Learning goals:
LG1 (apply): Explain, select, and apply basic design principles and object-oriented design patterns
LG2 (understand): Explain modern architectural designs

DARC:

- familiarize with design principles / design patterns
- program design patterns
- discuss about design patterns
- get to know current software architectures

Strategy Pattern:

- inheritance – super class
- subclasses decide how to implement steps in an algorithm
- interchangeable algorithms
    - disadvantages of using inheritance to provide duck behavior
        - code is duplicated across subclasses
        - runtime behavior change are difficult
        - hard to gain knowledge of all duck behaviors
        - changes can unintentionally affect other ducks
- interface
    - having subclasses implement flyable solves part

    - of problem
    - specific duck types just implement them

Design Principles:

- Identify the aspects of your application that vary and separate them from what stays the same.
    - Take what varies and “encapsulate” it so It won’t affect the rest of the code
    - result more flexibility in code and fewer unintended consequence
    - separating what changes from what stays the same
- Program to an interface, not an implementation
    - “program to a supertype”
    - behavior of ducks will live in separated classes
    - classes then can implement a particular behavior interface
- Favor composition over inheritance
    - HAS-A better than IS-A
    - instead of inheriting their behavior, the ducks get their behavior by being composed with the right behavior object
    - composition lets you change behavior at runtime

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

State Pattern:

- changes of internal state
- encapsulate interchangeable behaviors and use delegation to decide which behavior to use
- using state pattern will typically result in a greater number of classes in your design
- state pattern allows to change its behavior as the state of the context changes
- State and Strategy Pattern have the same class diagram, but they differ in intent

The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

Observer Pattern:

- Publisher + Subscriber = Observer Pattern
- We call publisher SUBJECT and subscribers the OBERVERS
- Subject manages data
- When data in subject changes, observers are notified
- The observers have subscribed to the subject to receive updates when the subjects data changes
- Loose coupling – When two objects are loosely coupled, they can interact, but they typically have very little knowledge of each other.
- We can add new observers any time

- We never need to modify the subject to add new types of observers
- We can reuse subjects or observers independently of each other.
- Changes to either the subject or an observer will not affect the other.

Design Principle:

- Strive for loosely coupled designs between objects that interact.

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Singleton Pattern:

- Ensures you have at most one instance of a class in your application
- Provides a global access point to that instance
- Multiple class loader could defeat the Singleton implementation and result in multiple instances

The Singleton Pattern ensures a class has only one instances and provides a global point of access to it.

Template Method Pattern:

- Template method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps

- Template method makes use of primitive Operations to implement an algorithm. Its decoupled from the actual implementation of theses operations.
- Gives us important technique for code reuse
- To prevent subclasses from changing the algorithm in the template method, declare the template method as final
- The Strategy and Template Method Patterns both encapsulate algorithms, the first by composition and the other by inheritance

Design Principle:

- The Hollywood Principle
  - Don't call us, we'll call you.
  - A low-level component never calls a high-level component directly
  - The high-level components control when and how

The Template Method Pattern defines the skeleton of an algorithm in a method, differing some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Factory Method Pattern:

- Creator has the abstract factory method what all creator subclasses must implement
- Concrete Creator implements the factory method which is the method that actually produces products
- Concreate Creator is responsible for creating one or more concrete products. It is the only class that has the knowledge of how to create these products

The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Abstract Factory Pattern:

- Pizza Factory, Pizza Store, Pizza

Design Principle:

- The Dependency Inversion Principle
  - o Depend upon abstractions. Do not depend upon concrete classes.

The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Adapter Pattern:

- The adapter implements the interface your class expect and talks to vendor interface to service your reqeust

Object-Oriented Design Patterns Finalization:

Discussed Patterns:

- **Strategy** – Behavioral
- **Observer** – Behavioral
- **Abstract Factory** – Creational
- **Factory Method –** Creational
- **Singleton** – Creational
- **Template Method** – Behavioral
- **State** – Behavioral
- **Adapter** – Structural

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

The Command Pattern encapulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests and support undoable opartions.

The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.

Design Principles:

- Identify the aspects of your application that vary and separate them from what stays the same.
- Program to an interface not an implementation.
- Favor composition over inheritance.
- Strive for loosely coupled designs between objects that interact.
- *Open Closed Principle:* Classes should be open for extension, but closed for modification.
- *Dependency Inversion Principle:* Depend upon abstractions. Do not depend upon concrete classes.

- *Law of Dementer:*Principle of Least Knowledge – talk only to your immediate friends.
- *The Hollywood Principle:* Don't call us, we'll call you.
- *Single Responsibility Principle:* A class should only have one reason to change.
- DRY – Don't repeat yourself.

Patterns:
- Layered Architecture
- Event-Driven Architecture
- Microservices Architecture Pattern

Layered Architecture:
- Layered architecture pattern, also called n-tier architecture
- Standard for most Java EE applications
- Closely matches traditional IT communication/ organizational structures

Pattern Description:
- Horizontal layers, each performing specific role e.g presentation logic or business logic
- Four standard layers: presentation, business, persistance, database
- In some cases business layer and persistence layer are combined
- Each layer forms abstraction, e.g. does not need to know about the other layers

- Seperation of concerns among components
- Well-defined component interfaces
- Limited component scope

Key Concepts:

- Typically, each layer is closed, request moves from layer to layer, layer of isolation
- An open layer can makle sense, e.g. services offering auditing and logging place below the business layer, business layer might directly access persistence layer (not via the services layer)
- It is important to document which layers are open/ closed and why

Pattern Example:

- Request from a business user to retrieve customer information
    - Customer screen – accepts request, display customer information
    - Customer delegate – knows which module can process that request
    - Customer object – aggregates the information
    - Customer dao – data access object

Considerations:

- Solid genera-purpose pattern
- Danger of architecture sinkhole anti-pattern
- Tend to lend iteself to monolithic applications

Pattern Analysis:

- Overall agility: Low
- Ease of depolyment: Low
- Testability: High
- Performance: Low
- Scalability: Low
- Ease of development: High

Event-Driven Architecture:

The event-driven architecture

- is a distributed, asynchronous pattern
- produces highly scalable applications
- is highly adaptable
- is suited for small and larage, complex applications
- is made up of processig components that
    - are highly decoupled
    - serve a single purpose
    - asynchronously receive and process events

Two different topologies:

- Mediator Topology
- Broker Topology

Mediator Topology:

- is useful for multiple step events
- is for events that require orchestration

Broker Topology:

- event-processor component
    - process event
    - publish new event
- there is no mediator
- broker topology is like a relay race
- relatively simple event processing flow
- no need for central orchestration

Considerations

- Complex to implement (asynchronous, distributed)
- Difficult to maintain a transactional unit of work
- Difficult to create, maintain and govern event-processor contracts
- Very important to settle on a standard data format (e.g. JSON, XML, Java Object…)

Pattern Analysis:

- Overall agility: High
- Ease of depolyment: High
- Testability: Low
- Performance: High
- Scalability: High
- Ease of development: Low

- EAI – Enterprise Application Integration
- ESB – Enterprise Service Bus

- BPMN – Business Process Model and Notation
- BPEL – Business Process Execution Language
- SOA – Service Oriented Architecture

Space-Based Architecture:

- great for small set of users
- with each layer – harder to scale
- Spaced-Based Architecture Pattern = Cloud Architecture Pattern
- Tuple space → distributed shared memory
- High scalability → replace DB by replicated in-memory data grids
- Virtualized-middleware is controller and manages
    - Requests
    - Sessions
    - Data replication
    - Distributed request processing
    - Process-unit deployment
- Complex and expensive pattern to implement
- Good choice for smaller web-based applications with variable load, e.g. social media sites, auctions etc.
- Not well suited for traditional large-scale relational database applications with large amounts of operational data
- typically comes with centralized data store for initial in-memory data grid load and asynchronously persist data updates
- common practice to seprate volatile transactional data

from non-active data (reduces memory footprint of in-memory data grid)
- although called cloud-based architecture, processing units and virtualized middleware can also run locally – no need for cloud based hosted services
- is specifically designed to address and solve scalability and concurrency issues

Processing unit contains:
- Application modules
- In-memory data grid
- (optional) asynchronous persistent store
- Data-Replication Engine

Messaging Grid:
- Manage input requests and session information
- Forwards requests to process unit
- Complexity ranges from simple round-robin to complex next-available algorithm

Data Grid – The Most Critical Component:
- Manages data replication between processing units
- Each processing unit contains exactly the same data
- Figure shows synchronous data replication between processing units
- In reality this is done in parallel asynchronously and very quickly

Pattern Analysis:

- Overall agility: High
- Ease of depolyment: High
- Testability: Low
- Performance: High
- Scalability: High
- Ease of development: Low

Microservices Architecture:

Core Concepts:

- Separately deployable units
    - Easier deployment
    - Increased scalability
    - High degree of application/component decoupling
- Distributable architecture
    - All components are decoupled
    - All components are accessed through some remote access protocol (JSM, REST, SOAP, RMI …)

API REST-based Topology

- Useful to expose small, self-contained individual services
- Consits of very fine-grained service components
- Service components are accessed via REST-based interface

- Requests are received through traditional web-based application screens
- User-interface layer is deployed as a seperate web application
- Service components tend to be larger and represent a portion of the application

Centralized Messaging Topology – Lightweight Message Broker instead of REST

Avoid Dependencies and Orchestration:
How small/large should a service be?
- Service components are too fine-grained if
    - Service components ne to be orchestrated
    - Inter service communication is required
- Too fine-hrained vs too coarse-grained

Considerations:
Microservices Architecture Pattern
- Is capable of real-time production deployment
- Change is isolated to specific service components
- Shares some of the complex issues of the event driven architecture

Pattern Analysis:
- Overall agility: High
- Ease of depolyment: High
- Testability: High

- Performance: Low
- Scalability: High
- Ease of development: High