap algorithms used inside an object at runtime. You can isolate the implementation details of an algorithm from the code that uses it. You can replace inheritance with composition. Open/Closed Principle. You can introduce new strategies without having to change the context.

Cons: If you only have a couple of algorithms and they rarely change, there's no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern. Clients must be aware of the differences between strategies to be able to select a proper one.



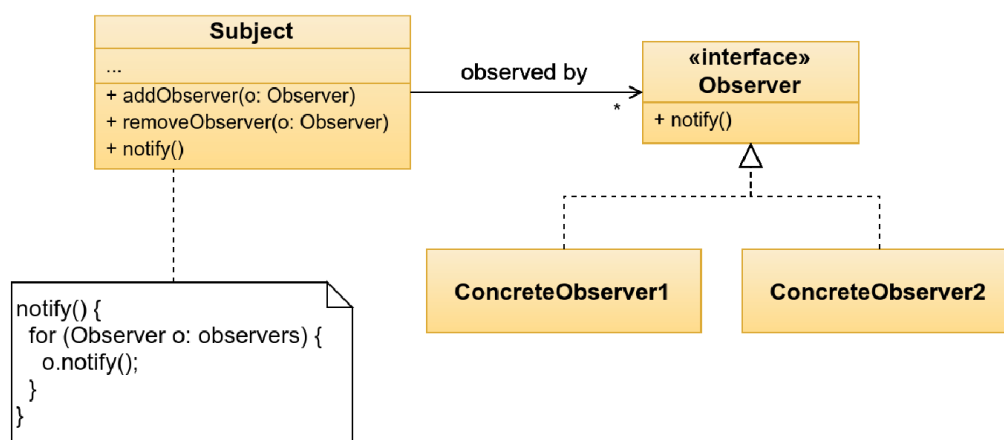
2.The Observer Pattern

problem: A subject is a source of events. One or more observers need to know when events occur.

solution: Define an interface type for observers; concrete classes must implement this and provide a notify method specifying response to event. Subject class maintains a collection of the observers that are interested in its events. When an event occurs, subject must call `notify()` on all registered observers.

Pros: Open/Closed Principle. You can introduce new subscriber classes without having to change the publisher's code. You can establish relations between objects at runtime.

Cons: Subscribers are notified in random order.

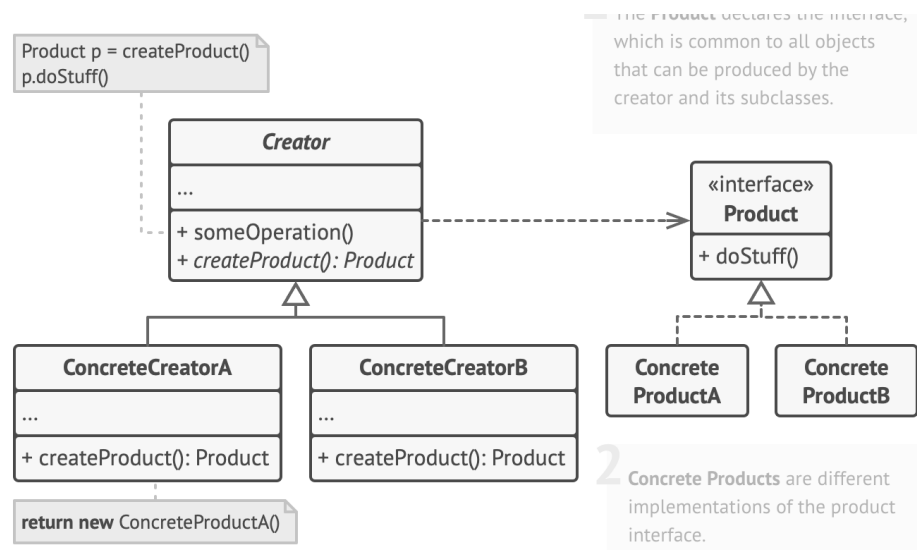


3.The Factory Method Pattern

Pros: You avoid tight coupling between the creator and the concrete products. Single

Responsibility Principle. You can move the product creation code into one place in the program, making the code easier to support. Open/Closed Principle. You can introduce new types of products into the program without breaking existing client code.

Cons: The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.

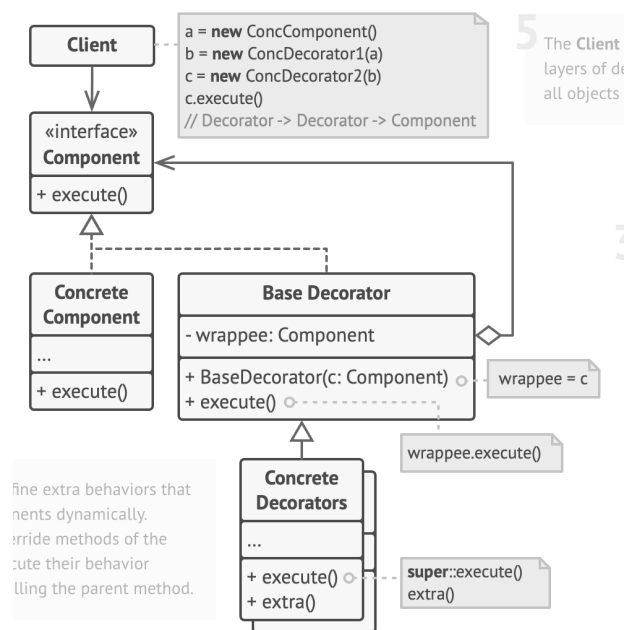


4.The Decorator Pattern

Delegation pattern: a class has methods that forward calls to identical methods supplied by another class. Behaviour of an object is modified dynamically by 'wrapping other objects around it'. Decorator and recipient of decoration have same interface, so decoration is transparent.

Pros: You can extend an object's behavior without making a new subclass. You can add or remove responsibilities from an object at runtime. You can combine several behaviors by wrapping an object into multiple decorators. Single Responsibility Principle. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.

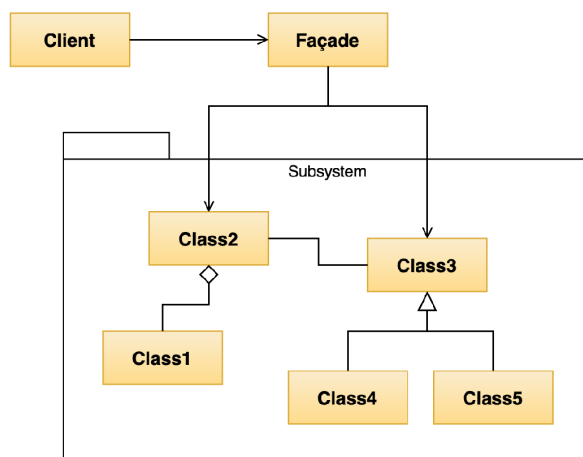
Cros: It's hard to remove a specific wrapper from the wrappers stack. It's hard to implement a decorator in such a way that its behavior doesn't depend on the order in the decorators stack. The initial configuration code of layers might look pretty ugly.



5.The Façade Pattern

Problem: Complex subsystem (multiple classes). Subsystem implementation could change. Clients of subsystem need a simplified view of subsystem and a coherent, stable entry point.

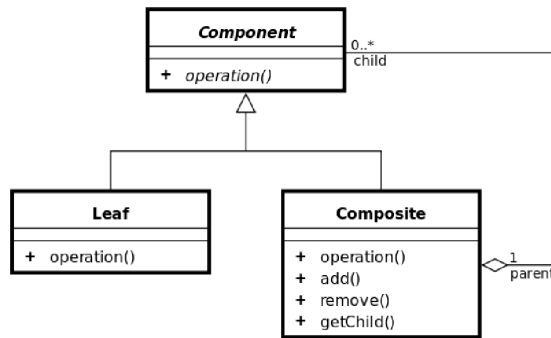
Solution: Define a façade class, providing a new & simpler interface. Façade methods expose desired portion of subsystem functionality to clients, by delegation. Subsystem classes know nothing about façade, and client need know nothing about subsystem classes.



6.The Composite Pattern

Problem: Deals with problems in which Objects can be grouped together, Groups can contain other groups (tree-like hierarchy).

Solution: One or more leaf classes. A composite class, representing a collection of leaf and other composite objects. An interface implemented by, or abstract superclass of, leaf and composite classes.



Lecture 9,10

1.1 Why we use SOLID?

SOLID help us to arrange code sensibly as interrelated classes, in such a way that the structure Tolerates change; Is relatively easy to understand; Can form the basis of reusable components.

1.2 What is SOLID?

Single Responsibility Principle (SRP)

Definition: A module should be responsible to one (and only one) actor. A module should have only one (unified) source of changes.

Actor: one or more stakeholders or users who specify what the module must do.

Why SRP: SRP can help programers isolate and organize modules by ensuring each module is responsible for a single, well-defined functionality. This improves code clarity, maintainability, and reduce the risk of unintended side effects when changes are made.

Dependency Inversion Principle (DIP)

Definition: To maximise flexibility, source code dependencies should reference abstractions rather than concrete classes. The most flexible systems are those in which source code dependencies refer to abstractions. In programs written in procedural languages, source code dependencies follow the flow of control. With object-oriented languages and polymorphism, we can reverse this; i.e., we have absolute control over the direction of source code dependencies. Using abstract classes & interfaces, we can rearrange source code dependencies to point in the opposite direction to flow of control.

Implications: (1). In a business application, we can rearrange source code dependencies so that database & user interface depend on the business rules, rather than the reverse DB and UI therefore become 'plugins' that can be easily replaced, without changes to business rules.

(2). Independently deployable: when source code in a component changes, only that component needs to be redeployed. This also implies independent developability.

Why DIP: DIP can help programers decouple high-level modules from low-level , volatile

components by relying on abstractions such as abstract classes and interfaces. This enhances flexibility, enables independent development and deployment, and make the system more resilient to changes.

Open-Closed Principle (OCP)

Definition: A software artifact should be open for extension but closed for modification. In other words: it should be possible to extend our design to incorporate new behaviour without having to modify existing code.

How to satisfy OCP: We separate system functionality based on how, why and when it changes, then we organise that separated functionality into a hierarchy of components, in such a way that the higher-level components are protected from changes in the lower-level ones. SRP and DIP are the guiding principles here; using them properly gives us a system that satisfies OCP.

Why OCP: OCP help programmer improve system flexibility, development efficiency, and reduce the risk of unintended modification while extending system functionality.

Liskov Substitution Principle (LSP)

Definition: Subclasses should be substitutable for their superclasses.

Why LSP: The Liskov Substitution Principle (LSP) enhances system flexibility by enabling using subclasses to replace superclasses without affecting the correctness of the program. It is also a fundamental principle for implementing the Dependency Inversion Principle (DIP), as it ensures that abstractions and their implementations remain interchangeable.

Interface Segregation Principle (ISP)

Definition: No code should be forced to depend on methods, functions or objects it does not use. Having many client-specific interfaces is better than having one large general-purpose interface.

Why ISP: Using ISP can reduce the risk of forgetting to modify indirectly dependent modules, as it encourages creating specific interfaces conform to client needs, minimizing unnecessary dependencies.

Lecture 11

CCP (Common Closure Principle)

Definition: Gather into components those classes that change for the same reasons and at the same times. Separate into different components those classes that change at different times and for different reasons. Like SRP, but for components! Makes it more likely that code changes will be confined to a single component, simplifying maintenance. Common Closure' because we group classes that are closed to the same type of changes

Why CCP: CCP can simplify code maintenance and clear the side line between different classes CCP helps simplify maintenance by grouping related classes that are likely to change for the same

reason and at the same time, minimizing the impact of changes on other parts of the system and making the boundaries between different responsibilities clearer.

CRP (Common Reuse Principle)

Definition: Don't force users of a component to depend on things that they don't need Like ISP, but for components! Classes that tend to be reused together ideally belong in the same component. If we depend on another component, we want to depend on every class in that component.

Why CRP: CRP helps improve system flexibility and simplify maintenance by ensuring only the necessary classes depend on a module, reducing the risk of reducing the risk of unnecessary changes when unused parts of the module are modified.

Coupling: What are the 'right' relationships between components? How do we avoid problems with dependencies?

Acyclic Dependencies Principle (ADP)

Definition: Allow no cycles in the component dependency graph. Components should ideally be units of work assigned to an individual developer or small team. Team should be able to release a version of the component for others to use, while they work on next version. Other teams should be able to decide when they use a particular release. Cycles of dependency make this impossible!

Question Type: Is the system in Figure 3 compliant with the Acyclic Dependencies Principle?

Justify your answer.

The system in Figure 3 complies with the acyclic dependency principle. To prove this statement, I regard each component as a node and the dependencies as directed edges to draw a dependency graph of the system. Then, I start from each node and try to see if I can get back to the original node along the dependencies. I found that no node can get back to the original node. Thus the dependency graph of this system is a directed acyclic graph, the system is compliant with the acyclic dependency principle.

Stable Dependencies Principle (SDP)

Depend in the direction of stability. Suppose we have a volatile component Y, and another component X that is difficult to change. X should not depend on Y. Otherwise, Y will also be difficult to change. SDP says that the Instability value of a component should always be larger than the Instability values of the components that it depends on.

Stable Metrics:

Fan-in: number of incoming dependencies – i.e., number of classes outside this component that depend on classes within this component (Fin).

Fan-out: number of outgoing dependencies – i.e., number of classes inside this component that

depend on classes outside this component (Fout)

instability: $I = F(\text{out}) / (F(\text{in}) + F(\text{out}))$, $I = 0$: maximum stability, $I = 1$: maximum instability.

Stable Abstractions Principle (SAP)

Definition: A component should be as abstract as it is stable. High-level policies ('business rules') should be at the centre, where dependencies are pointing to. SDP tells us that such components should ideally have maximum stability ($I = 0$).

But this creates inflexibility: what if we need to change the implementation of such a component?

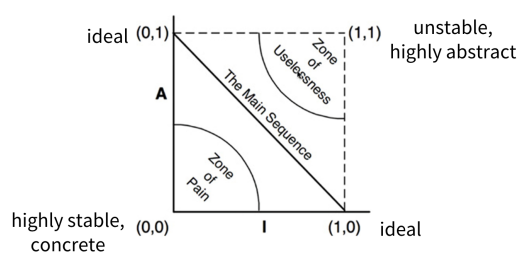
OCP tells us that we should write code that is open for extension, closed for modification.

Abstract classes / interfaces help us achieve this. If a component is sufficiently abstract, then it can be extended without modification – so it can still be stable. Flip side of this is that it is OK for unstable components to be concrete.

Abstractness: The ratio of abstract classes and interfaces in a component to the total number of classes in that component. $A = N(\text{abs}) / N(\text{total})$. The instability of a component is supposed to be inversely proportional to its abstractness.

Instability/Abstractness Graph

UNIVERSITY OF LEEDS



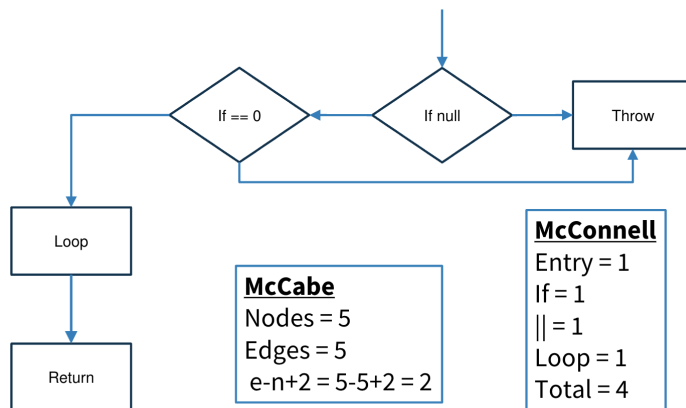
Components should be as close as possible to the 'Main Sequence', an inverse relationship.

Lecture12

Cyclomatic Complexity(McCabe Complexity): Treats paths through a method as a graph.

Statements are the nodes. Two nodes are linked by an edge if control may pass from one node to the other. If e is the number of edges and n the number of nodes, then cyclomatic complexity C is given by $C = e - n + 2$.

McConnell Complexity: Start with 1, for straight path through a method. Add 1 for each of these keywords or their equivalents: if, while, repeat, for, and, or. Add 1 for each case in a switch statement. All exits combined count as one.



Lecture13

1.what software architecture is: Architecture = 'shape' given to a system by the architect

and the other developers (partial definition!) Form of that shape comes from: The division of the system into components; The way these components are arranged; The ways that these components communicate. The purpose of that shape is to facilitate development, deployment, operation and maintenance.

2.why we need software architecture:

All systems can be decomposed into Policy and Details. Policy: high-level business rules/procedures, where the true value of the system resides. Details: things necessary to enable humans / other systems to interact with policy – I/O devices, databases, servers, communication protocols, etc. Architect's goal is to create a shape that recognises policy as the most essential element and makes details irrelevant to policy – allowing decisions about details to be deferred until later.

3. what can go wrong when software architecture is ignored: If core business logic enters outside world interfaces: (1). High coupling of the core and interface components make automated unit testing difficult.(2). Adding new external drivers or changing existing ones can become very difficult.(3). The high volatility of some outside drivers will leak back into the core.

4. Hexagonal Architecture: also known as 'Ports and Adapters' architecture. Domain entities and use cases occupy a central core, with no outgoing dependencies. Adapters exist outside the core and interact with it through well-defined ports. Adapters drive the application with inputs, or are driven by the application via outputs fed to them.

How to make Ports & Adaptors: (1).Work out the 'conversations' that can occur between drivers and the core.(2). For each conversation define a port, and based on the needs of the conversation define an Application Program Interface (API) for the port. (3). For each external driver write an adaptor.

Subscription weather system has four natural ports: 1. weather data feed, 2. administrator, 3. notified subscribers, 4. subscriber database.

5. Clean Architecture:

Entities Layer: Contains classes to represent the key domain objects. Encapsulates critical business rules. Methods in a class relate to the entity in general, not to specific business applications. No frameworks!

Use Cases Layer: Represents the use cases of the application. Encapsulates application-specific business rules. Use cases orchestrate the flow of data to and from the entities and direct them to use critical business rules to achieve the goals of each use case. A use case doesn't know what triggered it or how the results of computation are being presented. Pure business logic (possibility using utility libraries). No frameworks!

Interface Adapters Layer: Where use cases are triggered. Transfers data to/from different sources (databases, filesystem, network, etc). Converts between formats that suit use cases / entities and formats that suit those sources. Defines the interfaces needed by the data providers. Implements the interfaces defined in the Use Cases layer.

External Interface Layer

Advantages:

- (1). Independent of Frameworks. The architecture does not depend on the existence of some library of feature laden software. This allows you to use such frameworks as tools, rather than having to cram your system into their limited constraints.
- (2). Testable. The business rules can be tested without the UI, Database, Web Server, or any other external element.
- (3). Independent of UI. The UI can change easily, without changing the rest of the system. A Web UI could be replaced with a console UI, for example, without changing the business rules.
- (4). Independent of Database. You can swap out Oracle or SQL Server, for Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database.
- (5). Independent of any external agency. In fact, your business rules simply don't know anything at all about the outside world.

Lecture14

Quality characteristics: Functional Suitability, Performance efficiency, Compatibility, Usability, Reliability, Security, Maintainability, Portability.

These characteristics may conflict, architect must determine the best trade-off. How to do it?

Architecture Trade-off Analysis Method (ATAM): Present business drivers and primary quality characteristics to assembled stakeholders ... followed by proposed architecture, focusing on how it addresses business drivers and QCs.

Architect then defends proposed architecture in a series of scenarios presented by stakeholders: Use Case scenarios, Growth scenarios, Exploratory scenarios.

Lecture 15,16,17,18

Architecture Styles-Monolithic

Monolithic: One or more deployment units, but a single architecture quantum (AQ). Definition of AQ: the grouping of components that must be deployed together in order for the system to function correctly. Use of a single database that all components depend on often means that a system has a single AQ.

Layered Style: Partitioning done by technical category. Layers are closed: requests must come in at the top, through the presentation layer, and be passed down from layer to layer. Some requests may not need to engage with particular layers – so temptation is to make those layers 'open', allowing them to be bypassed.

When to use Layered Style: Simple applications, tight time/money constraints. Team organization mirrors technical partitions. Change is occasional and confined to specific areas (e.g., to business rules in one layer)

When not to use Layered Style: Scalability, elasticity, fault tolerance, extensibility are important. Change is frequent and domain-oriented (hence changes are spread across layers). Changes and deployments have to be fast.

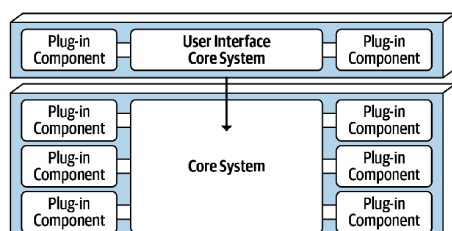
Modular Monolith: Partitioned by domain rather than by technical concerns.

When to use Modular Monolith style: Simple applications, tight time/money constraints. Teams are agile or organized by functional area. Change is occasional and confined to specific functional areas.(e.g., 'add a new payment system')

When not to use Modular Monolith style: Scalability, elasticity, fault tolerance, extensibility are important. Changes cut across multiple domains. Changes and deployments have to be fast.

Microkernel Style: Core system contains minimal functionality needed to run system, general business rules, etc. Plug-in modules are standalone, independent of each other and contain business rules for specific functions. Plug-ins contain the volatile parts of the application. Potential for dynamic configuration – where system detects available plug-ins when it runs, allows reconfiguration during execution, etc.

Partitioning: If the core is complex enough, it can be given a layered architecture... (technical partitioning), or the core can be a modular monolith (domain partitioning).



When to use Microkernel Style: You have a core of functionality, with optional extras. Core is fairly stable, change is concentrated at the periphery. You have varying configurations for each deployment / usage of the application.

When not to use Microkernel Style: Scalability, elasticity, fault tolerance are important. Tightly-coupled processing, not amenable to plug-in approach. Frequent changes to core are needed.

Architecture Styles-Microservices and Service-Based

Microservices: Inspired by concepts of domain-driven design, and specifically the idea of bounded context. Highly distributed architecture, comprising separate, independently deployable services that work together, each modelled around a business domain. Changes often localised in a single service, reducing impact on system, allowing a faster release cadence.

Pros and Cons: Supports rapid change. Highly scalable and elastic. Complex and expensive to build. Poorer performance than many other styles.

‘Bounded Context’ Concept: Each service models a domain or workflow, and contains everything needed for that workflow (code + data). Within a BC, code and DB schemas are coupled together to produce work, but they are not coupled to anything outside the BC. Implies data duplication, and code duplication! Code reuse is generally considered beneficial, but here it would couple services and prevent independent development / deployment!

Microservice Principles:

- (1).Single responsibility.
 - (2).Encapsulation and interface segregation: each microservice owns its data, communication through well-defined interfaces.
 - (3).Autonomy and ownership: the team that builds the microservice owns it and is responsible for its operation.
 - (4).Change-aware design: failures happen, requirements change, workload patterns change.
 - (5).Observability: key component to support efficient operation, debugging, troubleshooting.
- Additional: Common operational concerns are handled by adding a standard ‘sidecar component’ alongside each service

API Layer: Hides service endpoints, exposing only those that should be available for public consumption. Also acts as an endpoint proxy – allowing services to be renamed & reconfigured without breaking clients. Can do other things, such as load balancing. Multiple API layers are possible – e.g., one for mobile apps, one for web browsers...

Microservice Challenges: Latency (affecting performance), Data consistency, Partitioning, Choreography / orchestration.

Service-Based Style: Looks quite similar to microservices style. Fewer, more coarse-grained

service components, sharing database context. Benefits: reduced latency, hence better performance; improved robustness. Drawbacks: harder to develop, test, deploy.

SBA as a 'Stepping Stone': Big & risky leap from monolithic to microservices.

Service-based is a useful intermediate step: Find large-scale seams and split along them – e.g., separating out admin from customer-facing. Customer-facing part can then be split into an API layer and multiple microservices, but still a shared DB. Finally, can give microservices their own separate DBs.

Architectural Styles – Pipeline and Event-Based

Pipeline Style: a.k.a. 'Pipes and Filters'. Very modular, can be implemented as a monolith (e.g. Unix pipelines) or distributed (e.g. MapReduce). Filters: self-contained and independent, designed to perform single specific task. Pipes: unidirectional communication channels that connect one filter to another ○ Payload can be any kind of data.

Applications: Unix shell, Electronic Data Interchange for businesses, ETL (Extract, Transform & Load) tools, MapReduce programming model, Big Data, etc...

Filter Types: (1)Producer: starting point, with an outbound pipe only. (2)Transformer: has an inbound and an outbound pipe, and does some data processing. (3)Tester: has an inbound pipe and does conditional processing, either discarding or passing a message on. (4)Consumer: endpoint, inbound pipe only.

Pipeline Style Characteristics:

Can be easy to change: adding filters and pipes does not impact the rest of the codebase. limited impact of changes on a specific filter.

Pros and cons regarding performance: scalability can be achieved by adding more parallel filters. still, large amount of state needs to be transferred across filters.

Fault tolerance: single-point-of-failure vulnerabilities.

The Event-Driven Style: A distributed architecture consisting of highly-decoupled, single-purpose event processing components. Events are received & processed asynchronously. Key benefits: high scalability and performance. **Two topologies: broker and mediator.**

The Event-Driven Style-Broker Topology:

Two elements: (1).Event channels, which act as message queues, to which events are posted (2).Event processors, that listen to and post to channels, and do work.

Channels are provided by lightweight general-purpose message broker (e.g., Apache ActiveMQ) Suitable for applications needing a relatively simple event processing flow and no coordination.

The Event-Driven Style-Broker Topology Trade-offs(pros and cons):

Advantages:Highly-decoupled event processors. High scalability. High responsiveness. High performance. High fault tolerance.

Disadvantages: Workflow control. Error handling. Recoverability. Restart capabilities. Data inconsistency.

The Event-Driven Style-Mediator Topology: Still have event channels and an initiating event. and we add a central Event Mediator, posting events to other channels, to which Event Processors subscribe. All messaging goes through this Event Mediator, which is where key business rules are defined. Better option than Broker if we need coordination or more sophisticated error handling. BUT adding a new process requires changes to the Event Mediator!

Popular open source mediator implementations exist, e.g., Spring Integration or Apache Camel.

Various ways of specifying event flows: Java code, Domain-specific language (DSL), Business Process Execution Language (BPEL)

The Event-Driven Style-Mediator Topology Trade-offs(pros and cons):

Advantages:. Workflow control. Error handling. Recoverability. Restart capabilities.

More consistent data.

Disadvantages. More coupling of event processors. Lower scalability. Lower performance.

Lower fault tolerance. Modelling complex workflows.

Challenges of Event-Driven:

Implementation complexity: Remote process availability. How to handle lack of responsiveness.

How handle reconnection if a component fails

Lack of atomic transactions: Highly decoupled & distributed nature makes it hard to maintain transactional 'unit of work'. Shouldn't be using separate processors for tasks that belong in a single transaction.

Pipeline vs Event-Driven:

Pipeline	Event-Driven
Synchronous data filtering	Asynchronous event processing
Unidirectional	Request-reply
Filters are usually very small	Event Processors typically carry out more complex tasks
Monolithic or distributed	Highly distributed

Architectural Styles – Space-Based and Serverless

Space-Based Architecture: Primary goal is to eliminate the database constraints that limit scalability – by keeping data in memory.Processing units contain the application.PUs supported by virtualized middleware containing: Messaging grid, Data grid, Processing grid, Deployment manager.

Virtualized Middleware: does housekeeping and handles communications.

Messaging grid manages input requests and session info, forwards a request to an available PU.
Data grid interacts with replication engine in each PU to ensure that every PU contains the same data.

Processing grid is optional, needed for orchestration if the PUs are not all identical and have different roles.

Deployment manager spin up new PUs when needed, shuts them down when no longer needed.

Data Collisions: $C = N(\text{pu}) \cdot U^2 \cdot L / N(\text{rows})$

C: collision rate, N(pu): number of processing units, N: rows number of rows in PU's data grid,

U: rate of updates, L: replication latency.

Suppose we have 5 PUs and 50,000 data grid rows (because there are 50,000 products being sold). Assume that there are 20 updates per second and replication latency of 100 ms.

How many updates and collisions in 8 hours?

20 per sec = $20 \times 60 \times 60$ per hour = 576,000 in 8 hours

$C = 5 \times 20^2 \times 0.1 / 50,000 = 200 / 50,000 = 0.004$ **per second**
Hence in 8 hours there will be **~115 collisions**

Serverless Architectures: System runs on someone's server – just not yours!

Two flavours: Backend as a Service (BaaS), Functions as a Service (FaaS).

Backend as a Service: Traditionally, a browser talks to a web server, which talks to an app server, which talks to a DB – all of which are set up and managed by the system developer • BaaS replaces parts of this stack with standardized services from a service provider • e.g. authentication, encryption, file storage, databases • Developer only has to worry about how to consume those services via an API • Example: Google Firebase for data storage 19.

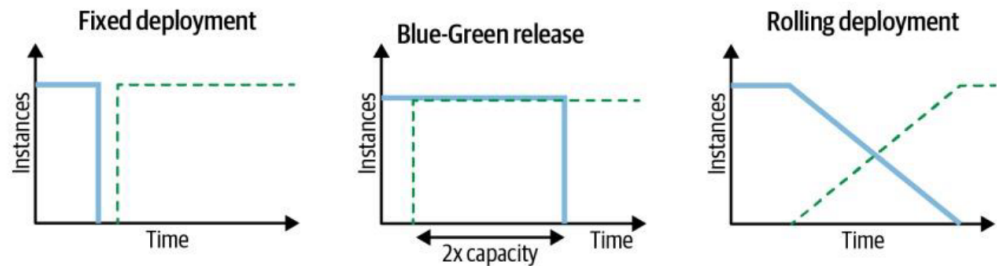
Functions as a Service: Developers write logic that is deployed in containers fully managed by a platform provider • Function is executed on demand (event-driven) • Provider takes care of machine provisioning and invocation of code, and bill you for compute time and data transfers • Developers have more control of their application, when comparing with BaaS, as they can create custom apps rather than relying on a library of prewritten services • Drawback: limited control over execution.

Key characteristics of containers hosting functions in FaaS serverless deployments: o Stateless, making data integration simpler but increasing communication overheads o Ephemeral, allowing them to be run for a very short time (all resources must be freed once execution is finished) o Event-triggered, so they can run automatically when needed (may take longer to react if not called frequently) o Fully managed by a cloud provider.

When to use Serverless Architecture: computation does not need to maintain state, or state is small enough to be easily retrieved from a temporary data storage service
 ○ short-running processes
 ○ can tolerate cold start latencies
 ○ asynchronous communication.

Lecture 19

Architectural Patterns



source: Ibryam & Huß, O'Reilly

Three deployment types, Type: deployment pattern • Impact: maintainability, reliability •

Suitable for: Microservices, Space-based, Serverless.

Deployment Stamps, Type: deployment pattern: Impact: performance efficiency, scalability •

Suitable for: Microservices, Serverless.

Sidecar Pattern: • Extends and enhances an existing architectural quantum without changing its core functionality
 ○ e.g. API adaptors, services and components supporting non-functional quality characteristics • Sidecar always deployed together with primary quantum
 ○ access to the same resources
 ○ negligible communication latency
 ○ observability • Non-intrusive, external APIs of the primary quantum remain unchanged.

Type: structural pattern: • Impact: maintainability, compatibility, performance • Suitable for:

Microservices, Service-based.

CQRS = Command-Query Responsibility Segregation: • A different way of creating 'systems of record' where there is a need to both read from and write to a DB • CQRS separates read (DB query) and update (DB update) operations for a data store when there are different engineering requirements for those operations.

Type: structural pattern • Impact: performance, maintainability • Suitable for: Event-Driven, Microservices, Service-based, Monolithic.

Valet Key: • Client requests a resource and needs to supply a large amount of data
 ○ e.g. large image or video files • Service must validate the request, but does not necessarily need to handle the data transfer
 ○ generates a temporary access token to the resource and returns it to the client
 ○ client uses the token to upload the data directly to the target resource source.

Type: behavioural pattern • Impact: performance, usability • Suitable for: Microservices,

Serverless.

Priority Queue: • Ability to provide different service levels to different users • Higher priority requests are processed more quickly ○ instead of typical FIFO behaviour.

Important design decisions ○ priority assignment ○ level of service per priority level ○ autoscaling (e.g. queue length) ○ starvation avoidance.

Type: behavioural pattern • Impact: performance, usability • Suitable for: Event-Driven, Microservices, Service-based.

Answer Template:

First, I observed that the `TransformMatrix.updateMatrix` method contains duplicate code smells, so it needed to be refactored.

To refactor it, I wrapped the two dimensional matrix multiplication logic in the method into a new function `matrixMultiplication`. The function input is two 2D matrices, and the output is the result of matrix multiplication. At the beginning of the function, the input matrices are verified and an `IllegalArgumentException` is thrown if the verification fails. After that, the matrix multiplication is performed and the result is returned.

Additionally, I wrote a unit test `MatrixMultiplicationTests` for the `matrixMultiplication` method. The unit test uses JUnit4's parameterized test and provides six sets of test data to verify the identity matrix, zero matrix, input parameter exception and other situations.

After verifying the `matrixMultiplication` method, I used it to replace the matrix multiplication logic in the `updateMatrix` method. Then, I created a variable named `result` (`double[]`) to replace temporary variables such as `tmp1` and `tmp2`, and finally assigned `result` to `TransformMatrix.matrix`.

The above operations simplify the structure of the code, making the code shorter and easier to maintain.