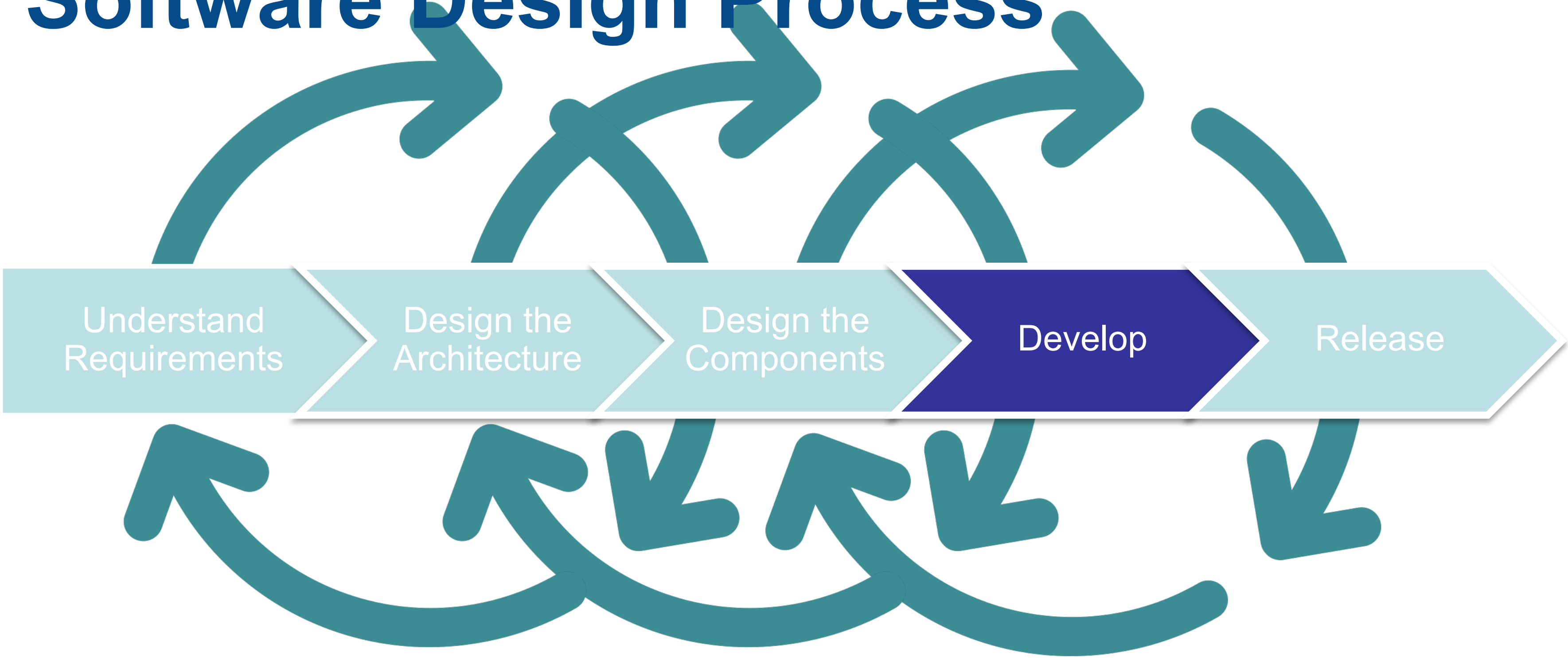
University of
South Australia

INFS 2044

Week 5
Patterns

Software Design Process



Software Design Recap

- Decomposition
- Interface Design
- Interaction & Implementation Design

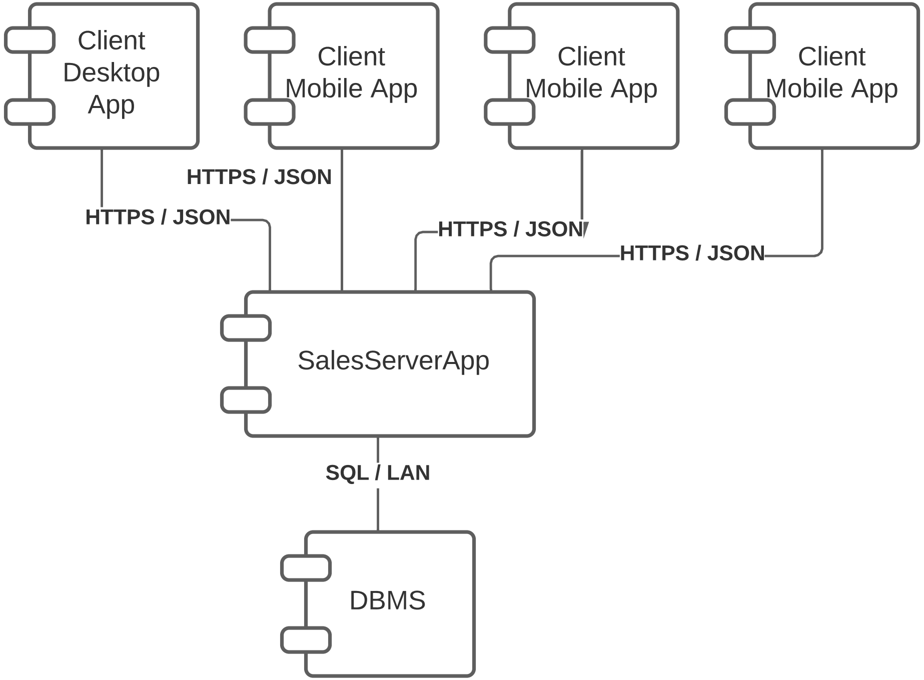
Learning Objectives

- Explain the different views of software (CO6)
- Understand software patterns (CO4)
- Apply design patterns (CO4)

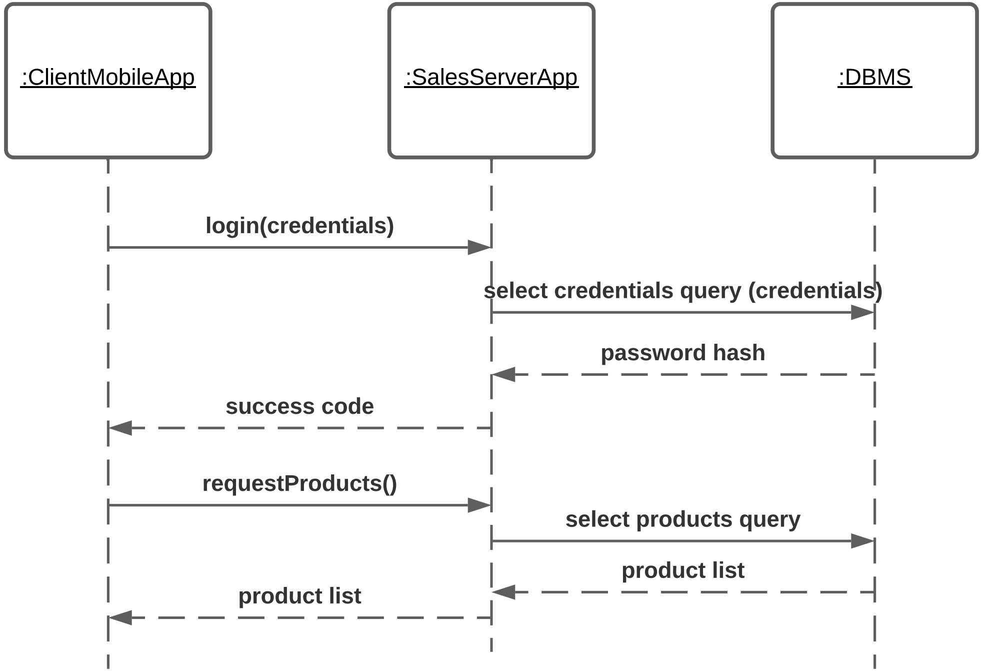
Software Views

- Module Viewpoints
  - Capture the logical entities in a system and how they are interconnected. [Modules, Layers, Packages]
- Component Viewpoints
  - Capture the runtime entities in a system and how they are interconnected [Components/Subsystems/Services, Queues]
- Allocation Viewpoints
  - Capture how entities are mapped onto other entities [Deployment: which components run where?]

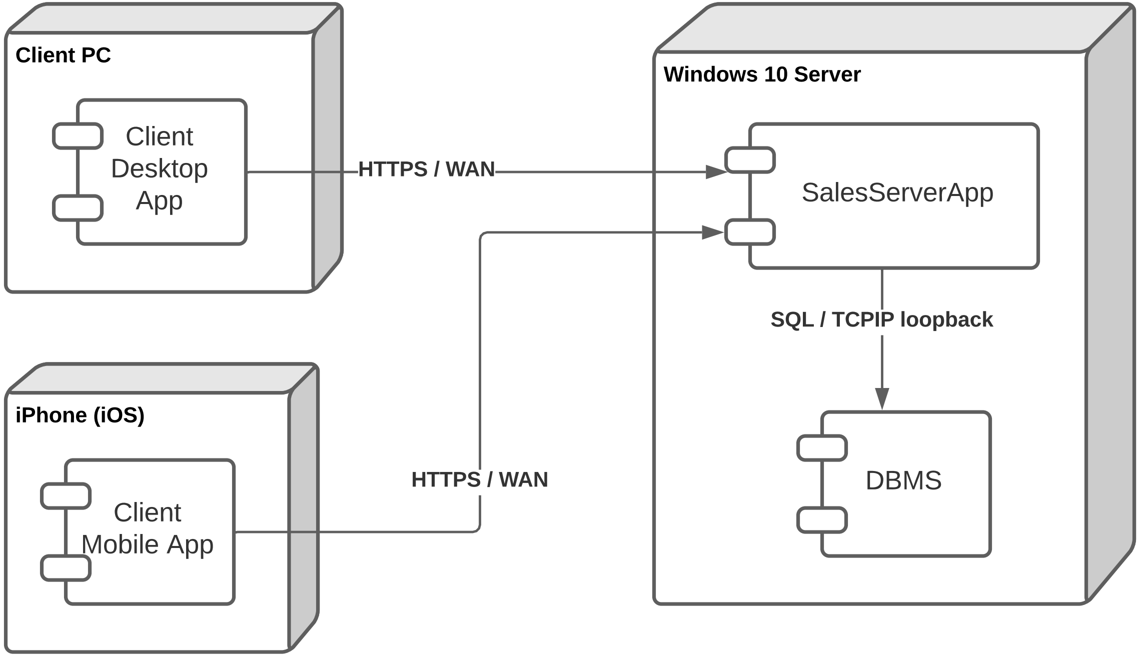
Component and Connector View



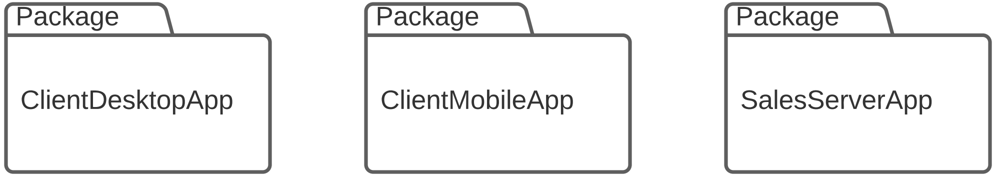
Interaction Views



Deployment View

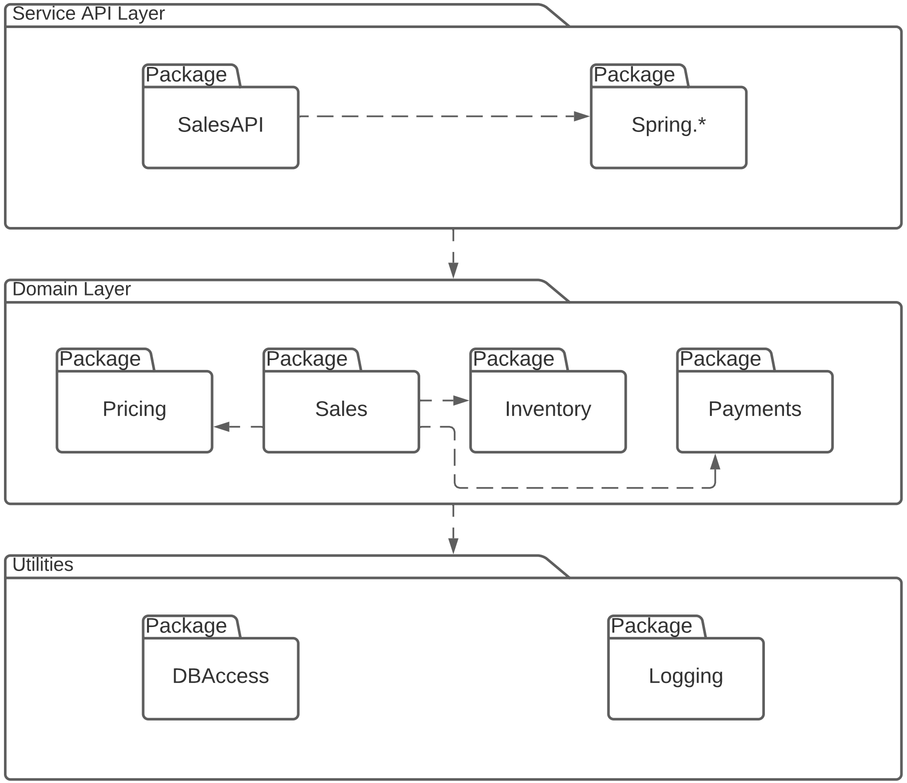


Modules / Packages View #1

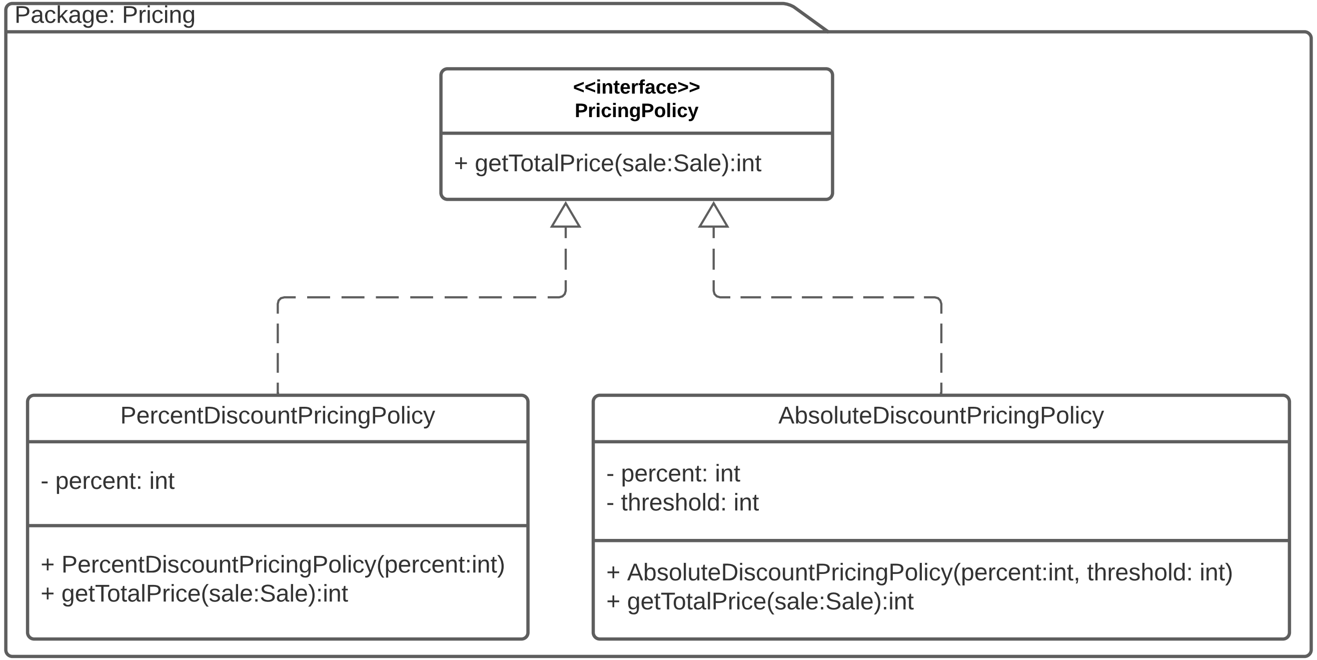


Modules / Packages View #2

Module Dependencies for SalesServerApp:



Implementation Structure View


Package: Pricing
<<interface>>
PricingPolicy
+ getTotalPrice(sale:Sale):int
PercentDiscountPricingPolicy
- percent: int
+ PercentDiscountPricingPolicy(percent:int)
+ getTotalPrice(sale:Sale):int
AbsoluteDiscountPricingPolicy
- percent: int
- threshold: int
+ AbsoluteDiscountPricingPolicy(percent:int, threshold: int)
+ getTotalPrice(sale:Sale):int

Patterns

- **Design Pattern**—standard design techniques and templates that are widely recognized as good practice
- For common design/coding problems, the design pattern suggests the best way to handle the problem.
- Provide common language among software engineers for communicating designs and implementation.

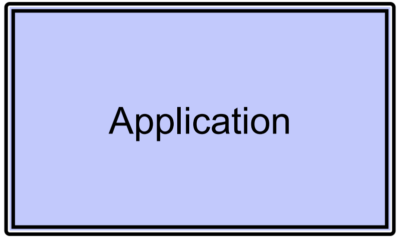
Pattern Elements

- Pattern name
- Problem that requires solution
- The pattern that solves the problem
- An example of the pattern
- Benefits and consequences of the pattern

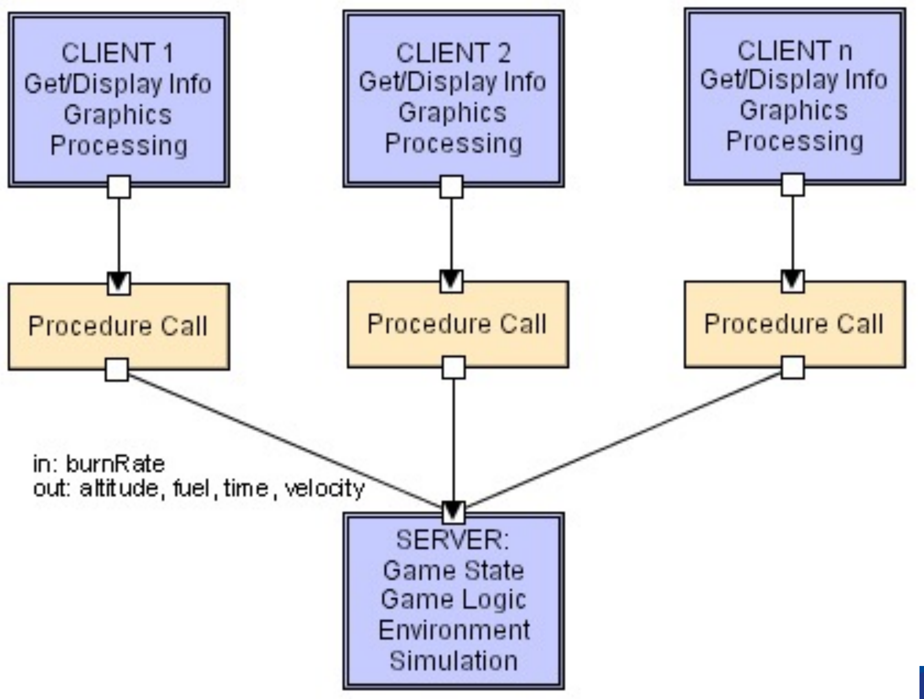
Architectural Patterns

- Architectural patterns provide general, reusable solution to a commonly occurring problem in software architecture within a given context
- Architecture = Big Picture
  - What the components are
  - Their roles and responsibilities
  - How they work together

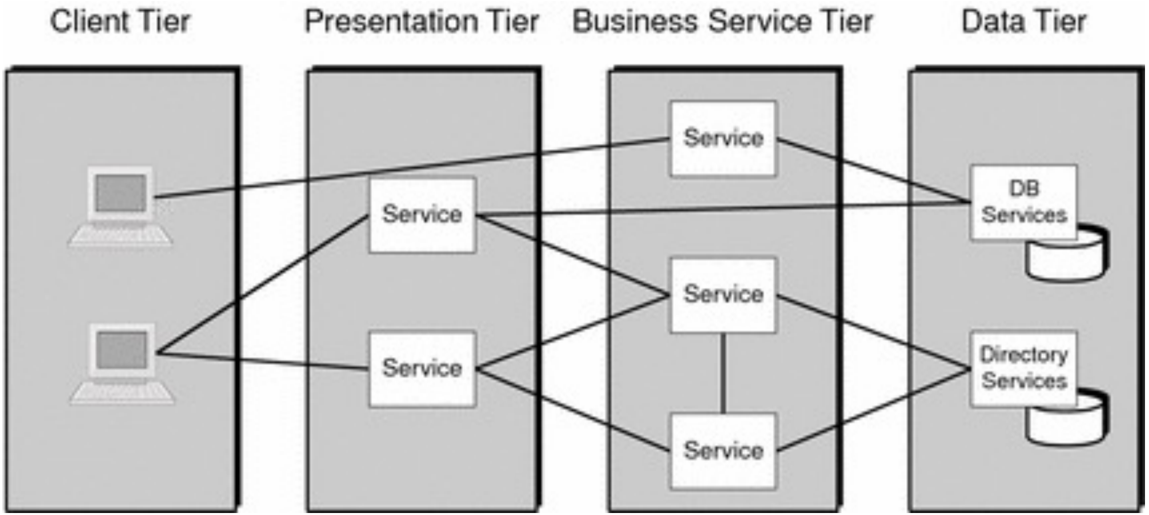
Monolith


Application

Client-Server Architecture

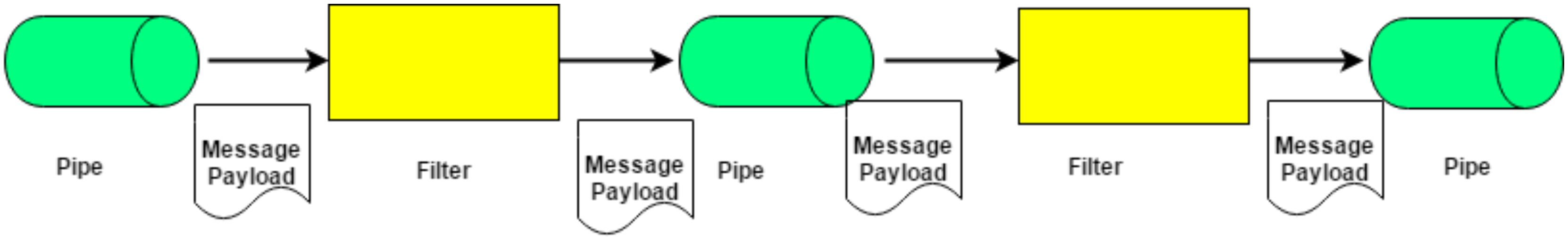


Multi-Tier Architecture



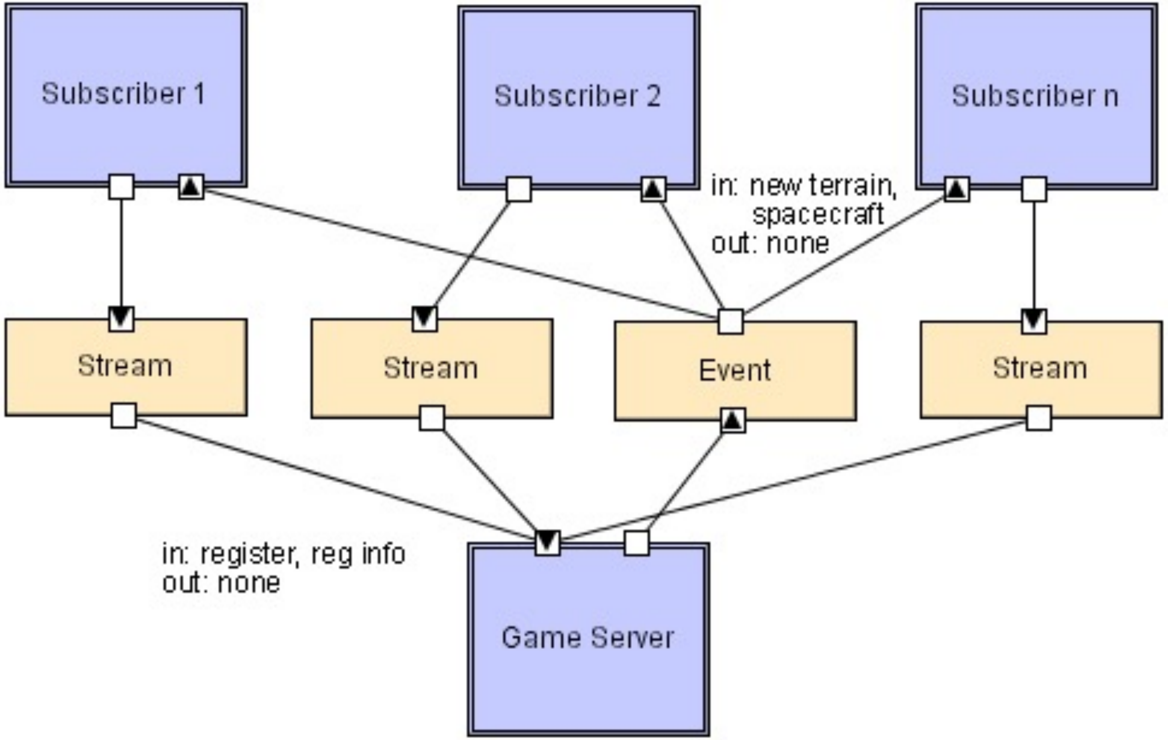
https://docs.oracle.com/cd/E19636-01/819-2326/images/MultiTierArch.gif

Pipes & Filters Architecture

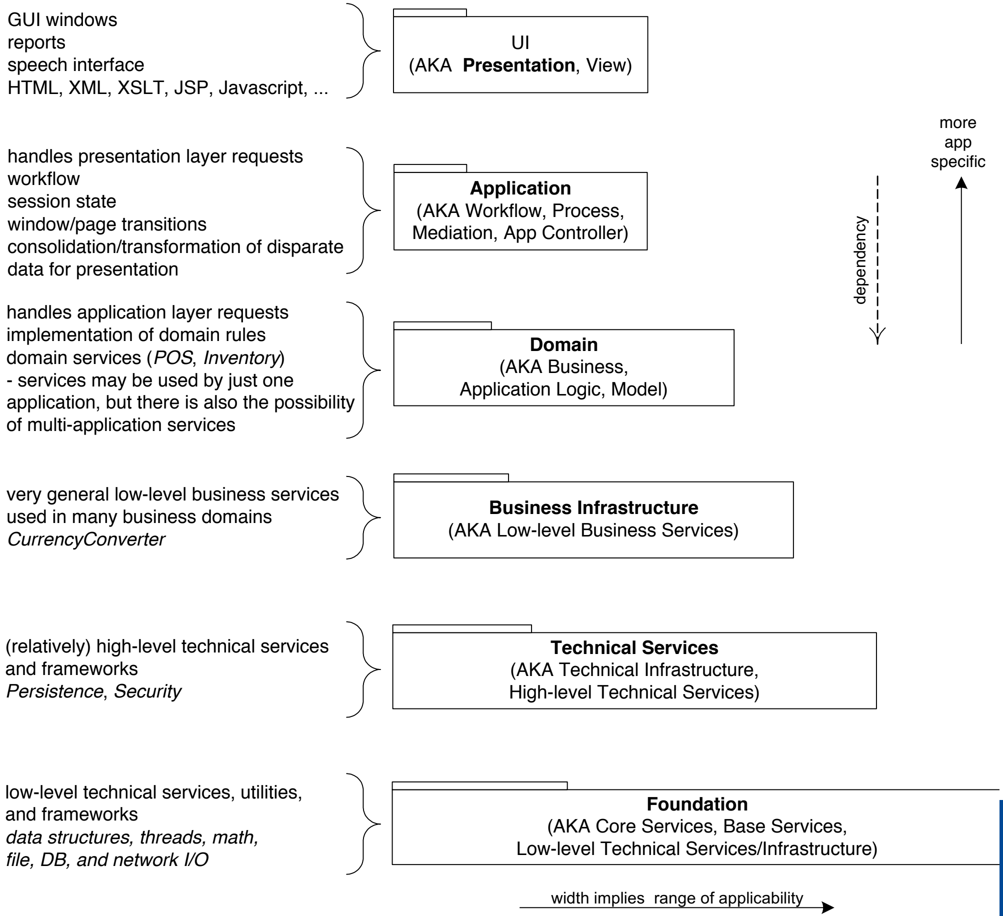


https://raghumb.gitbooks.io/a-guide-to-software-architecture/content/general_principles/pipes_and_filter_architecture.html

Event-Driven Architecture



Layered Architecture



GUI windows
reports
speech interface
HTML, XML, XSLT, JSP, Javascript, ...

UI
(AKA **Presentation**, View)

handles presentation layer requests
workflow
session state
window/page transitions
consolidation/transformation of disparate data for presentation

Application
(AKA Workflow, Process, Mediation, App Controller)

handles application layer requests
implementation of domain rules
domain services (*POS*, *Inventory*)
- services may be used by just one application, but there is also the possibility of multi-application services

Domain
(AKA Business, Application Logic, Model)

very general low-level business services used in many business domains
CurrencyConverter

Business Infrastructure
(AKA Low-level Business Services)

(relatively) high-level technical services and frameworks
Persistence, *Security*

Technical Services
(AKA Technical Infrastructure, High-level Technical Services)

low-level technical services, utilities, and frameworks
data structures, threads, math, file, DB, and network I/O

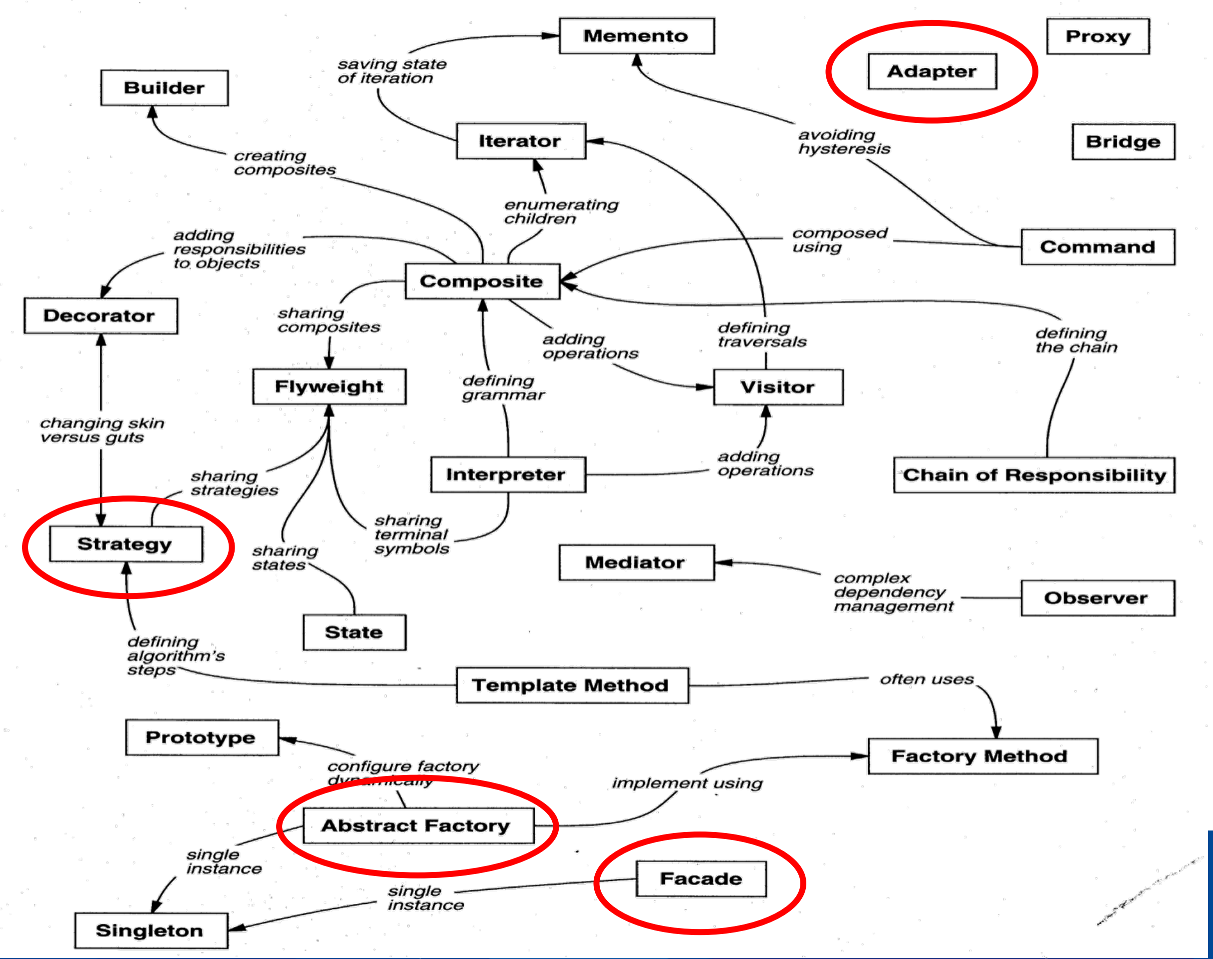
Foundation
(AKA Core Services, Base Services, Low-level Technical Services/Infrastructure)

Larman, C (2004) Applying UML and Patterns

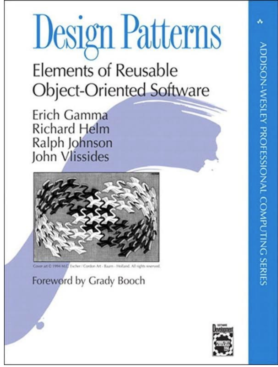
Design Patterns

- Design patters provide solutions to common design and implementation problems

- Three kinds of patterns
  - Behavioural patterns
  - Structural patterns
  - Creational patterns



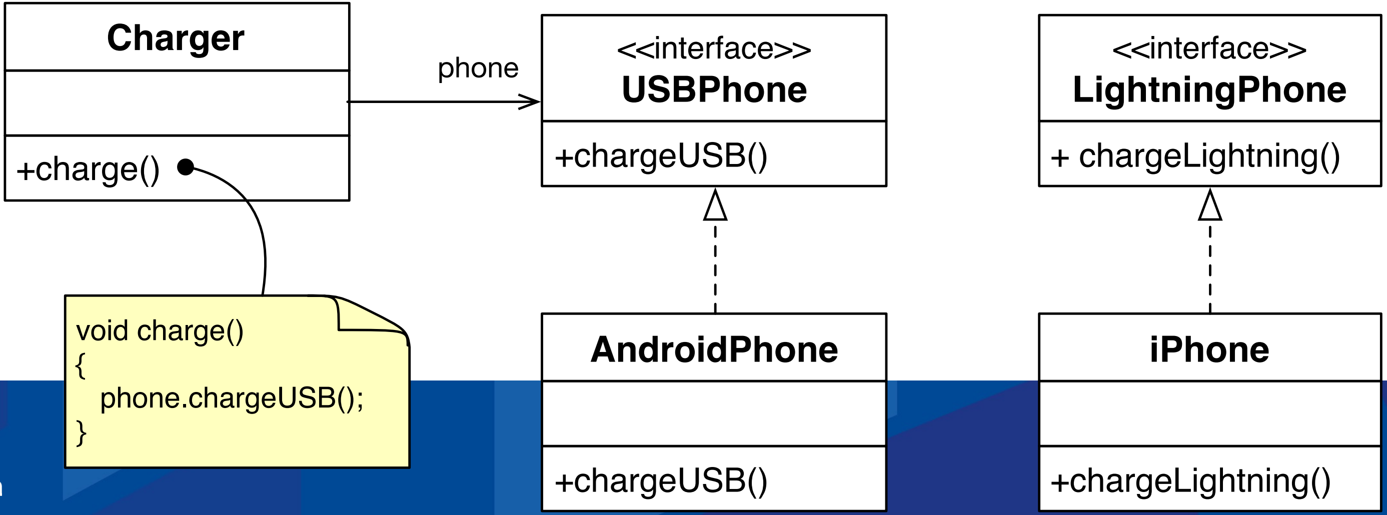
Memento
Builder
Adapter
Proxy
Bridge
saving state of iteration
Iterator
creating composites
avoiding hysteresis
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
Interpreter
adding operations
Chain of Responsibility
sharing strategies
Strategy
sharing terminal symbols
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
configure factory dynamically
Factory Method
implement using
Abstract Factory
single instance
Facade
single instance
Singleton



Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Adapter Pattern

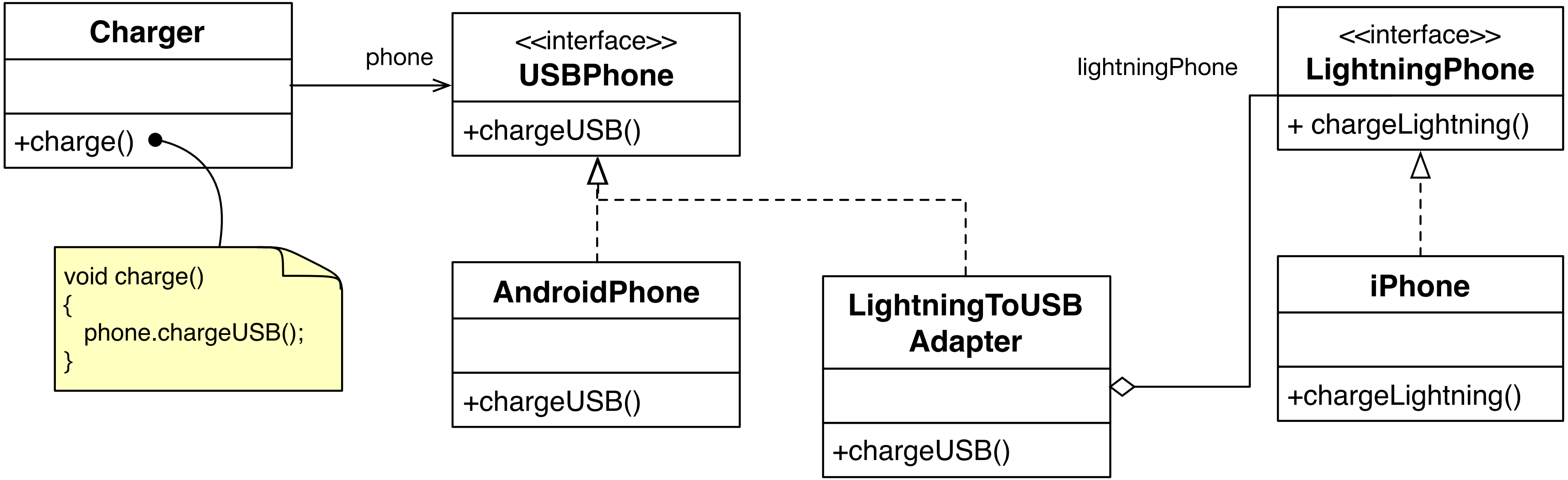
- **Problem:** How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?



```
void charge()
{
   phone.chargeUSB();
}
```

Adapter Pattern Solution

- **Problem:** How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?
- **Solution:** Convert the original interface of a component into another interface, through an **intermediate adapter object**.
- *A solution is to add a level of indirection with objects that adapt the varying external interfaces to a consistent interface used within the application.*

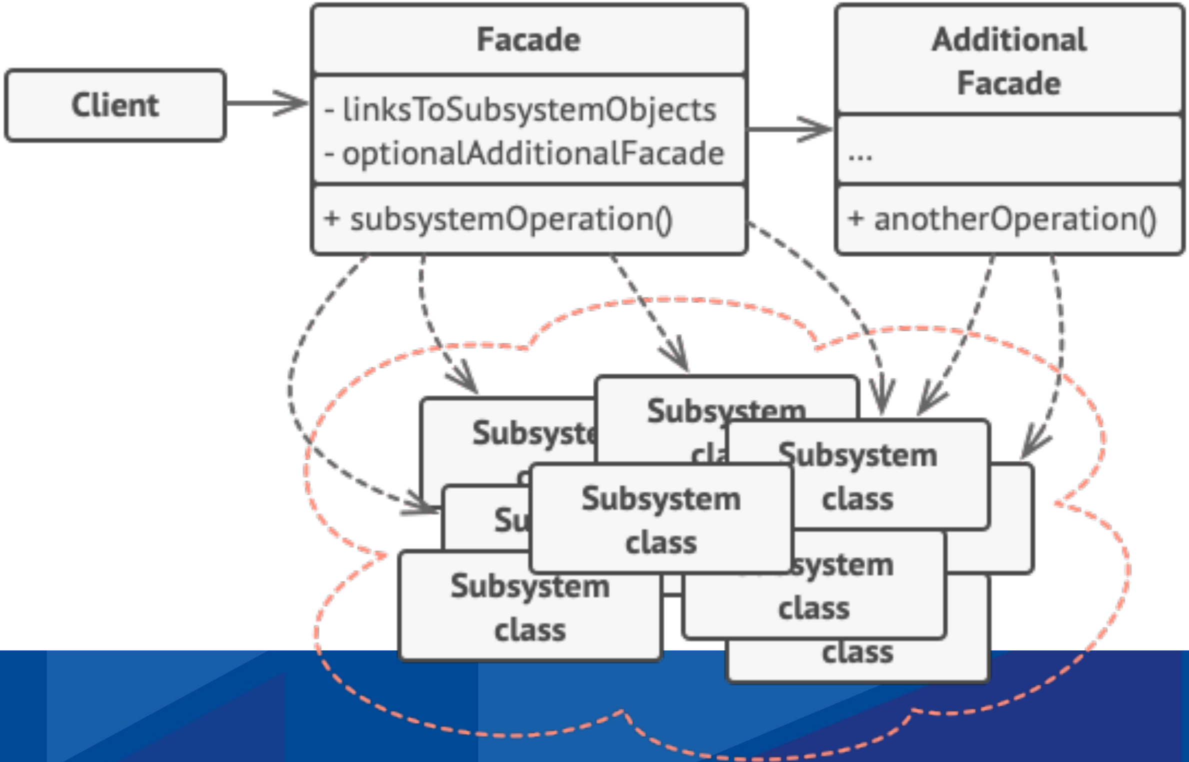


```
void charge()
{
   phone.chargeUSB();
}
```

Façade Pattern

- **Problem:** A common, unified interface to a disparate set of implementations or interfaces – such as within a subsystem – is required. There may be undesirable coupling to many things in the subsystem or the implementation of the subsystem may change.
- **Solution:** Define a single point of contact to the subsystem – a façade objects that wraps the subsystem. This facade object presents a single unified interface and is responsible for collaborating with the subsystem components.

Façade Pattern Solution



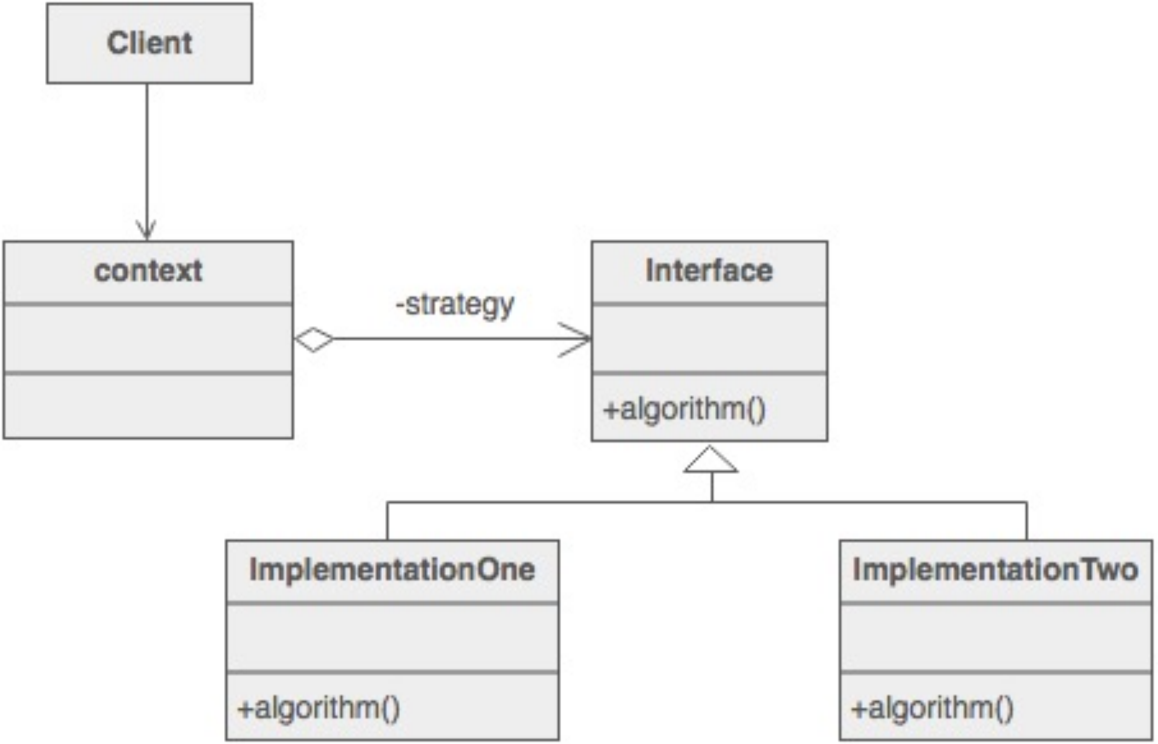
https://refactoring.guru/design-patterns/facade

Strategy Pattern

- **Problem:** How to design for varying but related algorithms or policies? How to design for the ability to change these algorithms or policies?

- **Solution:** Define each Algorithm/ policy/ strategy in a separate class, with a common interface.

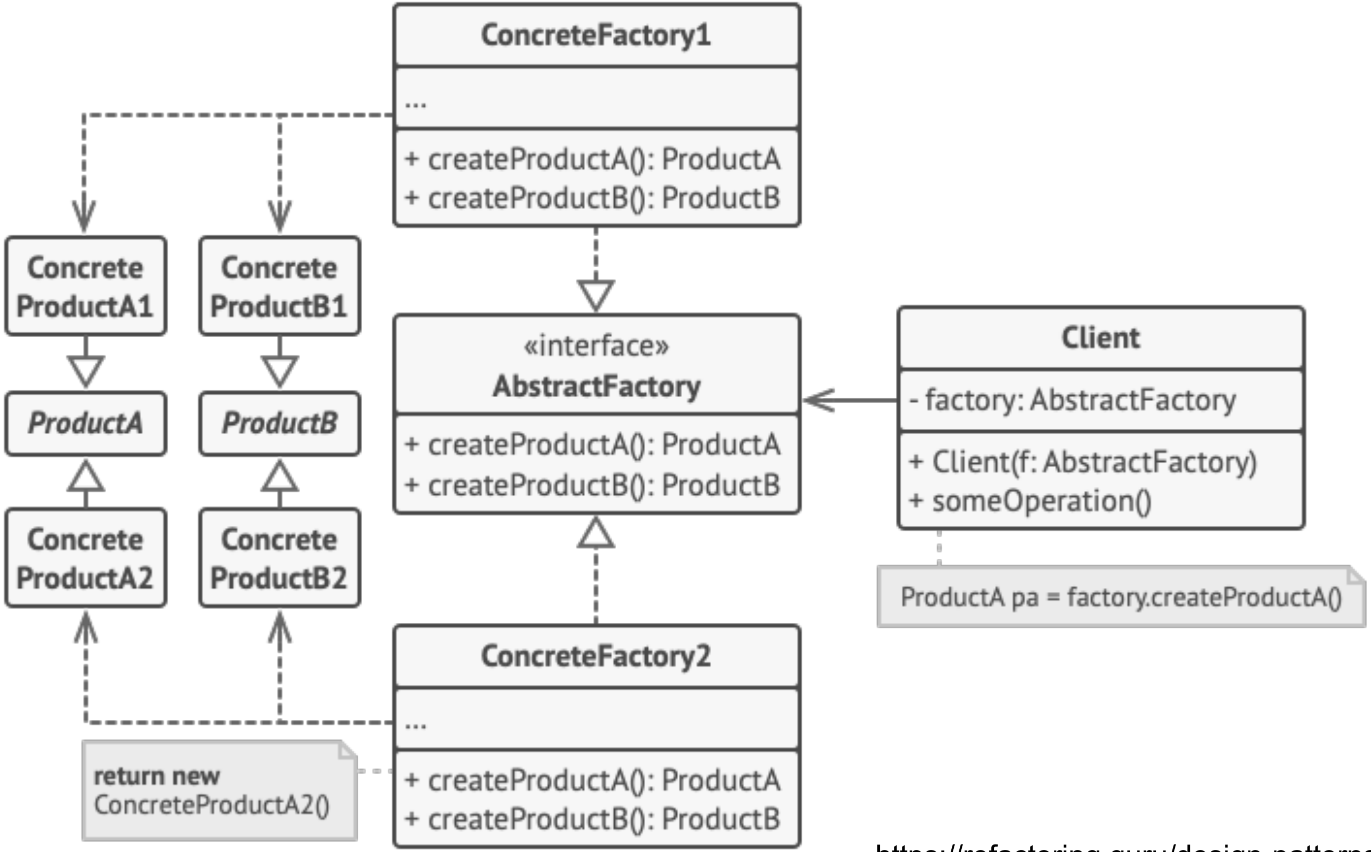
Strategy Pattern Solution



Abstract Factory Pattern

- **Problem:** Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities for better cohesion, and so forth?
- **Solution:** Create a new object called a Factory that handles the creation. Avoid using the “new” operator to create instances.

Factory Pattern Solution



ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
Concrete ProductB1
ProductA
ProductB
Concrete ProductA2
Concrete ProductB2
«interface»
AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

https://refactoring.guru/design-patterns/abstract-factory

Summary

- Choose the right software viewpoint for communicating key information with stakeholders
- Use patterns to leverage experience and facilitate communication
- Architectural patterns define the overall structure and behaviour of a system
- Design patterns provide prototypical solutions to common design problems
- As a software engineer, you need to know patterns to communicate effectively with your peers

Activities this Week

- Read the required readings
- Participate in Workshop 5
- Complete Quiz 5
- Continue working on Assignment 1

University of
South Australia