Software Design Patterns: Session Six

Sahar Mostafa
UC Berkeley Extension
Summer 2020

Week In Review

1. Behavioral Design Patterns
2. Observer Pattern
3. State Pattern
4. Strategy Pattern
5. Chain of Responsibility
6. Exercises

Construction

Sessions Review - Singleton

Reusable single object

Can be extended via interface or subclass

Hard to destruct the object, when all references are deleted

Harder to unit test

Introducing global state to the system harder to maintain

Sessions Review - Factory

One entity responsible for instantiation

Type of object is not predetermined

Separate instantiation from actual object behavior

Enforces subclass to exist per type

Sessions Review - Builder

Internal representation is flexible to alter

Encapsulates code for construction and representation

Requires creating a separate ConcreteBuilder

Builder must be mutable

Sessions Review - Prototype

Hides complexities of creating objects

Enables adding or deleting objects at runtime

Enforces classes to implement clone, cause circular reference

Structural

Sessions Review - Adaptor

Flexible using interfaces implementations may swap

Client is agnostic to the behavioral change

Overhead of adding an extra layer, performance

Adaptation chain various adaptors are required

Sessions Review - Decorator

Extend functionality at runtime
alternative to subclassing

Complicate the process of instantiating the component

Difficult for decorators to keep track of other decorators

Sessions Review - Proxy

Avoids duplication of heavy objects optimization

Ensures security of a system

Bottleneck with a new layer of abstraction introduced

Sessions Review - Facade

Enables libraries to be more readable

Hides complexities reducing dependency

Wrap poorly designed collection of APIs

Strong coupling between subsystem internals and facade

Need to change with any change to the subsystem

Behavioral

Sessions Review - State

Polymorphic behavior of object changing at runtime based on state

Improves cohesion state specific behavior is grouped together

Number of states may grow hard to keep track

Sessions Review - Observer

Loose coupling between objects that interact with each other

Observers can be added or removed anytime

ConcreteObserver involving inheritance composition not supported

Undependable behavior with inconsistency and race conditions

Sessions Review - Strategy

Easy to switch between strategies at runtime

Clean code without conditional-infested code

Client must be aware of the strategies to make a choice

Clients might be exposed to implementation issues

Class must exist per strategy, increase number of classes


15 mins Break

Design Principles

SOLID

Single Responsibility Principle

Conway's law: The best structure for a software system is heavily influenced by the social structure of the organization that uses it.

Each software module has one, and only one, reason to exist

Open-Closed Principle

Bertrand Meyer made this principle famous in the 1980

Software solutions should be flexible to change, design to allow the behavior of those systems to be changed by adding new code, rather than changing existing code.

Liskov Substitution Principle

Barbara Liskov's famous definition of **subtypes** 1988

Clear contract extending interchangeable parts

System parts must adhere to the contract allowing those parts to be substituted one for another

Interface Segregation Principle

Avoid depending on things the system does not use

ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy

Inheritance vs Composition

Inheritance: defined statically at compile time

special version of the super class

subclass can override some operations

Less flexible
Fixed behavior inherited from super class

Inheritance breaks encapsulation

Composition: defined dynamically at run-time through objects acquiring references to other objects

Objects respect each other interfaces

is-A sub type vs has-A relationship type

inter-dependecy relationship between objects

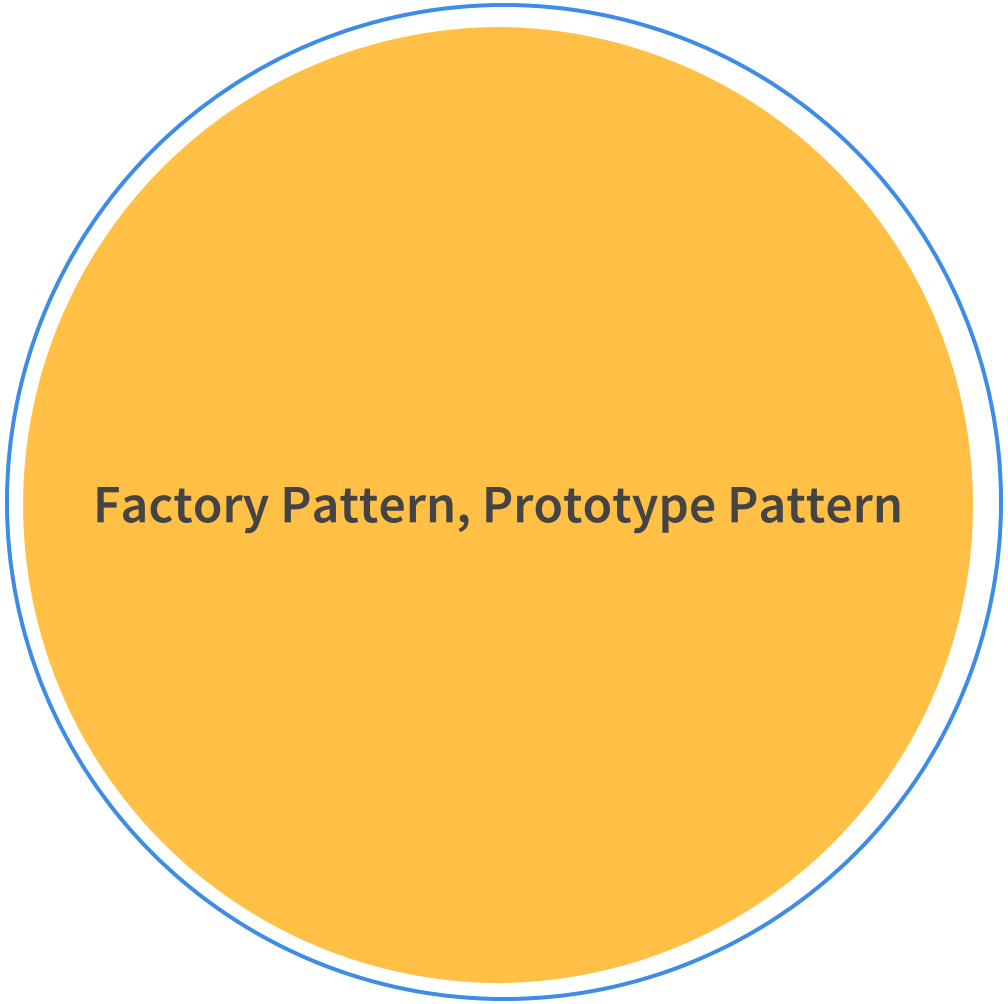
Encapsulation, smaller class hierarchies

More objects, inter-dependecy relationship between objects

Design Rules

- Program to an interface, not an implementation
- Favor Object Composition over Class Inheritance

General Design Rules

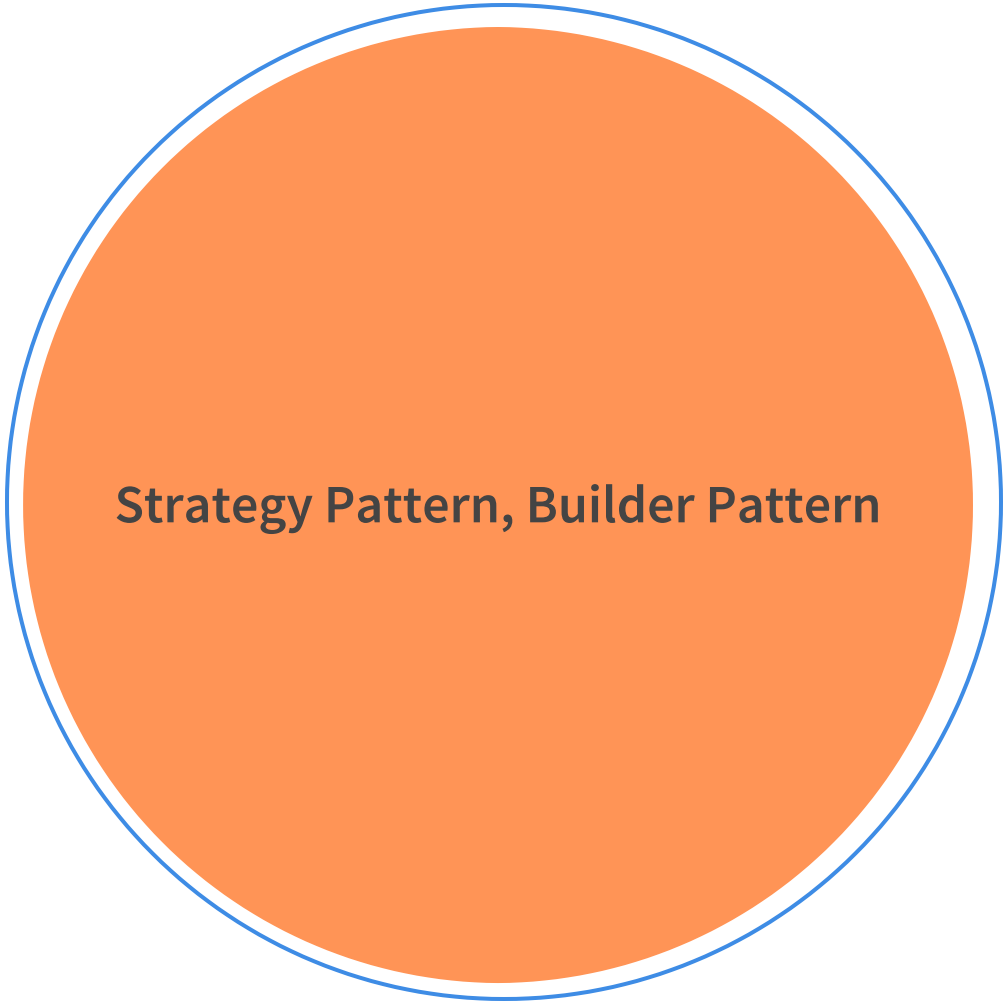


Create object indirectly

Do not commit to a particular implementation

Use interfaces

General Design Rules



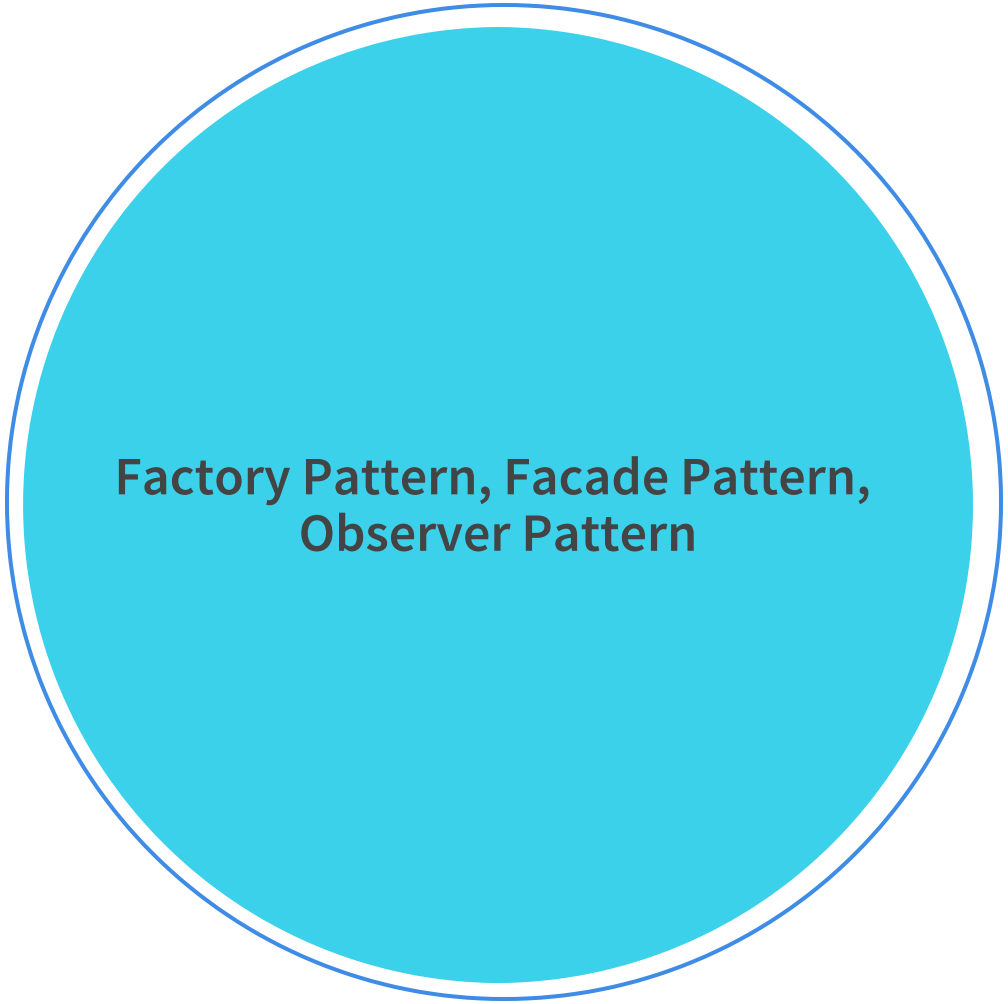
Algorithmic Dependency

Extending or optimizing an algorithm

Dependent objects will have to change

Algorithms requiring changes should be isolated

General Design Rules



Tight Coupling

Tightly coupled classed are hard to reuse in isolation

Leads to monolithic systems

Systems should be easier to learn, port, modify and extend easily

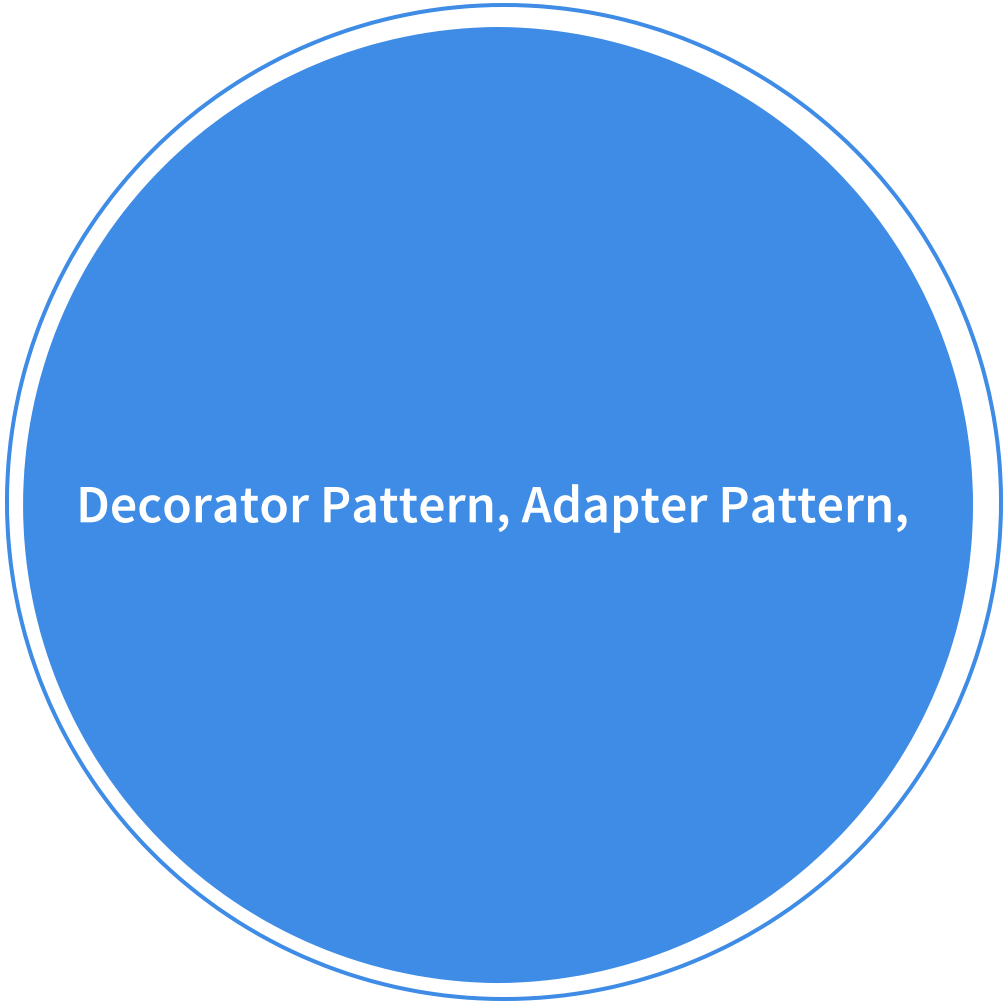
General Design Rules

Extending Functionality

Alternative to inheritance, avoid explosion of subclasses

Compose existing objects in new ways to achieve intended functionality

General Design Rules



Alter Classes

Modify class without access to source code, commercial class libraries

Modify existing subclasses


15 mins Break

Group Exercises

Shopping System

- Design and implement a shopping system that accepts users orders and add to a shopping cart
- Add, list, update, delete
- Items should have a category: clothing, decor, electronics ...etc
- Provide UML class diagram and test cases highlighting how the classes can be called
- Bonus point: Recommend other items

Design Analysis

- User management system with a singleton resource management
- Shopping cart assign per user shopping session
- Orders are immutable
- Recommendation Engine

Mid-term Project Assignment

- Design an invitation RSVP service that creates events and collects invitation responses Deliverables
- Feature Breakdown
- DB Storage tables design
- Implement methods for creating an account and recommendation of events
- UML diagram and unit tests

  Code must compile successfully

Questions

1. How to create a themed invitation to guests based on the event type?
2. How many Birthday/Wedding/Graduation Party/Get Together events the system has?
3. How many invitations sent for an event?
4. How many accepted/rejected invitation responses received?
5. How to notify a host when a guest responds?


15 mins Break

Exercise Solution

Code Review

Code Review

1. **Step 1: git repo clone session six**
   https://github.com/SMostaf/COMPSCIX418.2Step

2. **Discuss difference between the two directions**

Next Week

Session Seven

1. Transition to services
2. Software architecture building services
3. Microservices Introduction
4. Cloud emergence
5. Group Presentations
6. Exercises

Pizza Ordering System

- **Design an order management system for ordering pizza online, user can customize the pizza order add different toppings**
- **Represent items on the menu: pizza, beverages and appetizers**
- **All products have a price and a size: small, medium and large**

  Every pizza contains at least cheese and tomato sauce (required items)

  Additional toppings (pineapple, extra cheese, ..etc.) can be ordered as well

  Pizzas with certain topping combinations are named with base name appearing on order, for example: Pizza Margherita + topping names

  In addition pizza can have different crust variations
- **Design Patterns to use. Show the immutable objects in the system.**
- **Provide UML class diagram and test cases highlighting how the classes can be called**