

# DESIGN PATTERNS IN THE REAL WORLD

---



Zhuwei Zhang  
@zhuwei

Never spend 6 minutes doing something by hand when you can spend 6 hours failing to automate it





# ABOUT ME

---

## NAME

Courtney Cooksey

[courtney@thecrossingchurch.com](mailto:courtney@thecrossingchurch.com)

## OCCUPATION

Software Engineer at The Crossing EPC

## CURRENTLY BASED IN

Savannah, MO

# AGENDA

---

— Terminology & the Origin of Design Patterns —

## 1 | **What Qualifies as a Design Pattern**

## 2 | **A Few Design Patterns**

## 3 | **Why Design Patterns are Useful**

— Practical Application —

## 4 | **Using Design Patterns**

## 5 | **Anti-Patterns**

— Beyond Today —

## 6 | **Pattern Catalogs & Categories**

## 7 | **Resources**

Terminology & the Origin of Design Patterns

# WHAT QUALIFIES AS A DESIGN PATTERN

# A DESIGN PATTERN IS A SOLUTION TO A PROBLEM IN A CONTEXT.

---

The problem consists of forces

- **Goal**  
What we are trying to achieve.
- **Constraints**  
Restrictions on achieving our goal.

A solution that properly balances the goal with the constraints is a useful pattern.

# PATTERNS PROVIDE BLUEPRINTS, NOT CODE

---

The origin of patterns began, not with a software developer, but with an architect.

1977, Christopher Alexander and Co.

Coined the term *Pattern Language* to provide a new way of talking about problems in architecture and solutions to those problems.



# A PATTERN LANGUAGE EXAMPLE: THE ENTRANCE ROOM (130)

---

Arriving in a building, or leaving it, you need a room to pass through. This is the entrance room.

*The subtleties of saying goodbye.* When hosts and guests are saying goodbye, the lack of a clearly marked "goodbye" point can easily lead to an endless "Well, we really must be going now," and then further conversations and lingering on.

At the main entrance to a building, make a light-filled room which marks the entrance and straddles the boundary between indoors and outdoors.

See Also: FAMILY OF ENTRANCES (102), MAIN ENTRANCE (110), ENTRANCE TRANSITION (112), CAR CONNECTION (113), PRIVATE TERRACE ON THE STREET (140).

# DESIGN PATTERNS NEED TO BE PROPERLY DOCUMENTED

---

- **Name** NAME, CLASSIFICATION, ALSO KNOWN AS

We can't talk about a design pattern if we don't have a unique name for it.

The Entrance Room

- **Description** INTENT, MOTIVATION, APPLICABILITY

We need to know the intent or definition of the pattern, the motivation for using it, how it solves the given problem, and the situations it applies to.

The subtleties of saying goodbye

STRUCTURE, PARTICIPANTS, COLLABORATIONS, CONSEQUENCES,

- **Technical Details** IMPLEMENTATIONS, SAMPLE CODE, KNOWN USES

Class diagrams, information about the objects and classes needed, how they interact, known issues to watch out for, and some code samples to get started.

Diagrams I didn't include

- **Related Patterns** RELATED PATTERNS

The relationship between this pattern and other patterns.

Private Terrace on the Street

# DESIGN PATTERNS SHOULD BE REUSABLE

---

A solution is not truly a design pattern unless it is applicable in multiple situations.

## The Rule of Three

---

A pattern can only be called a pattern if it has been applied in a real-world solution at least three times.

# PATTERNS IN ACTION

---



## Problem

How do I get to work on time?



## Context

I locked my keys in my car.



## Solution

Break the window.  
Let myself into my car.  
Start engine.  
Drive to work.

Why isn't this a pattern?

# QUALIFICATIONS FOR A DESIGN PATTERN

---

- **Solution to a Problem in a Context**

The pattern must balance the goal with the constraints to be considered useful.

- **Reusability and Flexibility**

A pattern isn't a pattern if it can't be applied to multiple unique situations.

- **Documentation**

Our pattern needs a name and definition, something we can easily hand to another person so they can implement it for their unique situation.

- **Community Acceptance**

There needs to be at least **three** examples of this pattern being used in the real world.

Technical Details & Practical Information

# A FEW DESIGN PATTERNS

# PIZZAS WITH PIZZAZ

---

A new pizza shop is opening up and we're going to write their software

## REQUIREMENTS

---

- **Menu**

Customers need to know what they can order

- **Pizza Prep**

The chef needs to know how to make each pizza

- **Bake**

We can't serve a raw pizza



```
public class PizzaStore {  
    private Dictionary<string, string> _menu { get; set; }  
    public void initializeStore() {  
        _menu = new Dictionary<string, string>();  
        _menu.Add("Cheese", "Dough, Sauce, Cheese");  
    }  
    public Dictionary<string, string> getMenu() {  
        return _menu;  
    }  
    public IPizza order(string selection) {  
        IPizza _pizza;  
        if(selection == "Cheese") {  
            _pizza = new CheesePizza();  
        } else {  
            throw OrderNotFoundException("Sorry, we don't serve " + selection + " pizzas.");  
        }  
        _pizza.prep(); _pizza.bake(); _pizza.cut();  
        return _pizza;  
    }  
}
```

```
public interface IPizza {  
    public string name;  
    private List<string> _ingredients;  
  
    public void prep();  
    public void bake();  
    public void cut() {  
        Console.WriteLine("Cutting Pizza...");  
    }  
}
```

```
public class CheesePizza: IPizza {
    public string name = "Cheese";
    private List<string> _ingredients = new List<string>() { "Dough", "Sauce", "Cheese" };

    public void prep() {
        rollOutDough();
        addSauce();
        addCheese();
    }

    public void bake() {
        int time = 23; int temp = 350;
        while(time > 0) {
            Console.WriteLine("Baking at " + temp.ToString());
            time--;
        }
    }
}
```

# NEW REQUIREMENT: INGREDIENT UPDATE

---

Cheese isn't a great description, we need to specify that our pizza is made with Mozzarella

```
public class CheesePizza: IPizza {  
    private List<string> _ingredients = new List<string>() { "Dough", "Sauce", "Mozzarella"  
};  
}
```

```
public class PizzaStore {  
    public void initialize() {  
        _menu.Add("Cheese", "Dough, Sauce, Mozzarella");  
    }  
}
```

We're also going to introduce a new menu item, Pepperoni Pizza!

```
public class PizzaStore {  
    public void initializeStore() {  
        _menu = new List<IPizza>() { new CheesePizza(), new PepperoniPizza() };  
    }  
    public Dictionary<string, string> getMenu() {  
        Dictionary<string, string> menu = new Dictionary<string, string>();  
        foreach(var pizza in _menu) { menu.Add(pizza.name, pizza.getInfo()); }  
        return menu;  
    }  
    public IPizza order(string selection) {  
        IPizza _pizza;  
        if(selection == "Cheese") {  
            _pizza = new CheesePizza();  
        } else if(selection == "Pepperoni") {  
            _pizza = new PepperoniPizza();  
        } else {  
            throw OrderNotFoundException("Sorry, we don't serve " + selection + " pizzas.");  
        }  
        _pizza.prep(); _pizza.bake(); _pizza.cut(); return _pizza;  
    }  
}
```

```
public interface IPizza {  
    public string name;  
    private List<string> _ingredients;  
  
    public string getInfo() {  
        return String.Join(", ", _ingredients);  
    }  
  
    public void prep();  
    public void bake();  
    public void cut() {  
        Console.WriteLine("Cutting Pizza...");  
    }  
}
```

```
public class PepperoniPizza: IPizza {
    public string name = "Pepperoni";
    private List<string> _ingredients = new List<string>() { "Dough", "Sauce", "Cheese", "Pepperoni" };
}

public prep() {
    rollOutDough();
    addSauce();
    addCheese();
    addMeat();
}

public bake() {
    int time = 27; int temp = 375;
    while(time > 0) {
        Console.WriteLine("Baking at " + temp.ToString());
        time--;
    }
}
```

# NEW REQUIREMENT: CRAZY CUPCAKES

---

The owner's sister is opening a bakery. She doesn't have enough start up money to build her own software so we're going to use the pizza store software we already have.



```
public interface IStore {
    private List<IProduct> _menu { get; set; }
    public void initializeStore();

    public Dictionary<string, string> getMenu() {
        Dictionary<string, string> menu = new Dictionary<string, string>();
        foreach(IProduct product in _menu) {
            menu.Add(product.name, product.getInfo());
        }
        return menu;
    }

    public IProduct order(string selection) {
        IProduct item;
        item = createMeal(selection);
        item.make();
        item.serve();
        return item;
    }
    private IProduct createMeal(string selection);
}
```

```
public interface IProduct {  
    public string name;  
    private List<string> _ingredients;  
  
    public string getInfo() {  
        return String.Join(", ", _ingredients);  
    }  
  
    public void make();  
    public void serve();  
}
```

```
public class PizzaStore: IStore {
    public void initializeStore() {
        _menu = new List<IProduct>(){ new CheesePizza(), new PepperoniPizza(), new SideSalad() };
    }

    private IProduct createMeal(string selection) {
        IProduct _item;
        if(selection == "Cheese") {
            _item = new CheesePizza();
        } else if(selection == "Pepperoni") {
            _item = new PepperoniPizza();
        } else if(selection == "Salad") {
            _item = new SideSalad();
        } else {
            throw OrderNotFoundException("Sorry, we don't serve " + selection);
        }
        return _item;
    }
}
```

```
public class CupcakeStore: IStore {  
    public void initializeStore() {  
        _menu = new List<IProduct>(){ new VanillaCupcake(), new MiniCheesecake() };  
    }  
  
    private IProduct createMeal(string selection) {  
        IProduct _item;  
  
        if(selection == "Vanilla") {  
            _item = new VanillaCupcake();  
        } else if(selection == "MiniCheesecake") {  
            _item = new MiniCheesecake();  
        } else {  
            throw OrderNotFoundException("Sorry, we don't serve " + selection);  
        }  
  
        return _item;  
    }  
}
```

# INTRODUCING THE FACTORY METHOD PATTERN

---

By encapsulating the creation of our pizza and cupcake objects we've implemented the **Factory Method Pattern**.

```
public IProduct order(string selection) {  
    IProduct item;  
    item = createMeal(selection);  
    item.make();  
    item.serve();  
    return item;  
}
```

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

# NEW REQUIREMENT: RESTAURANT SEATING

---

Our pizza store is doing great! The owner is opening a formal dine-in Italian restaurant. With this comes new requirements.

## REQUIREMENTS

---

- **Seating**

The hostess needs to know what tables are available for seating customers and the waitstaff need a way to bus tables.



```
public class ItalianRestaurant: IStore {
    private Dictionary<string, bool> _tables = new Dictionary<string, bool> {}; //Table Number, Is Occupied?
    public void initializeStore() {
        _menu = new List<IProduct>(){ new TortelliniAlfredo(), new GarlicBread(), new Risotto() };
        _tables.Add("One", false); _tables.Add("Two", false);
    }
    private IProduct createMeal(string selection) { /*We already know how this method works */ }
    public void seat(string tableNumber) {
        bool tableIsOccupied;
        if( _tables.TryGetValue(tableNumber, out tableIsOccupied) ) {
            if(!tableIsOccupied) {
                _tables[tableNumber] = true; //Our new customer is now occupying the table
            }
            throw TableIsOccupiedException(tableNumber + " is already occupied!");
        }
        throw TableNotFoundException(tableNumber + " is not a valid table.");
    }
    public void bus(string tableNumber) {
        _tables[tableNumber] = false;
    }
}
```

# NEW REQUIREMENT: TABLE IS READY NOTIFICATION

---

Our Italian restaurant is blowing up! The system for seating isn't very efficient though.

To get people seated faster we want to notify all our hosts as soon as a table is ready.



```
public interface IStore {
    private List<ISubscriber> _subscribers;

    public void subscribe(ISubscriber subscriber) {
        _subscribers.Add(subscriber);
    }

    public void unsubscribe(ISubscriber subscriber) {
        _subscribers.Remove(subscriber);
    }

    private void notify(string tableNumber) {
        foreach (ISubscriber subscriber in _subscribers) {
            subscriber.update(tableNumber);
        }
    }
}
```

```
public interface ISubscriber {  
    public void update(string tableNumber) {  
        Console.WriteLine(tableNumber + " is ready!");  
    }  
}
```

```
public class ItalianRestaurant: IStore {  
    public void bus(string tableNumber) {  
        _tables[tableNumber] = false;  
        notify(tableNumber);  
    }  
}
```

# INTRODUCING THE OBSERVER PATTERN

---

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.

```
public interface IStore {  
    public void subscribe;  
    public void unsubscribe;  
    private void notify(string tableNumber);  
}
```

```
public interface ISubscriber {  
    public void update(string tableNumber);  
}
```

You may have heard the Observer Pattern referred to as the "Pub/Sub Model" or an "Event Listener"

# HOW ARE WE ORDERING OUR INGREDIENTS ANYWAYS?

---

```
public interface IIngredient {  
    public string name;  
    public IngredientCategory category;  
    private int _quantity;  
    private ISupplier _supplier;  
    public void useIngredient() { _quantity--; }  
    public void stockIngredient(int count) {  
        if (_supplier.purchase(name, category)) {  
            _quantity += count;  
        }  
    }  
    public int getQuantityInStock() { return _quantity; }  
}  
public enum IngredientCategory { Vegetable, Fruit, Protein, Oil, Dairy, Grain }
```

# NEW REQUIREMENT: SWITCHING SUPPLIERS

---

Our restaurants are doing great but one of our suppliers is going under.

We have to use a new supplier and unfortunately they don't implement the `ISupplier` interface our other suppliers do.

Their ingredient categories are different than our other suppliers and their purchase method returns an `http response` instead of a boolean.



# OPTIONS TO HANDLE THE NEW SUPPLIER

---



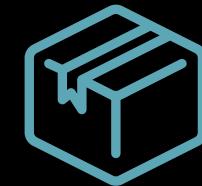
## Add Conditional Checks

We could add some if statements to our code to check for an instance of the new supplier and handle it differently.



## Override Purchase

We could add overrides for the purchase method but we'd still need to rewrite our ingredient interface to handle it.



## Wrap the New Supplier

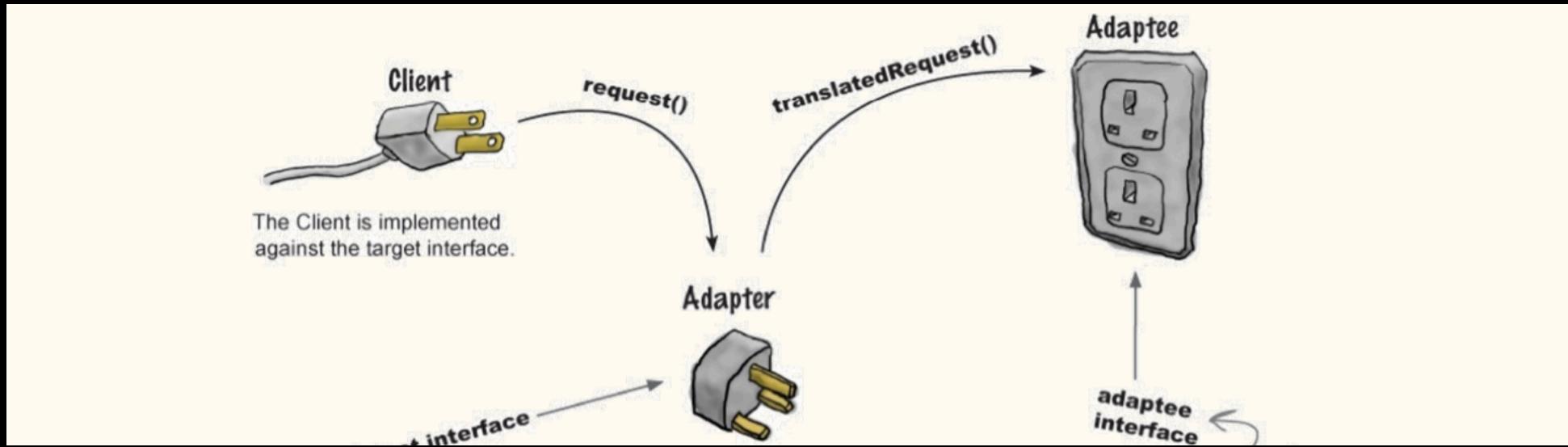
This supplier is the odd one out in our experience and they have a reputation for constantly rewriting their API.

```
public class NewSupplierWrapper: ISupplier {  
    private CustomSupplier _supplier;  
  
    public bool purchase(string name, IngredientCategory category) {  
        CustomIngredientCategory supplierCategory = findCategory(category);  
        HttpResponseMessage response = _supplier.purchase(name, supplierCategory);  
        if(response.StatusCode == 200) {  
            return true;  
        }  
        return false;  
    }  
    private CustomIngredientCategory findCategory(IngredientCategory category) {  
        //Code to map and convert the categories  
    }  
}  
public enum CustomIngredientCategory { Vegetable, Fruit, Meat, Tofu, Oil, Dairy, VeganDairy, Fat,  
    ... }
```

**With this solution we don't have to worry about making changes to our core codebase to support any changes the new supplier makes.**

# INTRODUCING THE ADAPTER PATTERN

By wrapping the new supplier in a class that allows it to implement our existing supplier interface we have used the Adapter Pattern.



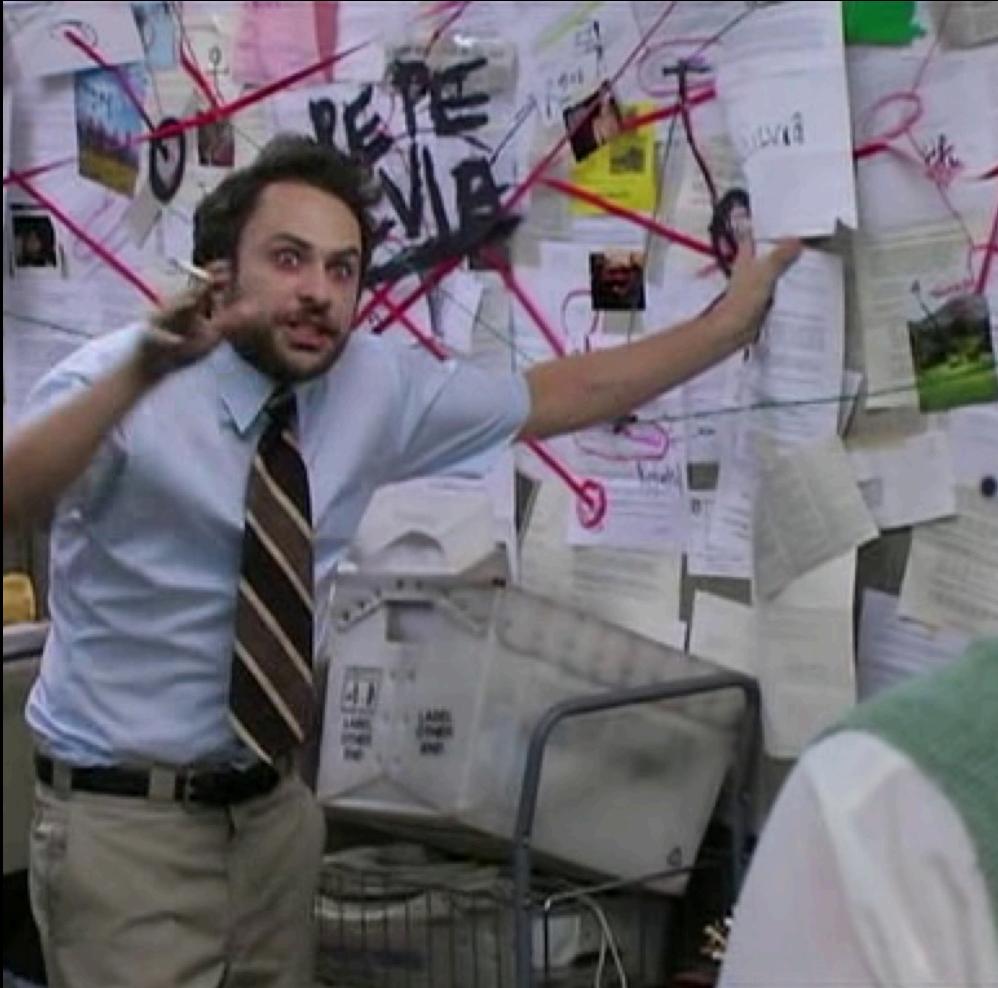
The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Terminology & the Origin of Design Patterns

# WHY DESIGN PATTERNS ARE USEFUL

# DESIGN PATTERNS PROVIDE A SHARED VOCABULARY

---



Communication struggles are the biggest time wasters in any project.

# PATTERNS ALLOW YOUR TEAM TO KEEP DESIGN DISCUSSIONS AT A CONCEPTUAL LEVEL

---

When discussing how to design a new system it is easy to get lost in the details of the specific implementation if we don't have common terminology for talking about the overarching design.



# DESIGN PATTERNS MAKE IT EASY TO REUSE OUR EXISTING CODE

---



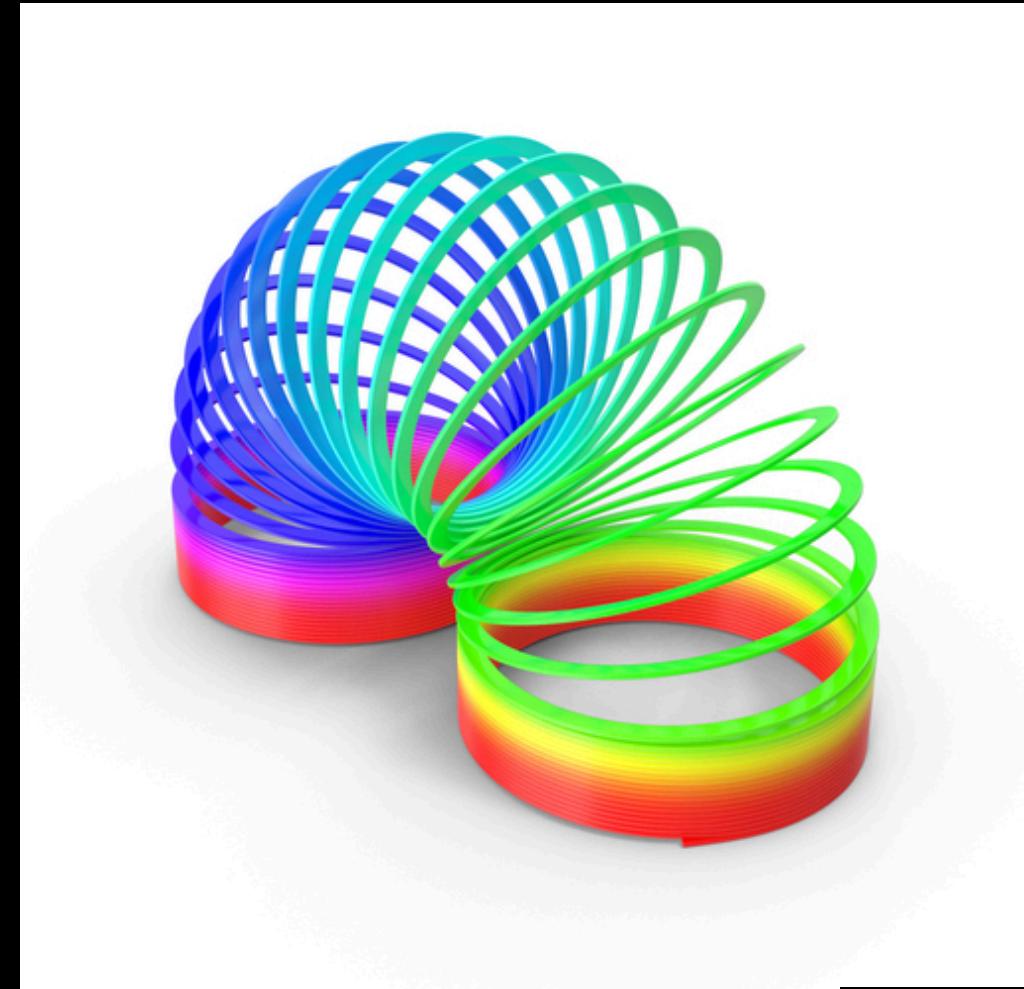
Any project we don't have to start from scratch is a good project.

We were able to create new menu items and stores without needing to rewrite the basic store and product operations.

# DESIGN PATTERNS KEEP CODE FLEXIBLE

---

Crazy Cupcakes doesn't prepare their cupcakes the same way Pizzas with Pizzaz prepares their pizzas. Since we encapsulated their product's preparation methods it wasn't a problem to add unique items.



# SYSTEMS BUILT USING GOOD DESIGN PATTERNS ARE GENERALLY EASY TO MAINTAIN

---

Debugging is like  
being the detective in a  
crime movie where you're  
also the murderer.

Design Patterns make it easier for other developers (or yourself months later) to read and understand your code.

They are also flexible enough that we can extend systems without rewriting them.

- Filipe Fortes -

# DESIGN PATTERNS HAVE THE BENEFIT OF BEING VERIFIED BY THE COMMUNITY

---

Patterns only become patterns when they have been used in multiple, unique, real-world systems.



# SOFTWARE ARCHITECT IS A DISTINCT JOB FOR A REASON

---

```
public interface IAnimal {  
    string _name;  
    string _sound;  
    void Walk();  
    void Fly();  
    string MakeSound();  
}
```

**Knowing object oriented principals doesn't automatically make you a good designer.**

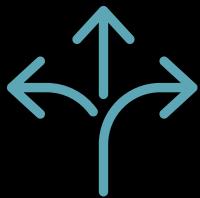
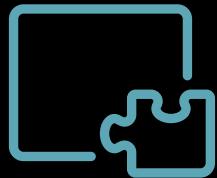
**We can use encapsulation and inheritance and still end up with bad code.**

Technical Details & Practical Information

# USING DESIGN PATTERNS

# WHEN TO USE A DESIGN PATTERN

When the need arises...



## When it would address a problem in your system

Hosts need to know when a table is ready for new customers.

## When you expect aspects of your system to vary

We know all restaurants prepare their products differently. If we expect to reuse our pizza store code we should use a flexible design pattern.

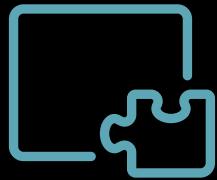
Are we building the first phase of a new enterprise system?  
Experimenting with a new vendor?

## When you find yourself with a massive if statement

Not a catch all, but if you have a function with 25 if/else statements there is a chance it could be refactored.

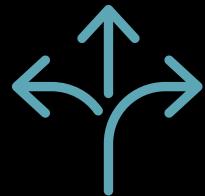
# WHEN NOT TO USE A DESIGN PATTERN

There are a lot of times we don't need a design pattern.



**When there isn't a practical reason we need to support change or variation**

While we want to design with future variation in mind, we shouldn't add unnecessary complexity based on hypothetical future changes.



**Overusing patterns can make code unnecessarily complex and inefficient**

If we have to wade through 13 interfaces and create 12 additional objects in memory just to perform one action, our design is probably over-engineered.



**When you're writing something for single use**

About to migrate ChMS systems? A one-time data conversion from one specific format to another probably doesn't require a design pattern.

# THE MOST IMPORTANT PRINCIPAL OF DESIGN IS SIMPLICITY

---

The goal of our software designs should always be to write the most simple solution that solves the problem.

**The simplest solution to a problem often involves a design pattern, but not always.**

Don't need a design pattern right now?

Don't write it.

Don't need to support the change you expected?

Remove the design pattern you originally included.

Using design patterns doesn't automatically make you a good developer.

Not using them doesn't make you a bad developer.

# FACTORING DESIGN PATTERNS INTO YOUR DEVELOPMENT PROCESS

---

## Natural Recognition Comes With Experience

It will take time to be able to naturally identify when a pattern would fit into your design

## Keep it Simple

The simplest solution is always the best, your design should always be focused on simplicity and following good OO practices, not trying to use a pattern

## Refactor Time is Pattern Time

Are you making a change to an application?  
Could this refactor have been avoided if our system used a design pattern?  
Time to evaluate the current design

## Look at Patterns When Designing

Brush up on your patterns  
Learn about new patterns that might be helpful in your design  
Don't force a pattern based solution

Technical Details & Practical Information

# ANTI-PATTERNS

# DEFINITION OF AN ANTI-PATTERN

---

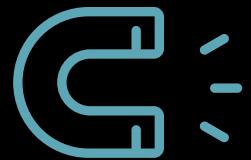


An Anti-Pattern is a **BAD solution** to a **problem** in a **context**. An Anti-Pattern has:

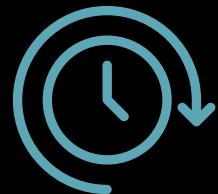
- A **name we can use to talk about it**
- A **description of the problem, context, and forces**
- **The supposed solution**
- **The refactored solution**
- **Real-world examples**

# WHY ANTI-PATTERNS ARE USEFUL

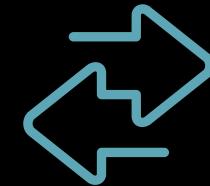
---



**Tell you why a bad  
solution is attractive**



**Tell you why a bad  
solution won't work  
long-term**



**Suggest other  
applicable patterns  
that may provide  
good solution**

# ANTI-PATTERNS IN ACTION: THE GOLDEN HAMMER

---

- **Problem**

You need to choose technologies for new development and you believe one technology must dominate the architecture

- **Context**

The new system/software that needs to be developed doesn't fit well with the technology the team is familiar with

- **Forces**

The team is committed to the technology they know

The team is not familiar with other technologies

Unfamiliar technologies are seen as risky

It is easy to plan and estimate development with familiar technology

- **Supposed Solution**

Use the familiar technology anyways, apply it obsessively to many problems, even if it is clearly inappropriate

- **Refactored Solution**

Expand the knowledge of the team through education, training, and book studies that expose them to new solutions

- **Examples**

Web companies that use their own internal caching systems instead of an open source alternative

Ministries with Excel workbooks?

Software dev learns about observer patterns wants to use it everywhere lol

Beyond Today

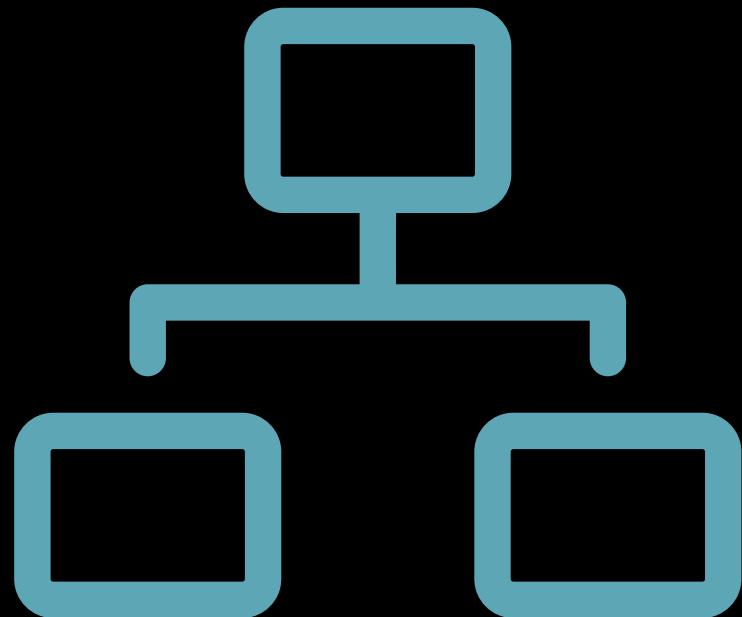
# PATTERN CATALOGS & CATEGORIES

# A PATTERN CATALOG IS AN ORGANIZED COLLECTION OF PATTERNS

---

Pattern Catalogs are useful references for designing software because they provide a definition of a number of patterns and the relationships between them.

Pattern Catalogs organize patterns into categories to help you find the one you need.



# PURPOSE-BASED PATTERN CATEGORIES

---

## Creational

Involve object instantiation

Abstract Factory  
Factory Method  
Singleton

## Behavioral

Concerned with how classes and objects interact

Command  
Iterator  
Observer  
State  
Strategy  
Template Method

## Structural

Let you compose classes or objects into larger structures

Adapter  
Composite  
Decorator  
Facade  
Proxy

# OTHER COMMON PATTERN CATEGORIES

---

## Class

Describe how relationships between classes are defined via inheritance. Relationships in class patterns are established at compile time.

Adapter

Factory Method

Template Method

## Object

Describe relationships between objects and are primarily defined by composition. Relationships in object patterns are typically created at runtime and are more flexible.

Decorator

Proxy

Strategy

Facade

Observer

Singleton

Beyond Today

# RESOURCES

# SOME BOOKS, PATTERN CATALOGS, AND WEBSITES

---



## **Head First Design Patterns**

Eric Freeman, Elisabeth Robson



## **Fundamentals of Software Architecture**

Mark Richards, Neal Ford



## **Refactoring Guru**



## **Design Patterns**

Erich Gamma, Richard Helm,  
Ralph Johnson, John Vlissides



## **A Philosophy of Software Design**

John Ousterhout



## **The Portland Patterns Repository**