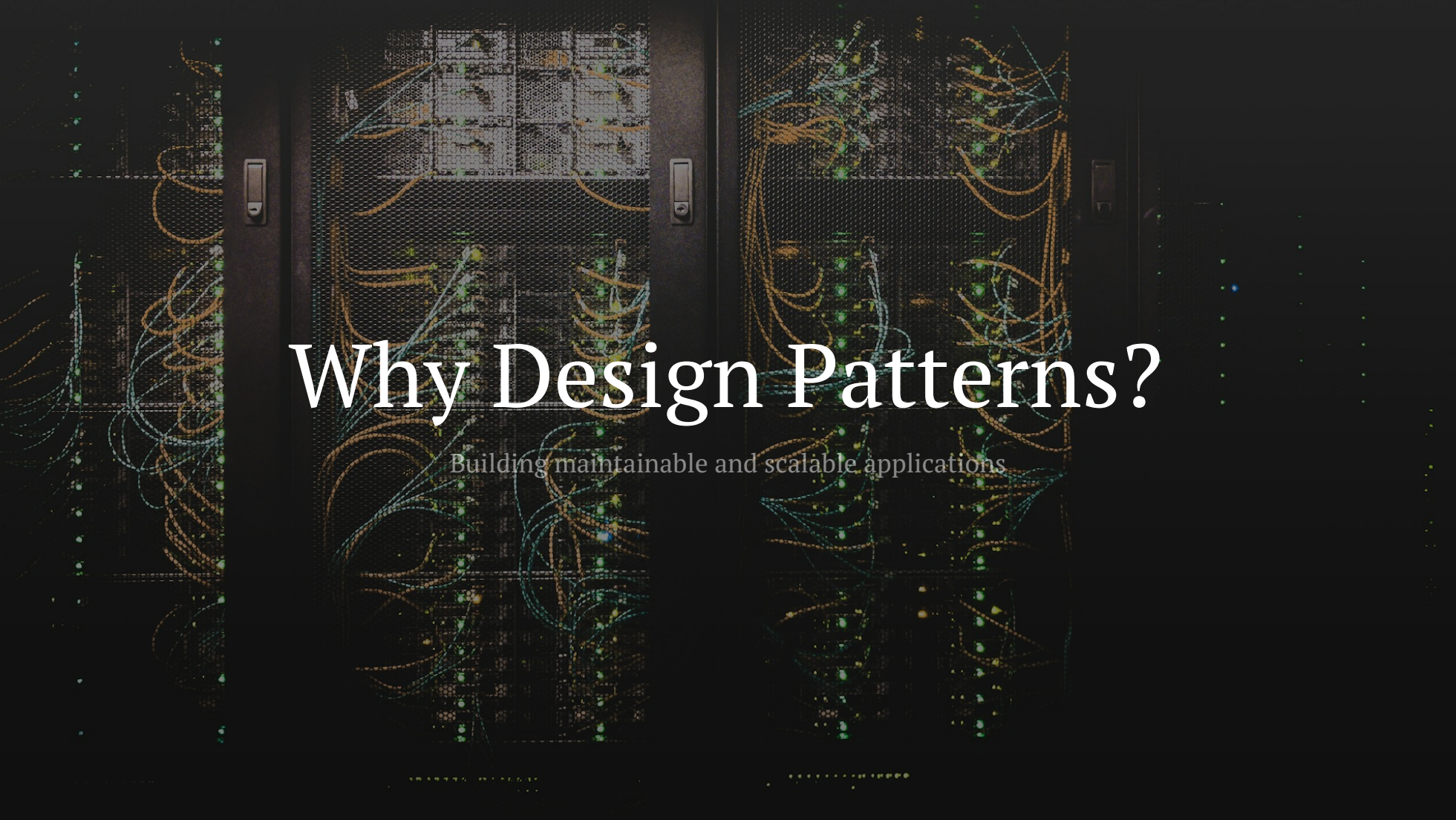
Advanced Concepts

Taking your skills to the next level

Press Space for next page →

Table of Contents



Why Design Patterns?

Building maintainable and scalable applications

The Challenge

As applications grow, developers face:

- **Complexity**: Code becomes hard to understand
- **Duplication**: Same solutions reimplemented
- **Maintenance**: Changes ripple through the codebase
- **Onboarding**: New team members struggle

The Solution: Patterns

Without Patterns

```
// Scattered, inconsistent code
class UserManager {
  private users = [];

  add(user) { /* ... */ }
  remove(user) { /* ... */ }
  notify(message) { /* ... */ }
}
```

With Patterns

```
// Clear, consistent structure
class UserRepository {
  private store: Store<User>;

  save(user: User) { /* ... */ }
  delete(id: string) { /* ... */ }
}
```

💡 **Key Insight:** Patterns provide a shared vocabulary and proven solutions.

What are Design Patterns?

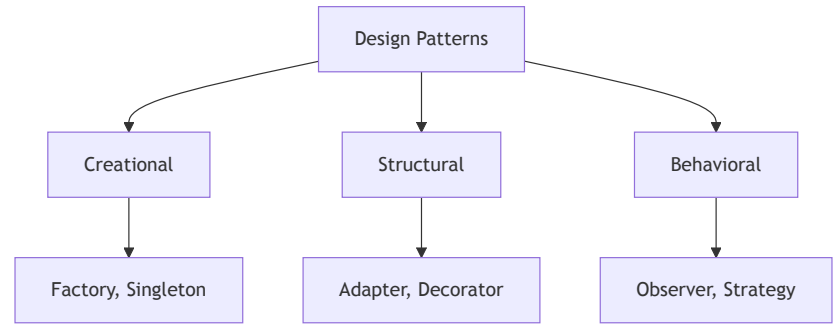
Core concepts and categories

Definition

Design Patterns Are

- **Reusable solutions** to common problems
- **Templates**, not finished code
- **Best practices** distilled from experience
- **Communication tools** for teams

Pattern Categories



Common Patterns Overview

Pattern	Category	Purpose
Singleton	Creational	Ensure single instance
Factory	Creational	Create objects without specifying class
Observer	Behavioral	Notify dependents of state changes
Strategy	Behavioral	Encapsulate interchangeable algorithms
Decorator	Structural	Add behavior dynamically

✅ **Pro Tip:** Start with patterns you encounter most often in your framework.

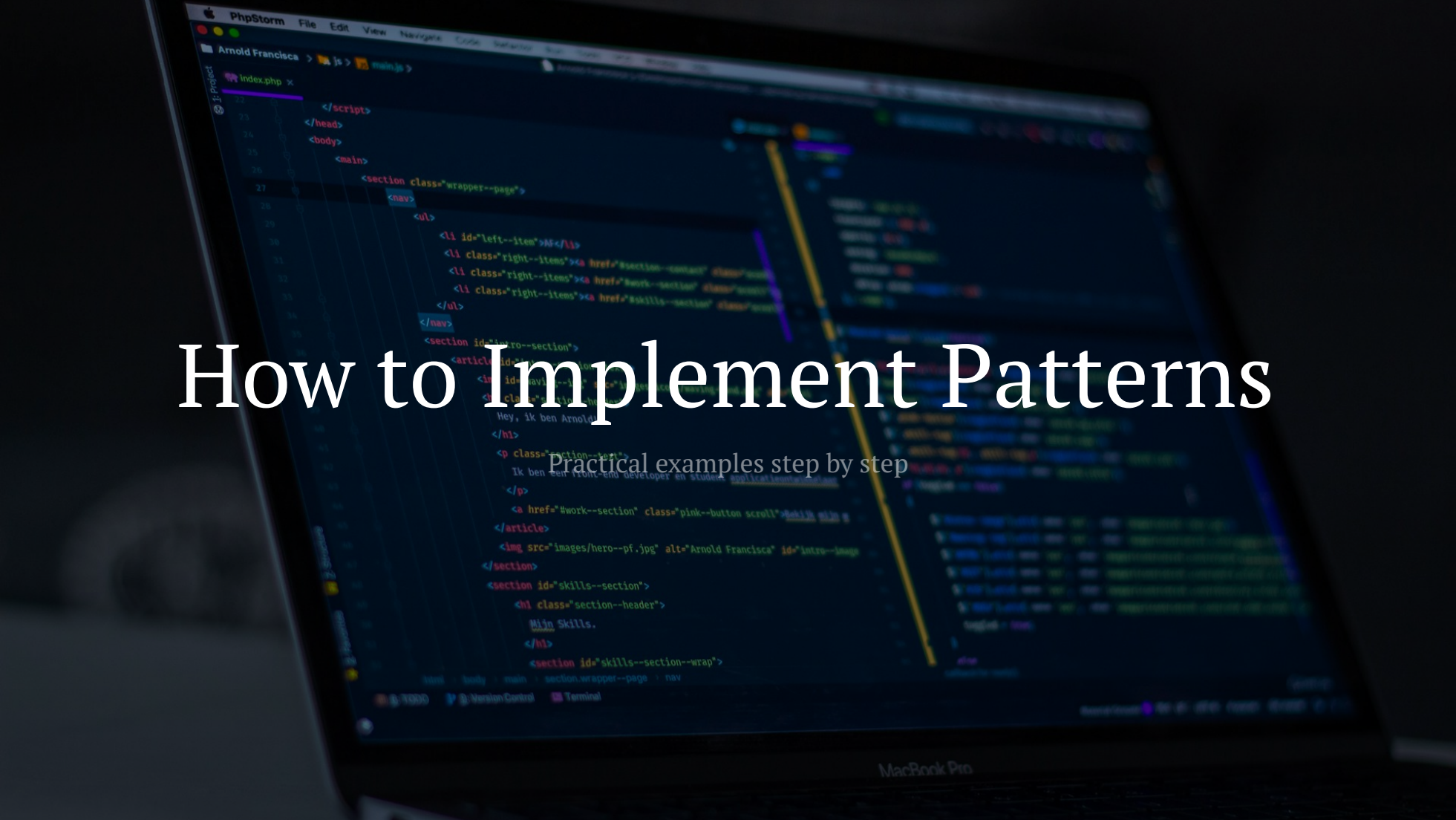
Pattern Anatomy

```
// 1. Intent: What problem does it solve?
// The Observer pattern defines a one-to-many dependency
// between objects so that when one object changes state,
// all its dependents are notified.

// 2. Structure: How is it organized?
interface Observer {
  update(data: any): void;
}

// 3. Implementation: How do you use it?
class Subject {
  private observers: Observer[] = [];

  subscribe(observer: Observer) { /* ... */ }
  notify(data: any) { /* ... */ }
}
```



How to Implement Patterns

Practical examples step by step

Observer Pattern: Step 1

Define the interfaces:

```
// Observer interface
interface Observer<T> {
  update(data: T): void;
}

// Subject interface
interface Subject<T> {
  subscribe(observer: Observer<T>): void;
  unsubscribe(observer: Observer<T>): void;
  notify(data: T): void;
}
```

Observer Pattern: Step 2

Implement the Subject:

```
class EventEmitter<T> implements Subject<T> {
  private observers: Set<Observer<T>> = new Set();

  subscribe(observer: Observer<T>): void {
    this.observers.add(observer);
  }

  unsubscribe(observer: Observer<T>): void {
    this.observers.delete(observer);
  }

  notify(data: T): void {
    this.observers.forEach(obs => obs.update(data));
  }
}
```

Step 3: Usage

```
// Create subject
const userEvents = new EventEmitter<User>();

// Create observers
const logger: Observer<User> = {
  update: (user) => {
    console.log('User changed:', user);
  }
};

const analytics: Observer<User> = {
  update: (user) => {
    trackEvent('user_update', user);
  }
};

// Subscribe
userEvents.subscribe(logger);
userEvents.subscribe(analytics);

// Trigger
userEvents.notify(currentUser);
```

Benefits

- ✅ Loose coupling
- ✅ Easy to add new observers
- ✅ Single responsibility
- ✅ Open for extension

Use Cases

- Event systems
- State management
- Real-time updates
- Pub/sub messaging

What If...?

Advanced scenarios and alternatives

Combining Patterns

Observer + Strategy

```
interface NotificationStrategy {
  send(message: string): void;
}

class NotificationService {
  private strategy: NotificationStrategy;
  private observers: Observer[] = [];

  setStrategy(s: NotificationStrategy) {
    this.strategy = s;
  }

  notify(message: string) {
    this.strategy.send(message);
    this.observers.forEach(o =>
      o.update(message)
    );
  }
}
```

When to Combine

- Complex notification systems
- Flexible processing pipelines
- Plugin architectures

⚠️ **Warning:** Don't over-engineer. Add patterns when needed.

Anti-Patterns to Avoid

Anti-Pattern	Problem	Solution
God Object	One class does everything	Split responsibilities
Spaghetti Code	No clear structure	Apply appropriate patterns
Golden Hammer	Using one pattern everywhere	Choose pattern for the problem
Premature Optimization	Patterns before need	Start simple, refactor later

🚫 **Remember:** Patterns solve problems. No problem = no pattern needed.

Modern Alternatives

🔄 RxJS

Reactive extensions replace many behavioral patterns

📦 Dependency Injection

Framework-level solution for creational patterns

🎯 Functional Programming

Composition over inheritance

```
// Modern approach: Functional composition
const processUser = pipe(
  validate,
  transform,
  save,
  notify
);

await processUser(userData);
```

Exercise: Design Patterns

Apply what you've learned!

Your Task

Objectives

1. **Implement** the Observer pattern
2. **Create** multiple observers
3. **Test** the notification flow
4. **Extend** with unsubscribe logic

Time

⏱️ **20 minutes**

Starting Point

```
// Implement this interface
interface Subject<T> {
  subscribe(obs: Observer<T>): void;
  unsubscribe(obs: Observer<T>): void;
  notify(data: T): void;
}

// Your implementation
class Store<T> implements Subject<T> {
  // TODO: Implement
}
```

Requirements

Must Have

- ☐ `subscribe()` adds observers
- ☐ `unsubscribe()` removes observers
- ☐ `notify()` calls all observers
- ☐ Observers receive correct data

Bonus

- ☐ Prevent duplicate subscriptions
- ☐ Add `once()` for single notification
- ☐ Type-safe with generics
- ☐ Memory leak prevention

💡 **Hint:** Use a `Set` instead of an array for automatic deduplication.

Start Coding!

Implement the Observer pattern in your project

Solution: `git checkout solution-observer-pattern`

Thank You!

Questions?