



# Object-Oriented Programming - 201

---

A beginner to intermediate exploration of Object-Oriented  
Programming

**Adam Beeson**  
Principal Engineer- OneRail





# Today's Topics

---

- The When - ADD-friendly introduction & history
- The What - Core concepts
- The Why - Design Principles
- The How - Design Patterns
- The End - Summary

Note: Code examples  
included here are using  
TypeScript syntax







# "Hello, World."

---



Above: Alan Kay writing SmallTalk, probably. (1969)

- Lambda Calculus
- Turing Computation
- Alan Kay (1966)
- Data Structure Representation
- "Message Sending, Encapsulation, and Dynamic Binding."
- Simula, SmallTalk





# "The Big Idea"

---

## Messaging

**Avoiding shared mutable state** by encapsulating state and isolating other objects from local state changes. The only way to affect another object's state is to ask (not command) that object to change it by sending a message. State changes are controlled at a local, cellular level rather than exposed to shared access.

## Encapsulation

**Decoupling** objects from each other — the message sender is only loosely coupled to the message receiver, through the messaging API.

## Data-Binding

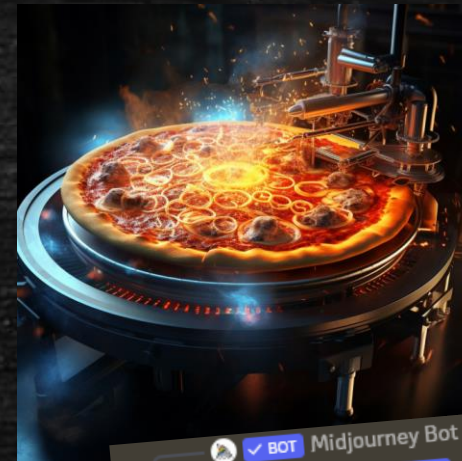
**Adaptability and resilience to changes** at runtime via late binding. Runtime adaptability provides many great benefits that Alan Kay considered essential to OOP.







# Core Concepts

- Classes
  - Class Members
- Inheritance
- Interfaces
- Implementations



Midjourney Bot  BOT the above pizza but way more technological. It should look like it is being created by a science fiction pizza factory - Image #4 @gerhard

Midjourney Bot  BOT Today at 11:11 AM



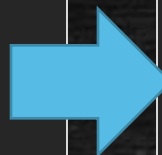


# Classes

A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit. A class provides a definition for *instances* of the class, also known as *objects*.

```
class Pizza {
  private _name: string;
  private _toppings: string[];
  get name() {
    return this._name;
  }
  get toppings() {
    return this._toppings;
  }
  constructor(name: string, toppings: string[]) {
    this._name = name;
    this._toppings = toppings;
  }

  describe(includeName: boolean): void {
    console.log(
      `This pizza has the following toppings:
      ${this.toppings}.` +
      (includeName ? ` It is named ${this.name}.` :
      '')
    );
  }
}
```



```
const myPizza = new Pizza(`Adam's Pizza`, ['pepperoni',
  'black olives']);

myPizza.describe(true); // "This pizza has the
  following toppings: pepperoni,black olives. It is
  named Adam's Pizza."
```



# Class Member Declaration

```
abstract class BaseClass {

    readonly var1: string; // Only can be assigned in the constructor.
    private var2: string; // Only can be accessed in the class.
    protected var3: string; // Only can be accessed in the class and its subclasses.

    constructor() {
        this.var1 = 'var1'; // readonly properties can be assigned in the constructor.
        this.var2 = 'var2';
        this.var3 = 'var3';
    }

    // static methods can be called without instantiating the class.
    static staticMethod(x: string, y: string): string[] {
        return [x, y];
    }
}

class SubClass extends BaseClass {

    constructor() {
        super();
        this.var1 = 'var1'; // Cannot assign to 'var1' because it is a read-only property.
        this.var2 = 'var2'; // Property 'var2' is private and only accessible within class 'BaseClass'.
        this.var3 = 'var3';
    }
}

const staticMethodResult = BaseClass.staticMethod('x', 'y'); // ['x', 'y']
const mySubClass = new SubClass();
mySubClass.var1 = 'aaa'; // Cannot assign to 'var1' because it is a read-only property.
mySubClass.var2 = 'bbb'; // Property 'var2' is private and only accessible within class 'BaseClass'.
mySubClass.var3 = 'ccc'; // Property 'var3' is protected and only accessible within class 'BaseClass' and its
                        subclasses.
```

Class members can have declarations that control their access as well. These are some examples in TypeScript that are widely available in other OOP languages.





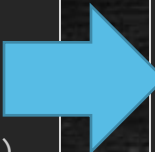
# Base Classes & Inheritance

A base class is a class with properties and functions, but those members can be inherited by another class and implement those features from the base class without redefining them.

```
// Food is the base class
abstract class Food {
  constructor(public name: string) {}


  describe(): void {
    console.log(`This food is named ${this.name}.`);
  }
}

// Pizza is a subclass of Food
class Pizza extends Food {
  constructor(name: string, public toppings: string[])
  {
    super(name);
  }
}
```



```
const myPizza = new Pizza(`Adam's Pizza`, ['pepperoni',
  'black olives']);

myPizza.describe(); // "This food is named Adam's
  Pizza."
```

- The **abstract** key prevents instantiation of a class.
- **super(name)** "Supes up" the base class (instantiates its members that are needed when it is constructed).
-  This example infers public properties at instantiation by setting constructor args to **public**.
- A subclass can only inherit a single base class.

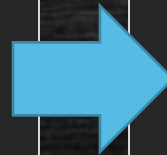




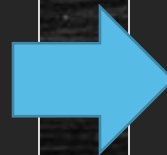
# Interfaces & Implementation

An interface defines a contract that can be implemented by classes and structs. You define an interface to declare capabilities that are shared among distinct types.

```
interface IPizza {  
    name: string;  
    toppings: string[];  
    describe(includeName: boolean): void;  
}  
  
class BadPizza implements IPizza {  
    constructor(name: string, toppings: string[]) {  
    }  
}  
  
class GoodPizza implements IPizza {  
    constructor(public name: string, public toppings:  
        string[]) {}  
    describe(includeName: boolean): void { }  
}
```



Class 'BadPizza' incorrectly implements interface 'IPizza'. Type 'BadPizza' is missing the following properties from type 'IPizza': name, toppings, describe



File change detected. Starting incremental compilation... Found 0 errors. Watching for file changes.

- A class can implement multiple interfaces. We will visit this more when talking about the Interface Segregation Principal.





# Design Principles (SOLID)

---

The SOLID acronym is a subset of OOP specific design principles decreed and pontificated by his Holiness, Robert "Uncle Bob" Martin. Software principles are some of the means which justify the ends of the strength of OOP.

- S – Single Responsibility
- O – Open/Close
- L – Liskov Substitution
- I – Interface Segregation
- D – Dependency Inversion







# S – Single Responsibility Principle

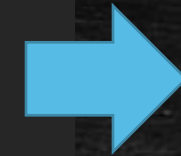
"A module should be responsible to one, and only one, actor."

- Code organized should be cohesive (independent of its physical containment)

A couple major benefits of SRP

- Prevents duplicate code
- Alleviates merge pain points
- Readability/Maintainability

```
class PizzaShop {
    calculateFee() {
        // ...
    }
    createLabel() {
        // ...
    }
    requestPickup() {
        // ...
    }
    createOrder() {
        // ...
    }
    cancelOrder() {
        // ...
    }
    getOrder() {
        // ...
    }
    authorize() {
        // ...
    }
    capture() {
        // ...
    }
    refund() {
        // ...
    }
}
```



```
class ShippingService {
    calculateFee() {
        // ...
    }

    createLabel() {
        // ...
    }

    requestPickup() {
        // ...
    }
}
```

```
class PaymentService {
    authorize() {
        // ...
    }

    capture() {
        // ...
    }

    refund() {
        // ...
    }
}
```

```
class OrderService {
    createOrder() {
        // ...
    }

    cancelOrder() {
        // ...
    }

    getOrder() {
        // ...
    }
}
```



# 0 – Open/Closed Principle

"An object should be open for extension, but closed for modification"

- Make code easy to extend without incurring a high impact of change.

## A couple major benefits of OCP

- Directional control and Information hiding.
- Prevents regression or consumer/actor dependence deprecation.

```
class PizzaMaker {  
  
    static preparePizza(name: string) {  
        console.log(`Preparing a new pizza for you, ${name}!`);  
    }  
  
}  
  
// before enhancing our preparePizza method, all is good.  
PizzaMaker.preparePizza('Adam'); // Preparing a new pizza for you, Adam!  
  
// after enhancing our preparePizza method, we get an error.  
class PizzaMaker {  
    static preparePizza(name: string, includeDate: boolean) {  
        console.log(`Preparing a new pizza for you, ${name}!`);  
    }  
}  
  
PizzaMaker.preparePizza('Adam'); // An argument for 'includeDate' was not provided.  
  
// making the second argument optional, we can now call the method without it.  
class PizzaMaker {  
    static preparePizza(name: string, includeDate?: boolean) {  
        console.log(  
            `Preparing a new pizza for you, ${name}!` + includeDate  
            ? `We threw it in the oven at ${new Date().toISOString()}`  
            : ''  
        );  
    }  
}  
  
PizzaMaker.preparePizza('Adam'); // Preparing a new pizza for you, Adam!  
PizzaMaker.preparePizza('Adam', true); // Preparing a new pizza for you, Adam! We  
threw it in the oven at 2019-07-28T15:00:00.000Z
```





# L – Liskov Substitution Principle

"Objects of a base class (or interface) should be able to be replaced with objects of a subclass (or parallelly inheritor) without affecting the correctness of the program"

Objects correctly respecting substitutability will always protect the correctness of application consumers.

The interface here protects the substitutional properties of either type implementing IPizza.

```
interface IPizza {
  name: string;
  toppings: string[];

  preparePizza(name: string, includeDate?: boolean): void;
}

// concrete implementations of IPizza
class DeepDishPizza implements IPizza {
  constructor(public name: string, public toppings: string[]) { }
  preparePizza(name: string, includeDate: boolean = false): void {
    // do something specific to deep dish pizza
    console.log(`This is a deep dish pizza!`);
    console.log(`Preparing ${this.name}'s pizza with ${this.toppings.join(', ')}');
    if (includeDate) {
      console.log(`Preparing pizza on ${new Date().toLocaleDateString()}`);
    }
  }
}

class TavernPizza implements IPizza {
  constructor(public name: string, public toppings: string[]) { }

  preparePizza(name: string): void {
    console.log(`Preparing Tavern Style Pizza for ${this.name} with ${this.toppings.join(', ')}');
  }
}

const myDeepDishPizza: IPizza = new DeepDishPizza('Adam', ['pepperoni', 'sausage', 'onions']);
// This is a deep dish pizza!
// Preparing Adam's pizza with pepperoni, sausage, onions
// Preparing pizza on Thu Jul 30 2020

const myTavernPizza: IPizza = new TavernPizza('Erin', ['green peppers', 'olives']);
// Preparing Tavern Style Pizza for Erin with green peppers, olives
```



# I – Interface Segregation Principle

"Operations should be as uniquely segregated by interfaces as possible"

```
interface IPizza {  
    toppings: string[];  
    preparePizza(  
        name: string,  
        includeDate?: boolean  
    ): void;  
}
```

```
interface IDeepDishPizza extends IPizza {  
    doughType: 'Seminole' | 'Wheat';  
}
```

```
class DeepDishPizza implements IDeepDishPizza {  
    constructor(  
        public toppings: string[],  
        public doughType: 'Seminole' | 'Wheat'  
    ) {}  
  
    preparePizza(name: string, includeDate: boolean = false): void {}  
}
```

```
interface ITavernPizza extends IPizza {  
    squareCutSize: 'S' | 'M' | 'L';  
}
```

```
class TavernPizza implements ITavernPizza {  
    constructor(  
        public toppings: string[],  
        public squareCutSize: 'S' | 'M' | 'L'  
    ) {}  
  
    preparePizza(name: string): void {}  
}
```





# D – Dependency Inversion Principle

---

"Code should depend on abstractions, not concretions."

```
class PizzaFactory {
    static createPizza(type: 'DeepDish' | 'Tavern', toppings: string[]): IPizza {
        switch (type) {
            case 'DeepDish':
                return new DeepDishPizza(toppings, 'Seminole');
            case 'Tavern':
                return new TavernPizza(toppings, 'Large');
            default:
                throw new Error('Invalid pizza type...');
        }
    }
}

const myPizza: IPizza = PizzaFactory.createPizza('DeepDish', ['cheese', 'pepperoni']);
myPizza.preparePizza('Adam', true);
```

- Promotes architectural boundaries around code.
- Allows for the flow of control to be inverted for dynamic programs.
- Changes to concrete objects do not require changes to abstractions.





# Design Patterns



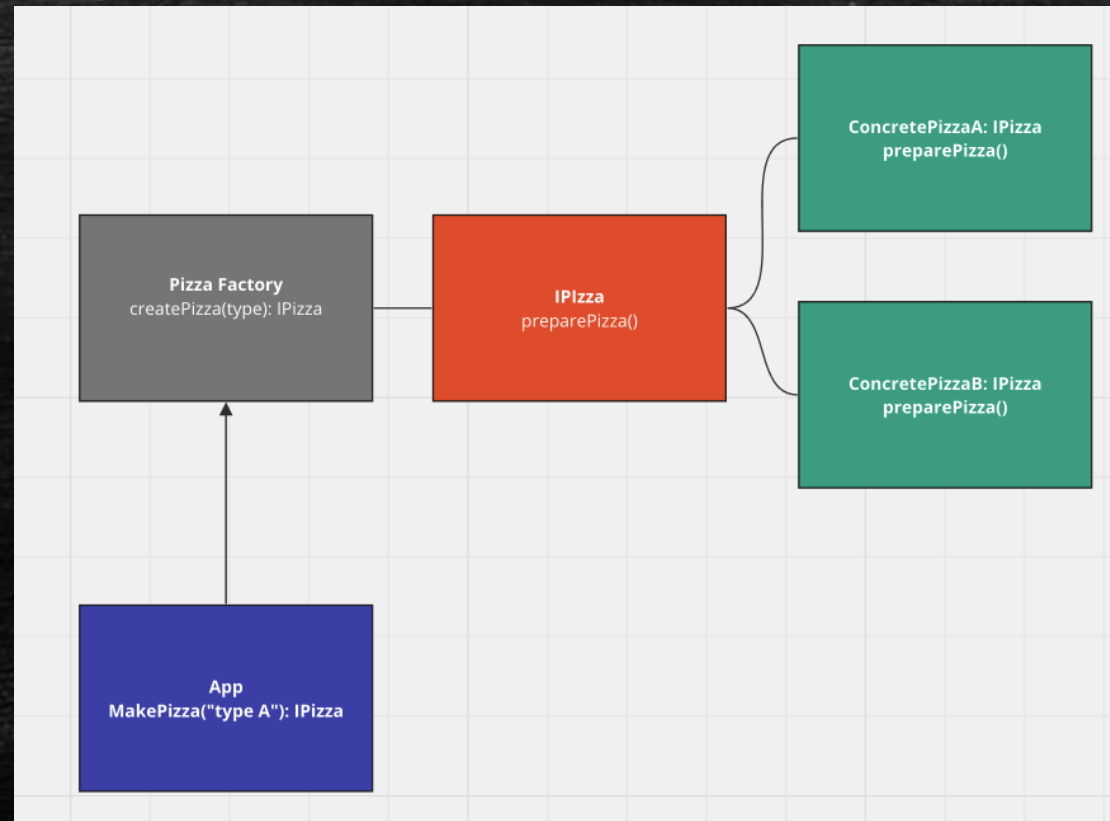
- Design Patterns are example best practices for building reusable logic that implements recommended and repeatable patterns for object oriented code.
- They Fall into 3 categories- Creational, Structural, and Behavioral
- They are most famously captured in the 1994 (!) text "Design Patterns," by programmers and authors infamously dubbed "The Gang of Four"





# Creational Example - Factory

The Factory Pattern allows us to define an interface for creating an object, but let subclasses decide which class to instantiate. Factory methods let a class defer instantiation to subclasses.





# Creational Example - Factory

---

```
class PizzaFactory {
  static createPizza(type: 'DeepDish' | 'Tavern', toppings: string[]): IPizza {
    switch (type) {
      case 'DeepDish':
        return new DeepDishPizza(toppings, 'Seminole');
      case 'Tavern':
        return new TavernPizza(toppings, 'Large');
      default:
        throw new Error('Invalid pizza type...');
    }
  }
}

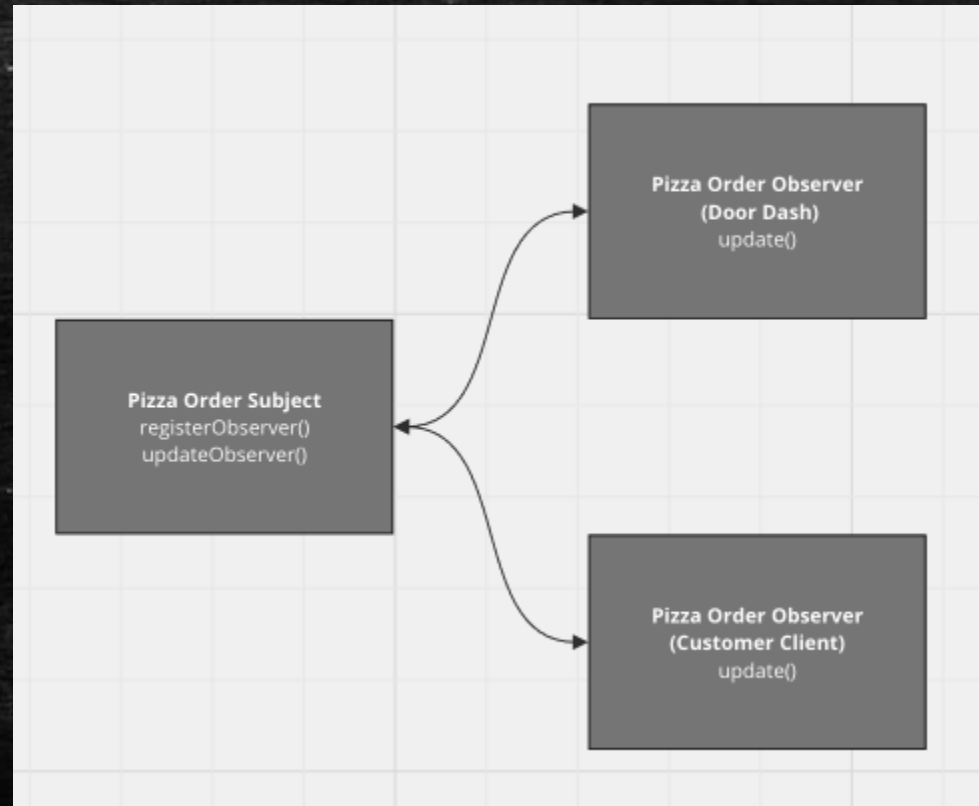
const myPizza: IPizza = PizzaFactory.createPizza('DeepDish', ['cheese', 'pepperoni']);
myPizza.preparePizza('Adam', true);
```





# Behavioral Example - Observer

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





# Behavioral Example - Observer

```
// concrete implementation of the subject.
class PizzaOrderSubject implements IPizzaOrderSubject {
  private observers: IPizzaOrderObserver[] = [];
  private pizzaState: IPizzaOrder = {
    pizza: PizzaFactory.createPizza('DeepDish', ['cheese', 'pepperoni']),
    orderDate: new Date(),
    status: 'Ordered'
  }
}

set orderStatus(status: 'Ordered' | 'Prepared' | 'Delivered') {
  this.pizzaState.status = status;
  this.pizzaState.deliveryDate = status === 'Delivered' ? new Date() : undefined;
  this.notifyObservers(this.pizzaState);
}

registerObserver(observer: IPizzaOrderObserver): void {
  this.observers.push(observer);
}

removeObserver(observer: IPizzaOrderObserver): void {
  const observerIndex = this.observers.indexOf(observer);
  if (observerIndex > -1) {
    this.observers.splice(observerIndex, 1);
  }
}

notifyObservers(order: IPizzaOrder): void {
  this.observers.forEach((observer: IPizzaOrderObserver) => observer.update(order));
}
}
```

```
// concrete implementation of the observer.
class PizzaOrderObserver implements IPizzaOrderObserver {
  constructor(public name: string, private subject: IPizzaOrderSubject) {
    this.subject.registerObserver(this);
  }

  update(order: IPizzaOrder): void {
    console.log(`Order updated detected- ${this.name}`, order);
  }
}
```





# Behavioral Example – Observer (cont).

---

```
// client code:

// create the subject.
const pizzaOrderStatusPublisher = new PizzaOrderSubject();

// create a couple of observers.
const deliveryProvider = new PizzaOrderObserver(`Door Dash Client`, pizzaOrderStatusPublisher);
const customer = new PizzaOrderObserver(`Customer Adam`, pizzaOrderStatusPublisher);

// change the state of the subject.
pizzaOrderStatusPublisher.orderStatus = 'Prepared';
    // Order updated detected- Customer Adam...
    // Order updated detected- Door Dash Client...

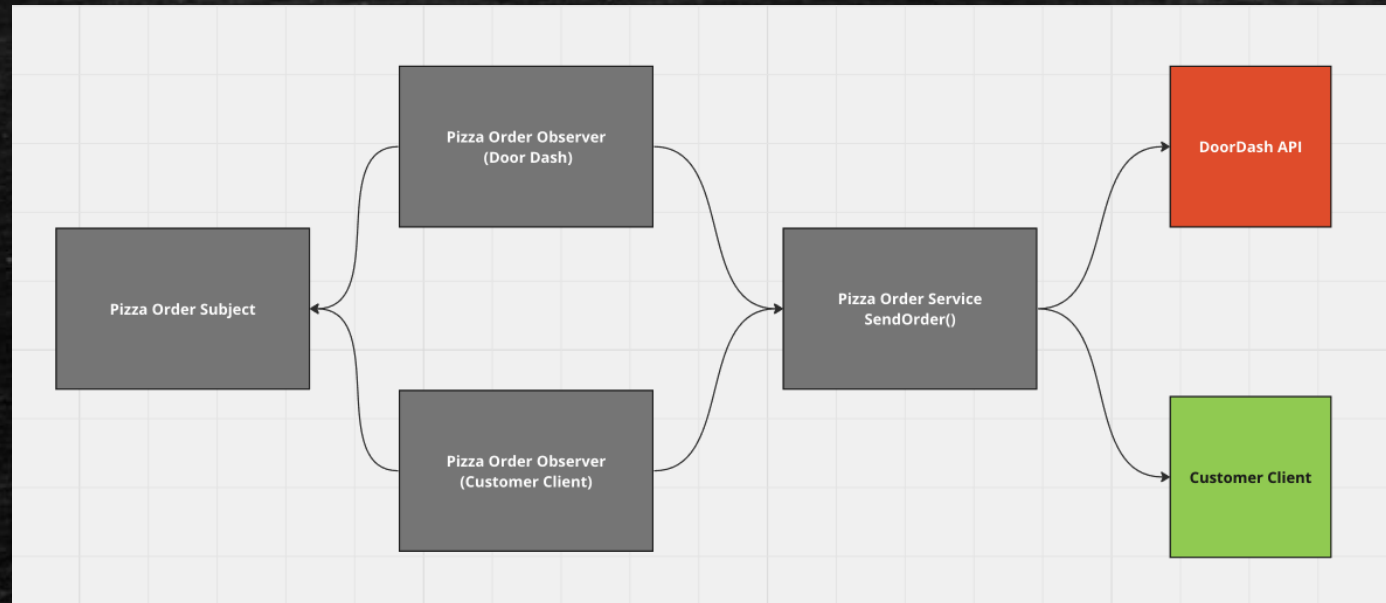
// do it again!
pizzaOrderStatusPublisher.orderStatus = 'Delivered';
    // Order updated detected- Customer Adam...
    // Order updated detected- Door Dash Client...
```

[Check out this complete working example in TS Playground!](#)



# Structural Example - Adapter

The Adapter Pattern converts the interface of a class into another interface the client expects. This allows classes to work together that otherwise wouldn't be able to due to incompatible interfaces.

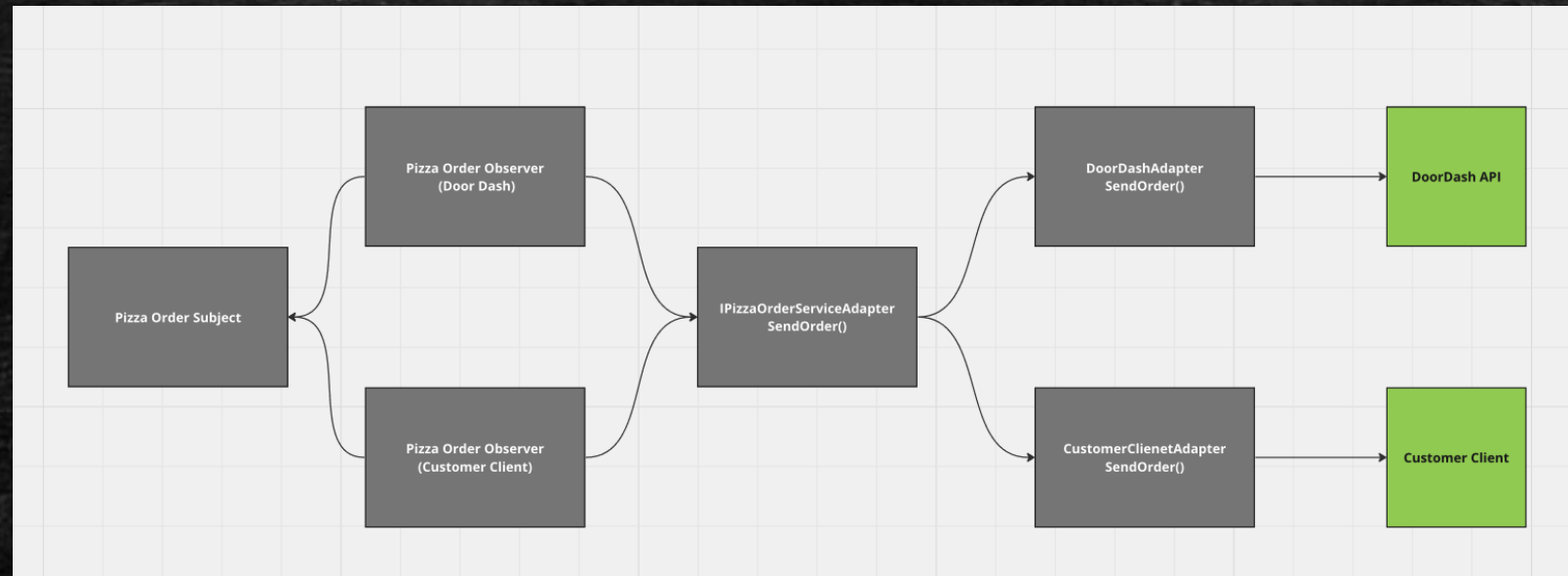






# Structural Example – Adapter (cont)

By instantiating concrete providers behind an abstract adapter interface, unique business logic can execute to "adapt" the request from the order flow to door dash.





# Structural Example – Adapter (cont)

```
interface IOrderServiceAdapter {  
    sendOrder(order: IPizzaOrder): void;  
}  
  
interface IDoorDashOrder {  
    item: any;  
    orderDate: Date;  
    pickupLocation: string;  
}  
  
interface ICustomerClientOrder extends IPizzaOrder {  
    customerLocation: string;  
}
```

```
class DoorDashServiceAdapter implements IOrderServiceAdapter {  
  
    mockDoorDashApiClient = {  
        getPickupLocation: (): string => '123 Fake St. Denver CO 80205'  
    }  
  
    sendOrder(order: IPizzaOrder): void {  
        const newOrder: IDoorDashOrder = {  
            item: order.pizza,  
            orderDate: order.orderDate,  
            pickupLocation: this.mockDoorDashApiClient.getPickupLocation()  
        };  
        console.log(`Order sent to provider client: `);  
        console.log(newOrder);  
    }  
}  
  
class CustomerClientServiceAdapter implements IOrderServiceAdapter {  
  
    mockCustomerClientApi = {  
        getCustomerLocation: (): string => '2350 Tremont Pl Denver CO 80205'  
    }  
  
    sendOrder(order: IPizzaOrder): void {  
        const newOrder: ICustomerClientOrder = {  
            ...order,  
            customerLocation: this.mockCustomerClientApi.getCustomerLocation()  
        };  
        console.log(`Order sent to customer client: `);  
        console.log(newOrder);  
    }  
}
```





# Summary (Final Thoughts)

---

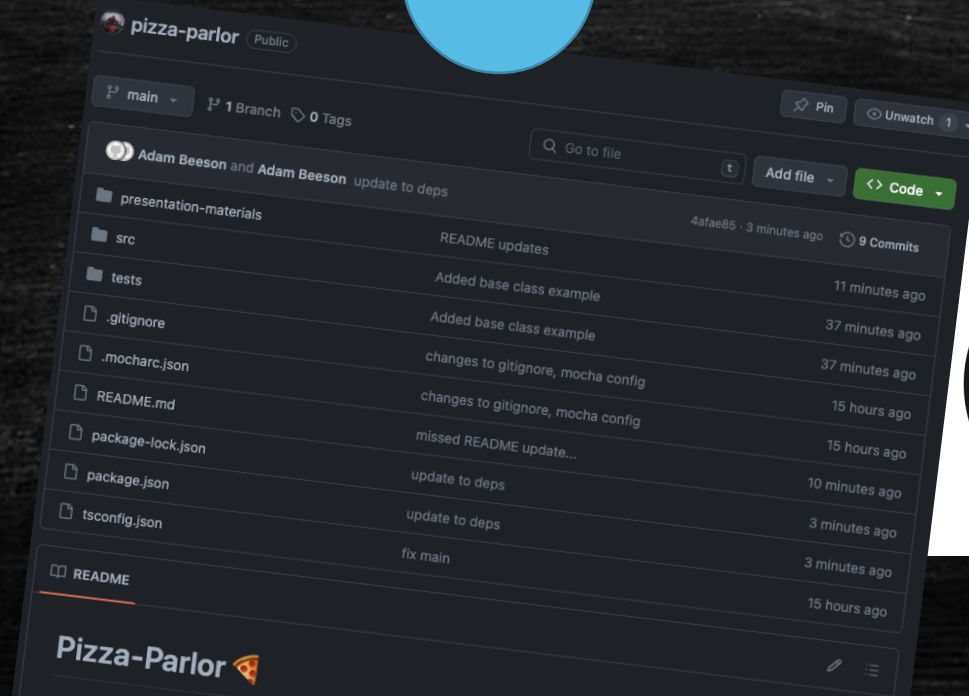
- Object Oriented Code allows for us as humans to easily understand code (readability, modeling, etc.).
- Things like Dependency Inversion and loosely coupling code with abstractions makes things like Unit Testing or extensibility really, really easy. Like, really easy.
- Patterns can be easily referred to and implemented for code health and complexity.
- Objects are representational of state and behavior of a thing by nature, making them excellent models for stateful code.
- Where code shouldn't be concerned with or burdened by the state of an application, or functional immutability is important, objects can work with and be composed with functional code.
- Object Oriented code can be needlessly complex for smaller or well-scoped applications.
  - Beyond complexity comes the dangers of unmanageable code when objects don't follow best practices or patterns.
- Object Oriented code can occupy more space in memory than their stateless, functional, counterpart applications.





# Code Example

Check out the working code for this presentation:  
[github.com/beeson/pizza-parlor](https://github.com/beeson/pizza-parlor)







?