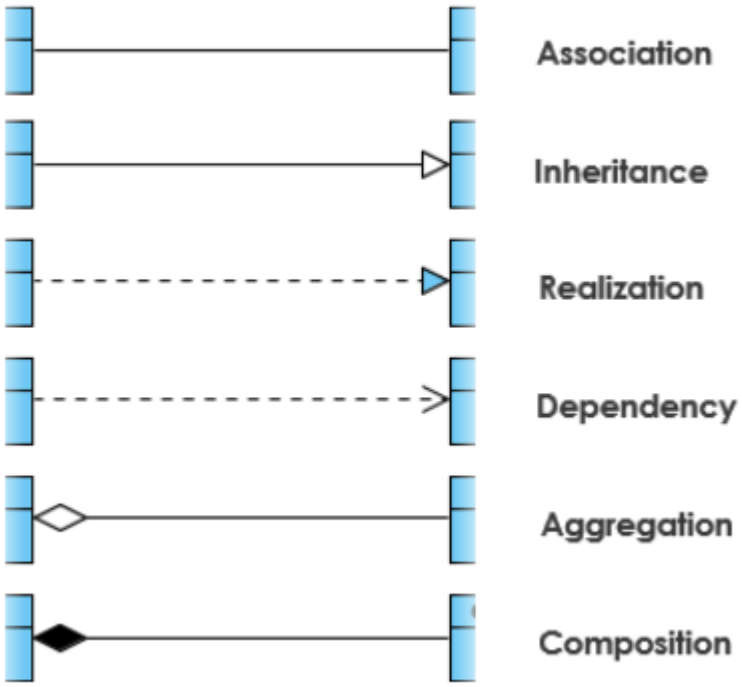
COMP2511

1. OOP in Java
    1. "is-a" --> Inheritance relationship
        1. rectangle is a shape
    2. "has-a" --> association relationship
        1. rectangle has a line
        2. it is a examples of creating new classes by composition of existing classes
    3. objects are instances of a class
    4. Polymorphism
        1. methods overriding
        2. an object's ability to decide what method to apply to itself, depending on where it is in the inheritance hierarchy, is called polymorphism
    5. Data hiding and encapsulation
        1. access modifiers:
            1. public --> visible to the world
            2. private --> visible to the class only
            3. protected --> visible to the package and all subclasses
            4. default --> visible to the package
    6. Abstract class vs Interfaces
        1. Interfaces can't store state
        2. you can only extend from one abstract class at a time whereas you can implement unlimited inferfaces.
    7. static:
        1. attibutes and methods belongs to the class, not object
    8. equal() method
        1. first check passed in object is not null
            1. `if (object == null) return false`
        2. check if the passed in object is the same instance as the calling object
            1. `if (this == object) return true`
        3. check the concrete type of calling object matches the concrete type of the passed in object
            1. `if (!this.getClass().equals(object.getClass())) return false`
        4. after confirmed that the passed in object is of the same type, cast the passed in object to the same type

5. then compare the fields inside the class

2. Domain Modelling
    1. UML class diagrams: relationships



    2. Dependency:
        1. loosest form of relationship
    3. Association:
        1. A class uses another class in some way
    4. Directed association:
    5. Aggregation:
        1. A class contains another class
        2. a course contains students
    6. Composition:
        1. the contained class is integral to the containing class.
        2. integral: the contained class cannot exist outside of the container
        3. the leg of a chair, engine in a car
    7. Composition vs aggregation:
        1. composition:
            1. the container strongly owns the part object
            2. if the container is destroyed, so is the part
        2. aggregation:
            1. the part can exist indenpendenly of the whole

3. Design By Contract
    1. responsibilities are clearly assigned
    2. prevent redundant checks --> simpler, easier maintainance
    3. contract should address:
        1. pre-condition
        2. post-condition
        3. invariant --> what does the contract maintain
            1. value of mark remains between 0 and 100
    4. contract should be declarative and must not include implementation details
    5. post-conditions in inheritance
        1. post-conditions may be strengthened in a subclass
        2. an implementation or redefinition (overriden method) may increase the benefits but not decrease it.
        3. the original contract requires returning a set
            1. the overriden method returns a sorted set, offer more benefit to a client
    6. class invariant constrains the state (values of certain variables) sorted in the object
    7. class invariants in inheritance
        1. must satisfy the invariants of all the parents, preventing invalid states.
4. Exceptions
5. Generics and Collections
    1. Generics:
        1. E - element
        2. K - key
        3. N - Number
        4. T - Type
        5. V - Value
    2. Bounded Type parameters
        1. `<U extends Number>`
    3. Wildcards
        1. in generic code, ? called the wildcard, represnets an unknown type.
        2. upper bound `<? extends Foo>`
        3. unbounded `List<?> list`
        4. lower bound `<? super A>`
    4. collections
        1. set
        2. list
        3. queue

4. deque

6. Software Design Principles
    1. SOLID principles
        1. Single responsibility principle
            1. single functionality
        2. Open/Closed principle
            1. open for extension but closed for modification
        3. Liskov Substitution Principle
            1. objects of a superclass should be replaceable with objects of subclasses without affecting the correctness of the program.
        4. Interface Segregation Principle
            1. clients should not be forced to depend on interfaces they do not use, facor many specific interfaces over a singe general purpose one.
        5. Dependency Inversion Principle
            1. Depend on abstractions, not concrete implementations. Higher level modules should not depend on lower-level modules but rather on abstractions.
    2. Cohesion and Coupling
        1. cohesion:
            1. the degree to which elements of a module/class belong together
            2. high cohesion:
                1. elements of the module work towards a single purpose
                2. example: calculateTotal(), applyDiscount(), generateInvoice()
            3. low cohesion
                1. elements are unrelated or loosely related
        2. coupling
        3. good design = high cohesion + low coupling
        4. Tips for high cohesion
            1. use single responsibility pinciple
            2. refactor when a class or method grows too large
        5. tips for low coupling
            1. minimize shared data
            2. use interface and abstractions
            3. apply dependency injection
            4. event-driven or observer patterns
    3. Principle of Least Knowledge (Law of Demeter)

1. only call methods on on objects you directly know

Code Example – Violating LoD (Tightly Coupled)

```
class Engine {
    public void start() { System.out.println("Engine started"); }
}

class Car {
    private Engine engine = new Engine();
    public Engine getEngine() { return engine; }
}

class Driver {
    public void startCar(Car car) {
        car.getEngine().start();
    }
}
```

Violates LoD,
accessing a "stranger" (engine)

Code Example – Respecting LoD (Loosely Coupled)

```
class Engine {
    public void start() { System.out.println("Engine started"); }
}

class Car {
    private Engine engine = new Engine();
    public void start() { engine.start(); }
}

class Driver {
    public void startCar(Car car) {
        car.start();
    }
}
```

Car mediates access

Talks only to its direct friend

4. Liskov substitution Principle

Examples: LSP

Violating LSP

```
class Bird {
    void fly() {
        System.out.println("Flying...");
    }
}

class Ostrich extends Bird {
    @Override
    void fly() {
        throw new UnsupportedOperationException("Ostrich can'
    }
}
```

Fixing the Violation

```
interface Bird {
    void eat();
}

interface FlyingBird extends Bird {
    void fly();
}

class Sparrow implements FlyingBird {
    public void fly() { System.out.println("Sparrow flies"); }
    public void eat() { System.out.println("Sparrow eats"); }
}

class Ostrich implements Bird {
    public void eat() { System.out.println("Ostrich eats"); }
}
```

1.

5. covariance and contravariance

1. covariance and contravariance describe how types behave in inheritance when method overriding
2. covariance: return type can be more specific
3. contravariance: parameter types can be more general

7. Refactoring
    1. common code smells
        1. duplicated code
            1. extract
        2. long method
            1. extract mehtod
        3. large class
            1. extract class
        4. long parameter list

❖ To avoid long parameter lists, encapsulate related parameters into a data class and pass an instance of that class instead.

```
void createUser(String name, int age, String email, String phone)
```

```
class UserInfo {
    String name;
    int age;
    String email;
    String phone;
}

void createUser(UserInfo user)
```

        5. divergent change

❖ A class is changed in many unrelated ways for different reasons.

❖ Violates Single Responsibility Principle.

❖ Increases risk of regression bugs due to unrelated modifications

Solution:

- Identify the reasons for change and separate them into cohesive classes.
- Use Extract Class to encapsulate each responsibility.

```
// Before
class DocumentManager {
  void print(Document doc) { ... }
  void save(Document doc) { ... }
  void exportToPDF(Document doc) { ... }
}

// After
class PrintService {
  void print(Document doc) { ... }
}

class PersistenceService {
  void save(Document doc) { ... }
}

class ExportService {
  void exportToPDF(Document doc) { ... }
}
```

        1.
        2. one class changes for many unrelated reasons
        6. shotgun surgery

- ❖ A small change requires updating many different classes.
- ❖ Makes code brittle and hard to maintain.

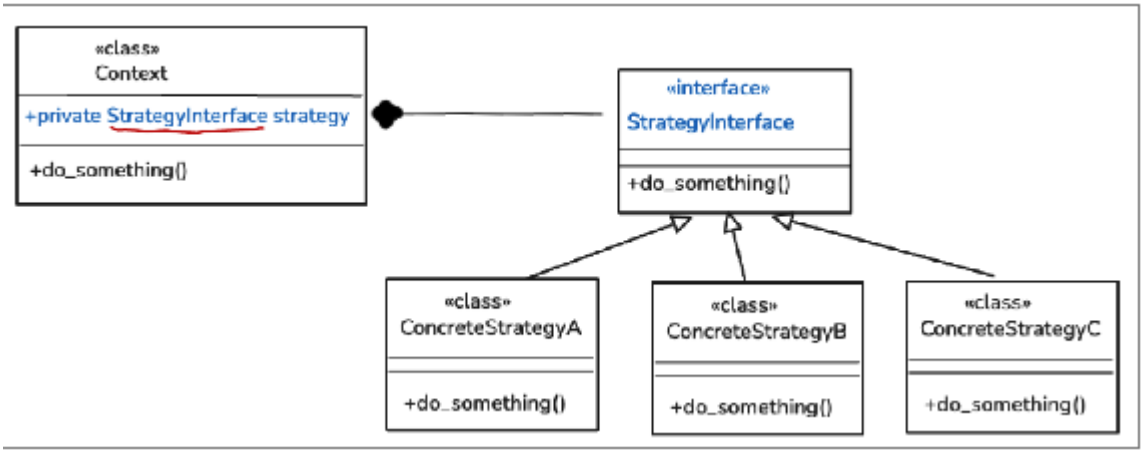
Solution:

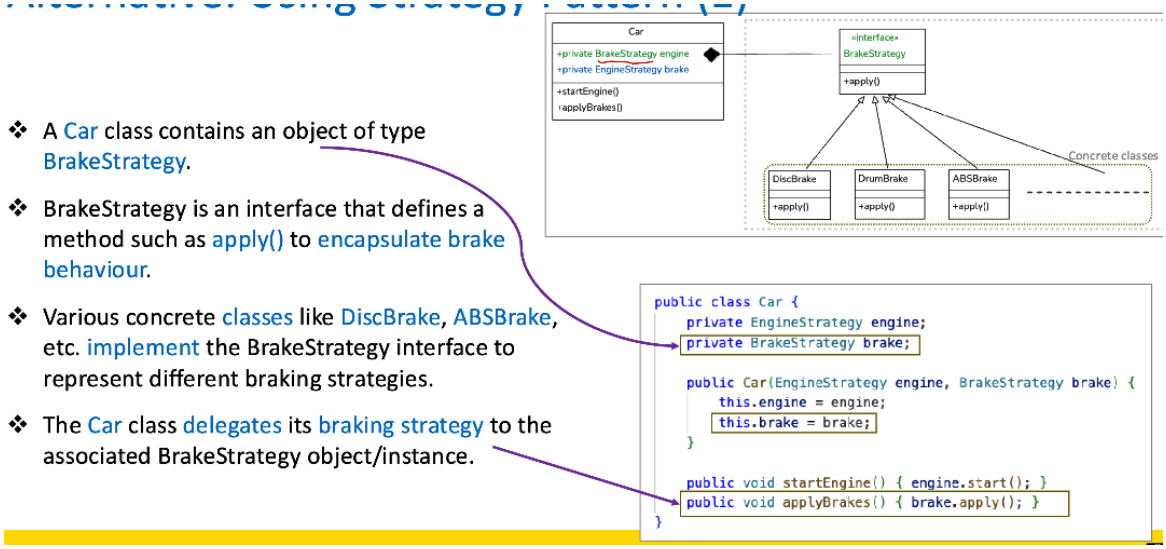
- Consolidate related changes into a single class.
- Use Move Method, Move Field, or Inline Class to localize the change.

1.
```
// Before: logic for logging exists in multiple classes
class Order {
  void logCreation() { Logger.log("Order created"); }
}
class Invoice {
  void logGeneration() { Logger.log("Invoice generated"); }
}
```
→
```
// After: Centralized logging
class LogService {
  static void logOrderCreation() { Logger.log("Order created"); }
  static void logInvoiceGeneration() { Logger.log("Invoice generated"); }
}
```

2. one change spreads across many classes

7. feature envy
   1. a method is more interested in another class's data than its own
   2. call a lot of methods from other class
   3. solution: move method to the class that owns the data
8. lazy classes
9. data classes

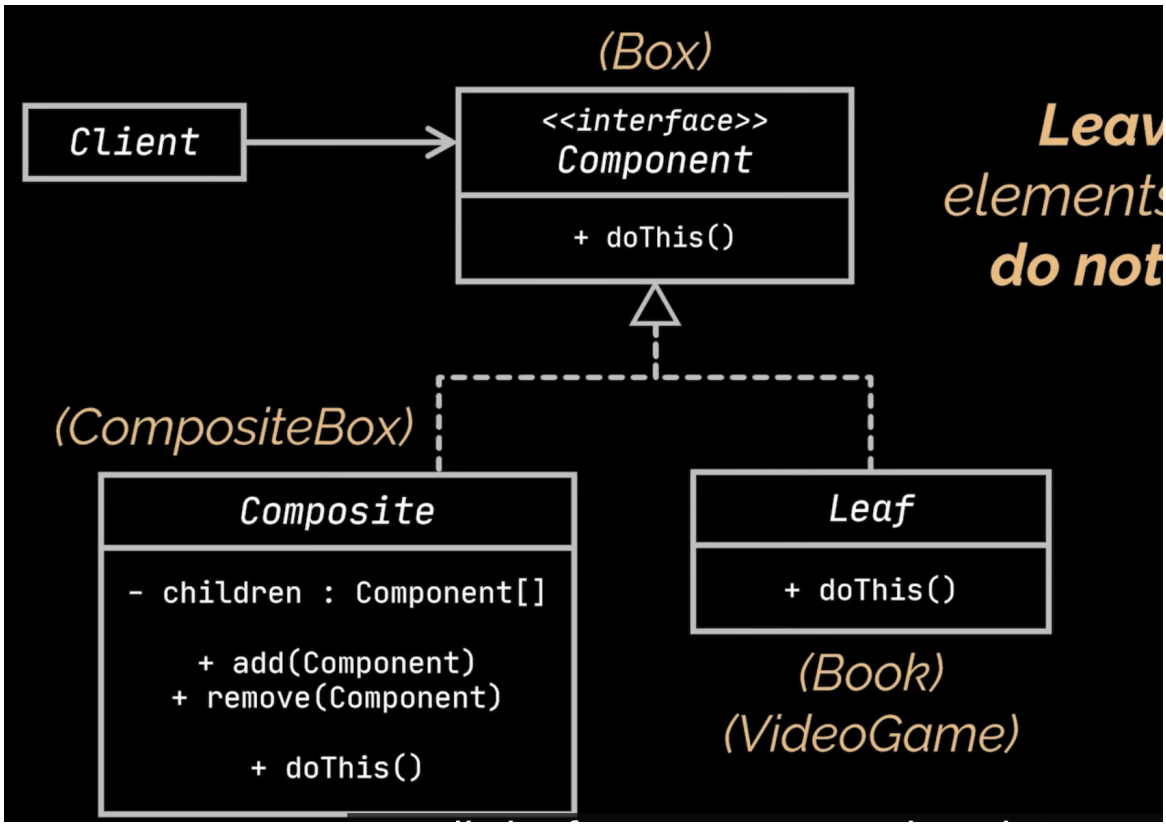
8. Strategy pattern (Behavioural patterns)
   1. problem
      1. car with different types of engines and brakes
      2. Hardcoding algorithm logic in a class makes it flexible
   2. solution:
      1. 

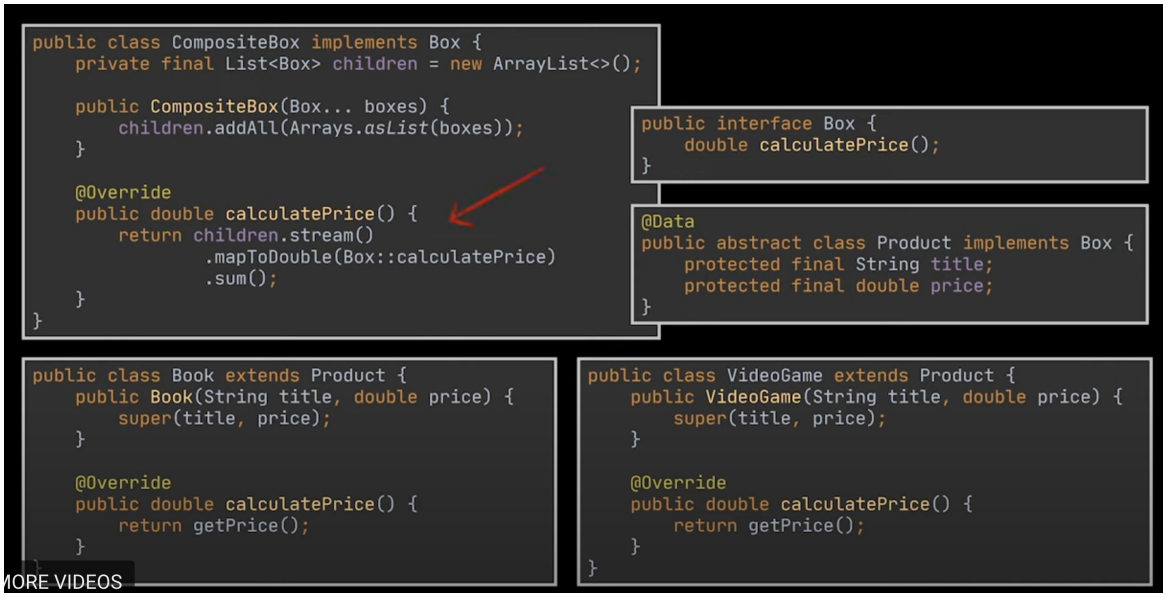


2.

9. Composite pattern (structural patterns)
    1. motivation:
        1. calculating size of a file should be same as a directory
        2. problem is tree like structure

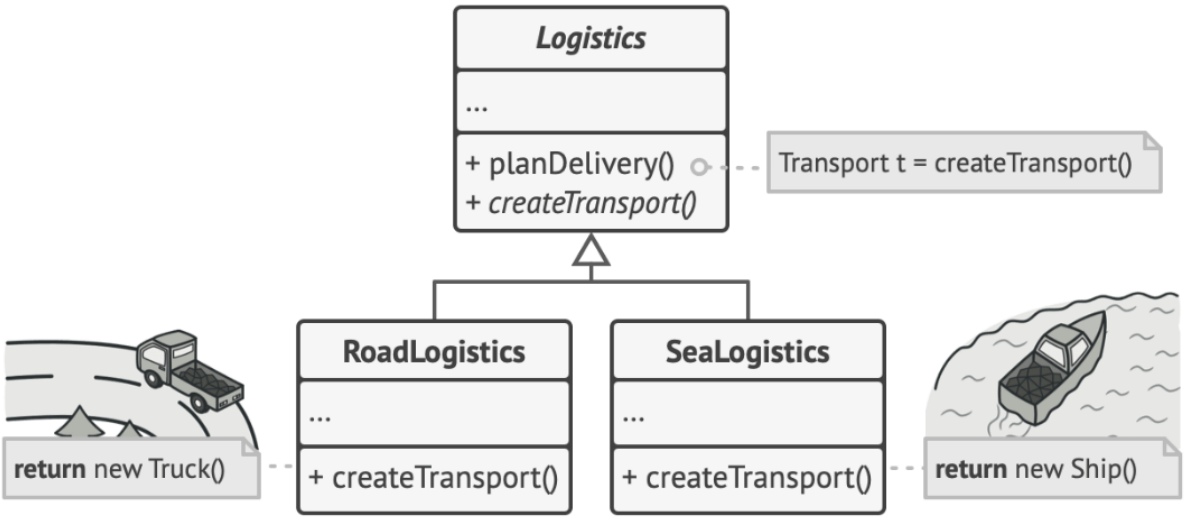
    2. soluiton:

1. 

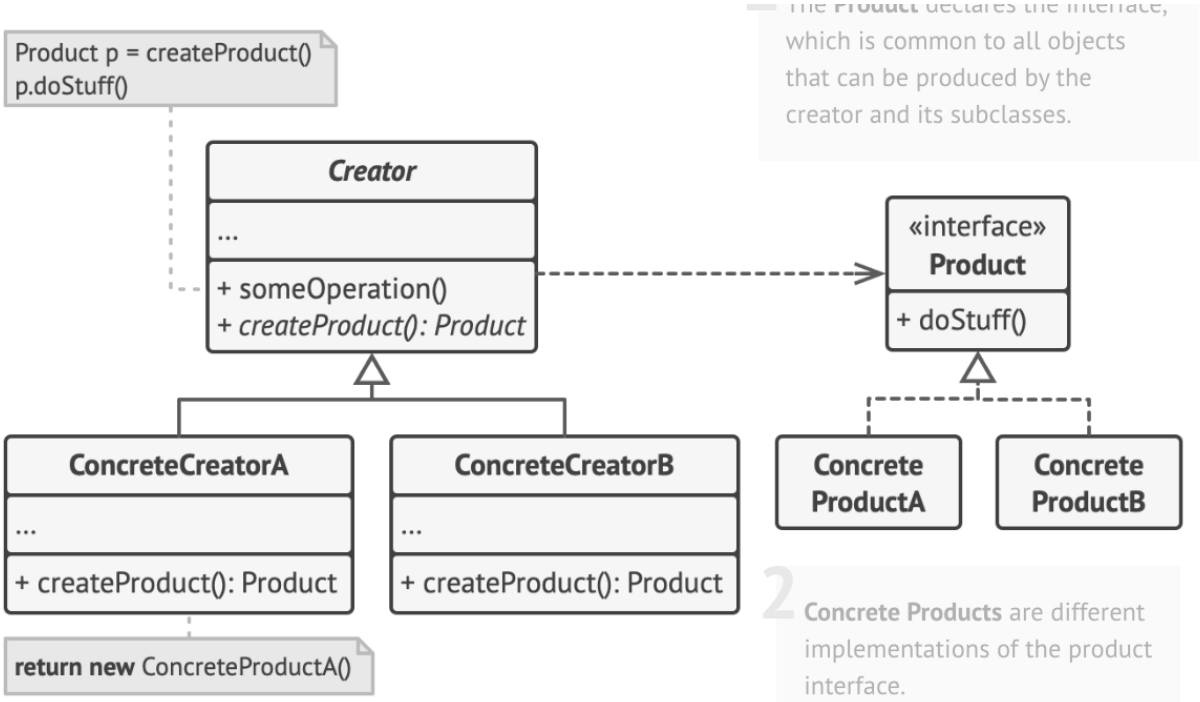
2. 

10. Factory Patterns (creational)

1. solution:



2. sturcture:

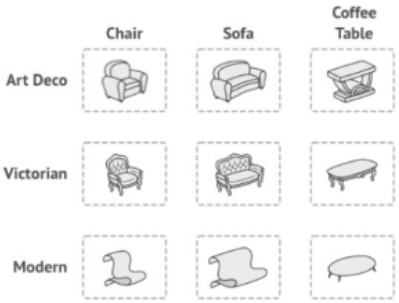


11. Abstract Factory Pattern
    1. problem: have a list of different products

Problem:

Imagine that you're creating a furniture shop simulator. Your code consists of classes that represent:

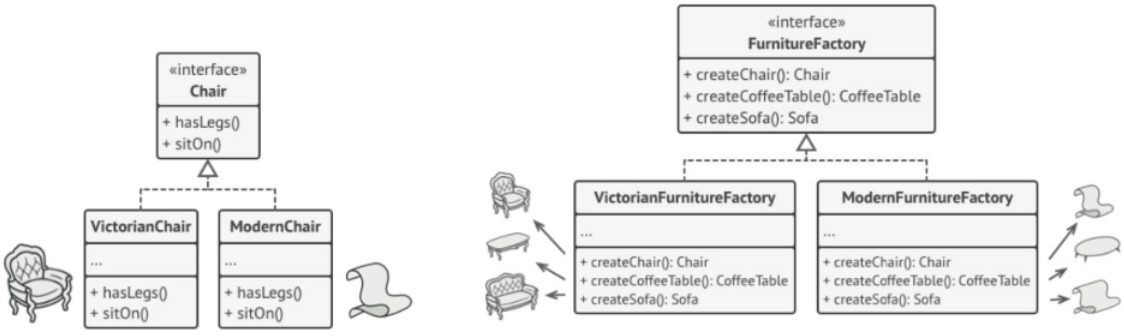
- ❖ A family of related products, say: Chair + Sofa + CoffeeTable.
- ❖ Several variants of this family.
- ❖ For example, products Chair + Sofa + CoffeeTable are available in these variants:



1. _____

2. solution:

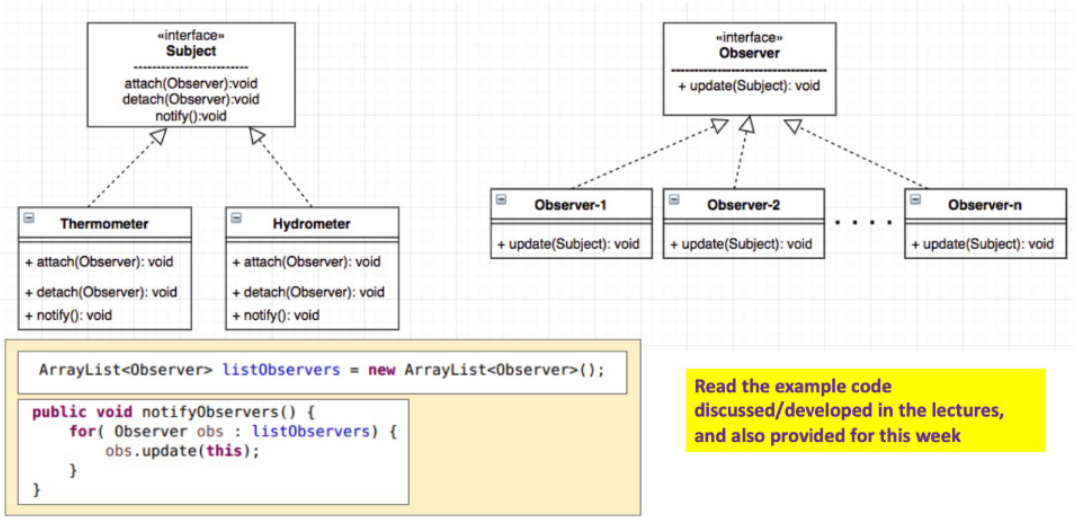
Possible Solution:

1. 

12. Observer Pattern (Behavioural Patterns)
    1. it is used to implement distributed event handling systems in event driven programming.
    2. an object called the subject, maintains a list if its dependents called observers
    3. notifies the observers automatically of any state changes in the subject.
    4. aim
        1. define a one-to-many dependency between objects without making the objects tightly coupled
        2. automatically notify/update an open-ended number of observers when subject changes state
        3. be able to dynamically add and remove observers
    5. solution
        1. define subject and observer interfaces
            1. subject:
                1. maintain a list of observers and to notify them of state chnages by calling their update()

    2. observers
        1. register / unregister on a subject
    2. loosely coupled
    3. observers can be added and removed independently at run time
    4. notification - registration == publish - subscribe

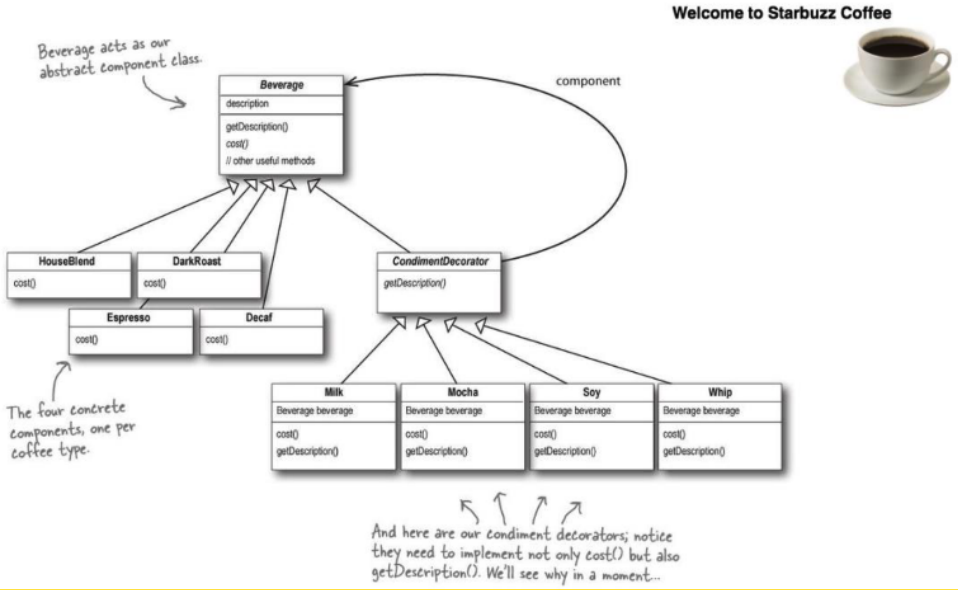
Observer Pattern: Possible Solution



6. ---

13. Decorator Pattern (structual)
    1. attach additional responsibilities to an obejct dynamically

Decorator Pattern: Example



2. ---

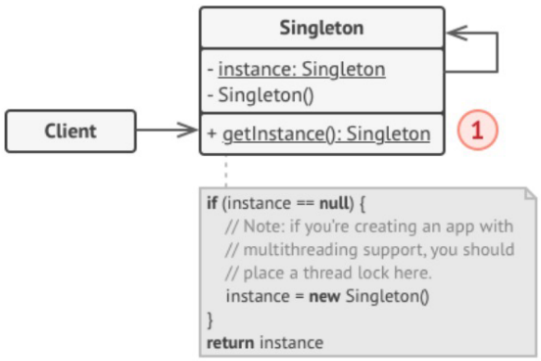
14. Functional Paradigm in Java
    1. Lambda Expressions
    2. method referenes
        1. using :: operator

        1. static method ClassName::methName
    3. pipelines and streams
        1. stream:
        2. filter operation
            1. returns a new streams that contains elements that match its predicate
        3. terminal operation
            1. e.g. for each
15. Singleton Pattern & Asynchronous Design (creational Patterns)
    1. singleton pattern
        1. a class has only one instance, while providing a global access point to this instance
        2. solution:
            1. make default constructor private
            2. create a static creation method that acts as a constructor
                1. call the private constructor and create an object and saves in a static field

        ❖ The Singleton class declares the static method *getInstance* (1) that returns the same instance of its own class.

        ❖ The Singleton's constructor should be hidden from the client code.

        ❖ Calling the *getInstance* (1) method should be the only way of getting the Singleton object.

        3.
    2. Synchronous vs Asynchronous software design

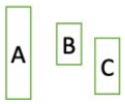
What is Synchronous programming?

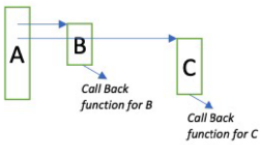
- In *synchronous* programming, operations are carried out in order.
- The execution of an operation is dependent upon the completion of the preceding operation.
- Tasks (functions) A, B, and C are executed in a sequence, often using one thread.

1. 

What is Asynchronous programming?

- In *asynchronous programming*, operations are carried out independently.
- The execution of an operation is not dependent upon the completion of the preceding operation.
- Tasks (functions) A, B, and C are executed independently, can use multiple threads/resources.

2. 



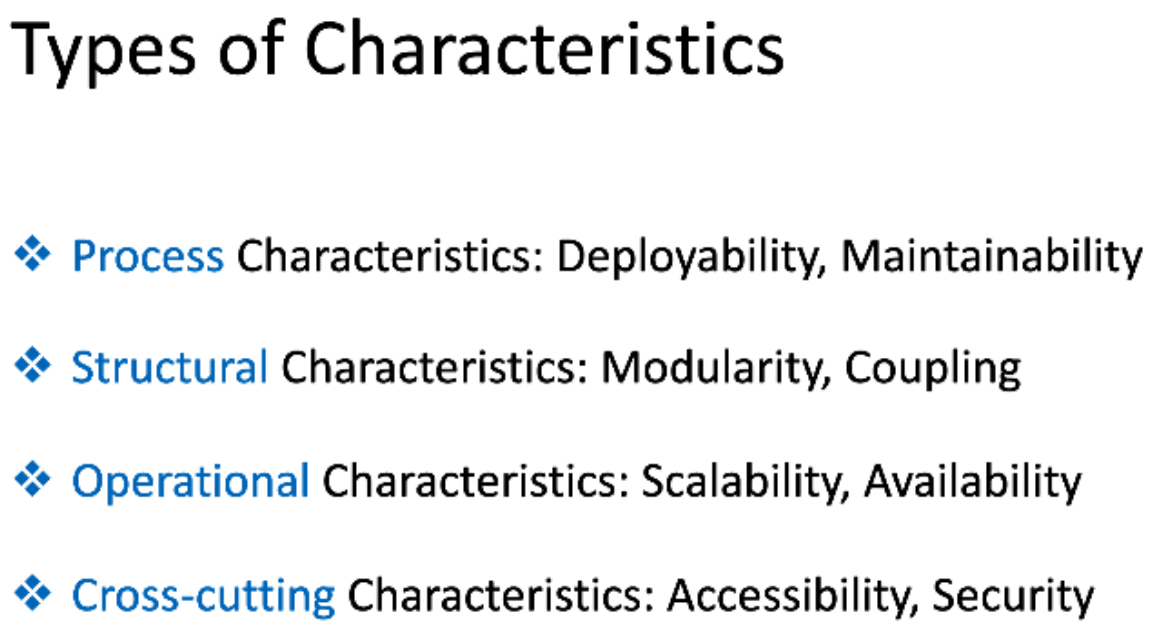
3. Kafka:
    1. an example of asynchronous software design
    2. streaming platform: enables the development of applications that can continuously and easily consume and process streams of data and events.
    3. real time, asynchronous, event-driven applications
    4. consumers have the option to either consume an event in real time or asynchronously at a later time.

16. Software Architecture
    1. Four dimentions of software architecture
        1. architectural characteristics
            1. define fundamental qualities
            2. scalability (support growth)
            3. reliability (consistent operation)

        4. availability (system uptime)
        5. testability
        6. security
    2. architectural decisions
        1. set constraints guiding future development
    3. logical components
        1. building blocks represneting business features
    4. architectural style
        1. layered (clear separation of concerns)
        2. microservices (highly scalable and agile)
        3. event-driven (responsive and scalable)
        4. examples
            1. netflix adopting microservices
            2. layered architecture -> traditional enterprise
2. architecture vs design
    1. architecture: structural decisions (hard to change)
        1. strategic decisions
    2. design: appearance and detailed decisions (easy to change)
        1. tactical decisions

17. Architectural Characteristics
    1. Scalability: handles growth in traffic or data size
    2. Availability: ensures system up time
    3. Maintainbility: Easy to update, fix or extend
    4. security: prevents unauthorized access
    5. Elasticity: automatically adjusts resources based on load
    6. deployabililty: enables safe, frequent updates
    7. responsiveness: provides quick feedback to users

8. types of characteristics:



    1. process characteristics
        1. softeare development process
        2. reflect how the system is built, tested, deployed, evoled
        3. guide decisions
    2. structural characteristics
        1. internal structure, composition of the sytem
        2. influence how components are coupled.
        3. modularity, cohesion, adaptability
    3. operational characteristics
        1. behavior at run time
        2. monitor, control, or adapt
        3. system reliability, performance, fault tolerance
    4. Cross-cutting characteristics
        1. span multiple parts of the sytem
    5. composite characteristics
        1. high-level characteristics
        2. reliability = availability + consistency = data integrity

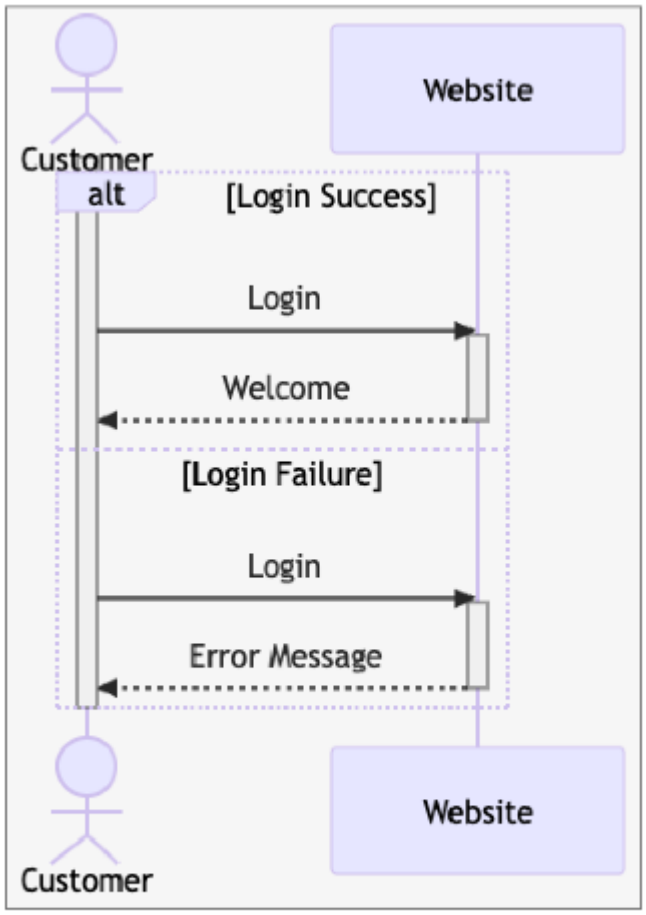
18. Architectural Decision Records
    1. why >> how
    2. structure:
        1. title
            1. identify and summarize the decision
            2. numbered

3. short and concise
4. example: ADR34: Transition to postgresql for inventory management

2. status
    1. types:
        1. proposed
        2. accepted
        3. superseded (replaced by another adr)
        4. RFC (open for feedback until a dealine)
3. context
    1. what situation led to this decision
    2. situation + alternatices under consideration
4. decision
    1. "we use ...."
    2. rationale
5. consequences
    1. pos and neg impacts
    2. known limitations
6. compliance
    1. define how decision enforcement is measured
    2. manual review
    3. automated tests
7. notes
    1. autor, approval datea, aprover

19. Behavioural Modelling
    1. sequence diagrams
        1. key components:
            1. actor
            2. objects
            3. lifelines
            4. messages
                1. synchronous
                    1. wait for server to complete
                2. asynchronous
            5. activation boxes
            6. optional intercation
                1. opt --> optional senatios
            7. conditional interaction

1. alt --> alternate senarios



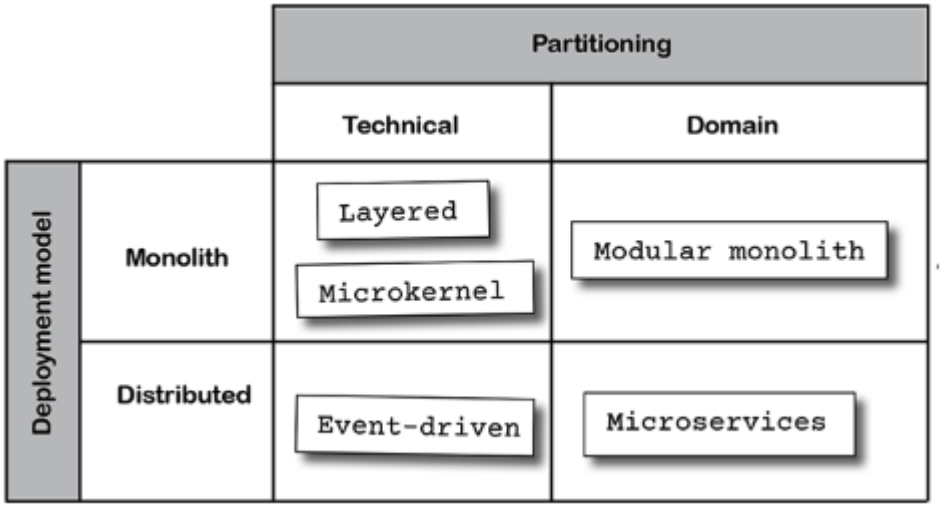
Illustrates **branching logic** based on condition results (success or failure).

8. looping interaction
    1. loop represent repeated actions
9. parallel processes
    1. par --> concurrent processes (simultaneously)

20. Logical components and modelling using C4
    1. logical components:
        1. functional building blocks of the system
        2. major features
        3. folders or modules in code base
    2. logical vs physical architecture
        1. logical: registration, payment,
        2. physical: apis, dbs, gateways, services
    3. C4 levels:
        1. system context level
            1. system + user + external systems

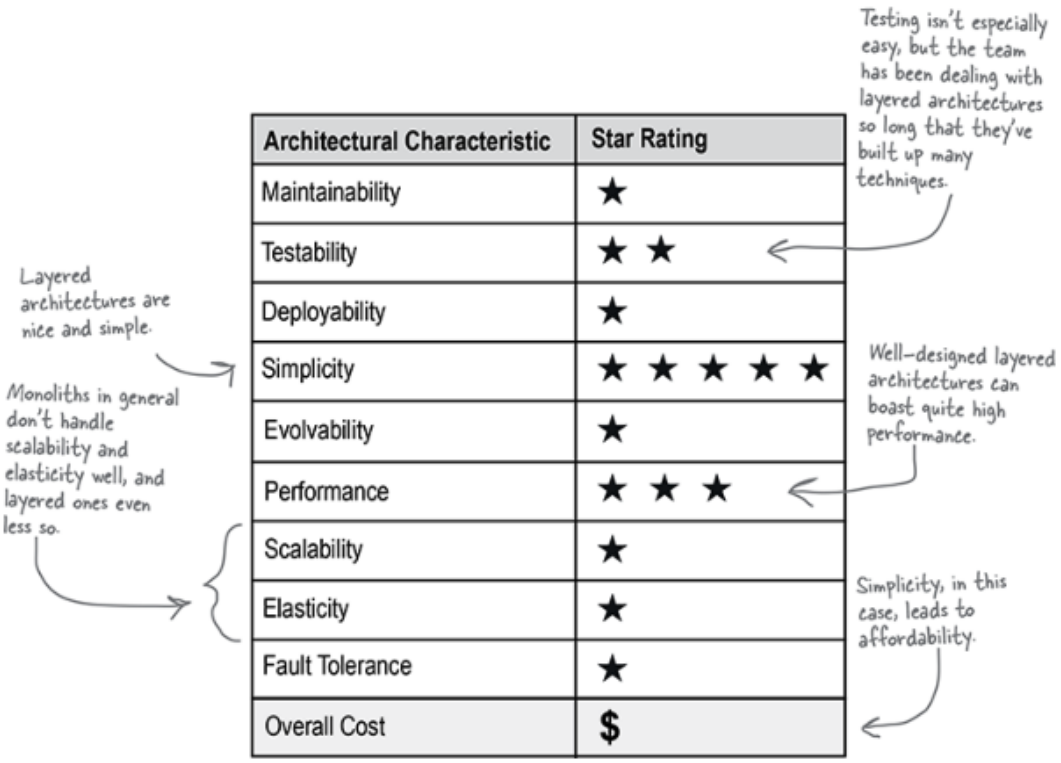
        2. for business stakeholders, non-techs
    2. containers level
        1. web apps, apis dbs
        2. for developers
    3. components level
        1. modules, services, classes
        2. developers
    4. code level
21. Architectural styles
    1. predefined patterns to guide how software sytems are structured adn deployed
    2. categorizing
        1. partitioning
            1. technical vs domain based
                1. technical:
                    1. presentation layer (ui)
                    2. business logic layer (services)
                    3. data persistence layer (databsase)
                2. domain:
                    1. customer domain (user accounts, user interface)
                    2. inventory domain (product catalog, stock management)
                    3. payment domain (billing, transactions)
        2. deployment
            1. monolithic vs distrubuted
            1. monolithic:
            1. pros
            1. easier initial development
            2. simplified debugging
            3. lower initial deployment cost
            2. cons
            1. difficult to scale
            2. single fail point
            3. inflexible when changing
            2. distributed:
            1. pros:
            1. scalability
            2. modular design
            3. fault isolation
            2. cons:

1. high complexity due to network dependence
2. increased maintatenance and debugging complexity
3. higher infrastrucutre and operational costs



22. Layered Architecuture
    1. separated technical responsibilities into distinct layers
    2. simplifies the design
    3. easy to understand and implement
    4. reuse and separation of concerns
    5. why choose?
        1. specialization
        2. physical sparation
        3. ease of reuse
        4. familiarity: easy for developers to grasp
    6. strengths
        1. feasibility: quick, cost effective
        2. technical partitioning
        3. data-intensive operations - local data processing
        4. performance
        5. fast developement -> ideal for small systems
    7. weaknesses
        1. deployability
        2. coupling
        3. scalability
        4. elaticity

5. testability

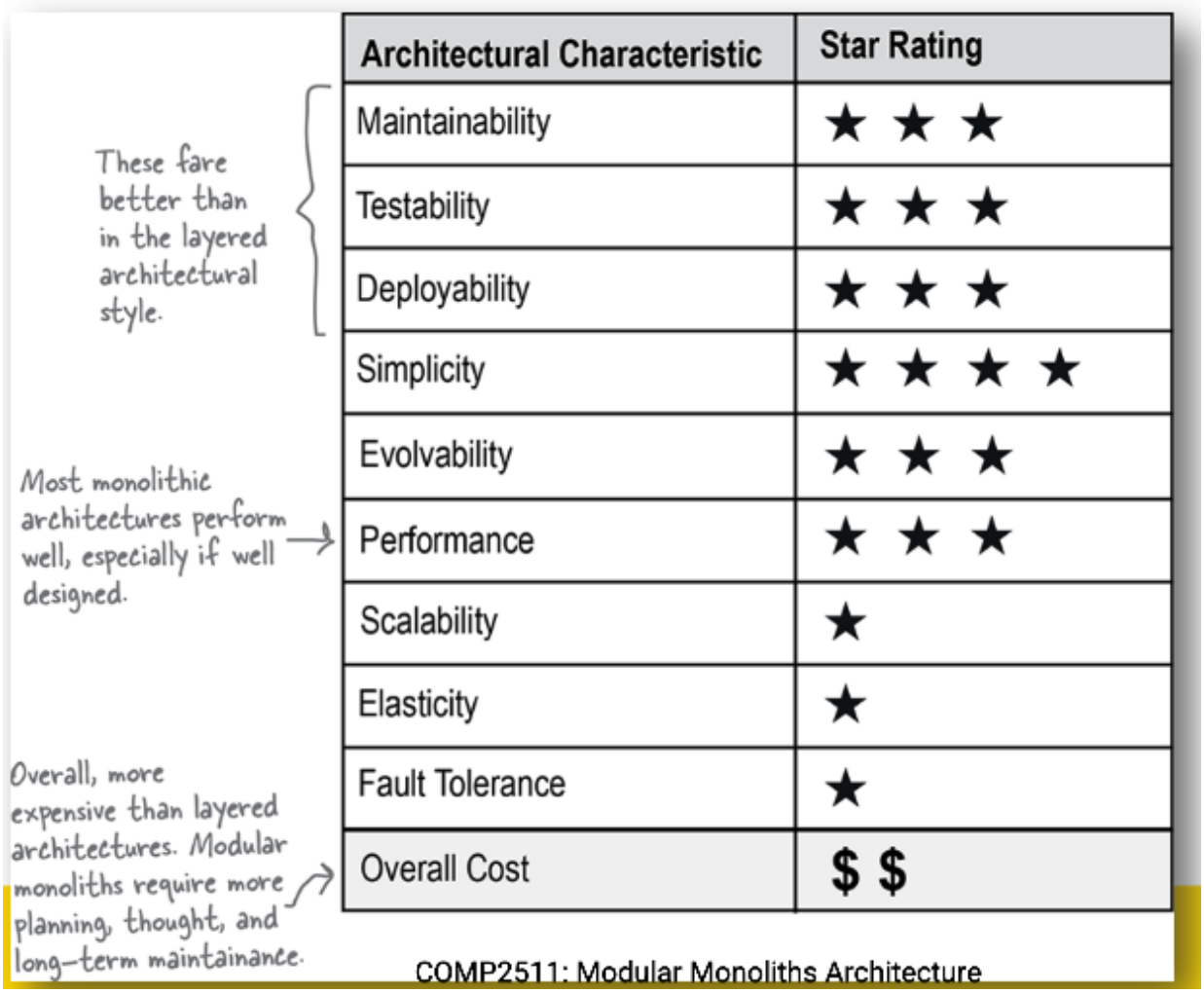


Architectural Characteristic	Star Rating
Maintainability	★
Testability	★ ★
Deployability	★
Simplicity	★ ★ ★ ★ ★
Evolvability	★
Performance	★ ★ ★
Scalability	★
Elasticity	★
Fault Tolerance	★
Overall Cost	$

8. use cases:
    1. small, simple, quick
    2. data-intensive apps with local db storage (crm)
    3. separate ui, backend, db teams

23. Modular monoliths architecture
    1. deployed as single unit, domain based
    2. domain (order, jpayment, inventory)
    3. module:
        1. independent unit within a domain
        2. contains all business logic for its domain
    4. why choose?
        1. business alignmet
        2. team ownership
        3. faster changes
        4. high performance
        5. easier testing
    5. benefits:
        1. domain partitioning
        2. performance
        3. maintainability
        4. testability

    5. deployability
6. limitations
    1. hard to share utilities
    2. no per-module customization
    3. fragile modularity (easy to break boundaries)
    4. operational limits
7. use cases:
    1. teams aligned to business domain
    2. apps must remain performant
    3. easy to test and deploy



24. Microservice architecture
    1. example:
        1. netflix
        2. amazon's product cataloue
    2. performs one specific function well
    3. own data
    4. advanatages

1. maintainbility
2. testability
3. deployability
4. evolvability
5. excetional scalability and fault tolerance
6. (continuous deployment at spotify, scalable services at netflix)

5. limitaitons
    1. compexity
    2. perfomace issues (interservice communications)
6. use case:
    1. business agility
    2. complexity handling
    3. team structure

Architectural Characteristic	Star Rating
Maintainability	★ ★ ★ ★ ★
Testability	★ ★ ★ ★ ★
Deployability	★ ★ ★ ★ ★
Simplicity	★
Evolvability	★ ★ ★ ★ ★
Performance	★ ★
Scalability	★ ★ ★ ★ ★
Elasticity	★ ★ ★ ★
Fault Tolerance	★ ★ ★ ★ ★
Overall Cost	$ $ $ $ $

25. Event Driven Architecture
    1. EDA, respond to events
    2. event:
        1. example
            1. user registered event might include the user id, name and email
    3. event vs messages
        1. evnets: boradcast something happend, with no expecttioin of response
            1. item added to cart
        2. messages: more targeted, often demanding action

        1. process payment
4. initiating events vs drived
    1. initiating events:
        1. triggered by users or external systems
        2. order placed
    2. drived events:
        1. consequences of those events and triggered by services
        2. payment arthorized
5. sync vs async
    1. sync: sender waits for a response
        1. microservices
    2. async: continues immediatly after sending
        1. loose coupling
        2. event-driven
6. advantages
    1. maintainbility
    2. performance
    3. scalability
7. challenges
    1. challenges with observability
    2. testing
8. key concepts:
    1. events
    2. asynchronous -> fire and forget
    3. combined with microservices for mordern architectures

Architectural Characteristic	Star Rating
Maintainability	★ ★ ★ ★
Testability	★ ★
Deployability	★ ★ ★
Simplicity	★
Evolvability	★ ★ ★ ★ ★
Performance	★ ★ ★ ★ ★
Scalability	★ ★ ★ ★ ★
Elasticity	★ ★ ★ ★
Fault Tolerance	★ ★ ★ ★ ★
Overall Cost	$ $ $

9.

26. serverless architecture
    1. Function as a service
    2. platforms
        1. AWS Lambda
        2. Azure functions
        3. google clound functions
        4. ibm cloud functions
    3. how it works
        1. user sends request
        2. api gateway receives and triggers a lambda/function
        3. function processes data and interacts with services
        4. result returned to user
    4. key characteristics
        1. auto scalling
        2. faster time-to-market
        3. high availability
        4. event-driven
        5. micro billing
        6. short lived functions

5. design principles
    1. stateless
    2. event-driven
    3. minimal and composable functions
    4. use queues
6. limitations and challenes
    1. cold starts
        1. latency
        2. mitigation: use warm up plugins or procisioned cocurrency
    2. vendor lock-in
        1. tied to provider ecosystem
    3. observability
        1. harder to trace request flows across functions
        2. solution: use distributed tracing (aws x-ray)
    4. resource limits
        1. timeout
        2. memeory and ephemeral storage constrains