Refactoring
For Observation

Novel Practices for Learning

Code Freeze 2020
David Laribee | dave@nerdnoir.com
Obvious
Cause and effect are repeatable and predictable.

Best Practice
(Sense-Categorize-Respond)
Standard Operating Procedures; Automation

Complicated
Cause and effect are separated over time and space.

Good Practice
(Sense-Analyze-Respond)
Predictive Planning; Rules; Expert Analysis

Complex
Cause and effect seen in retrospect and do not repeat.

Emergent Practice
(Probe-Sense-Respond)
Pattern Matching; Heuristics; Experimentation

Chaotic
Cause and effect are not usually perceivable.

Novel Practice
(Act-Sense-Respond)
Intervention; Crises Management; Rescue

Obvious
Cause and effect are repeatable and predictable.

Best Practice
(Sense-Categorize-Respond)
Standard Operating Procedures; Automation

Disorder
Obvious
Cause and effect are repeatable and predictable.

Best Practice
(Sense-Categorize-Respond)
Standard Operating Procedures; Automation

Complicated
Cause and effect are separated over time and space.

Good Practice
(Sense-Analyze-Respond)
Predictive Planning; Rules; Expert Analysis

Chaotic
Cause and effect are not usually perceivable.

Novel Practice
(Act-Sense-Respond)
Intervention; Crises Management; Rescue

Complex
Cause and effect seen in retrospect and do not repeat.

Emergent Practice
(Probe-Sense-Respond)
Pattern Matching; Heuristics; Experimentation

Disorder
Cause and effect are repeatable and predictable.

Best Practice
(Sense-Categorize-Respond)
Standard Operating Procedures; Automation

Complicated
Cause and effect are separated over time and space.

Good Practice
(Sense-Analyze-Respond)
Predictive Planning; Rules; Expert Analysis

Complex
Cause and effect seen in retrospect and do not repeat.

Emergent Practice
(Probe-Sense-Respond)
Pattern Matching; Heuristics; Experimentation

Chaotic
Cause and effect are not usually perceivable.

Novel Practice
(Act-Sense-Respond)
Intervention; Crises Management; Rescue

Obvious
Cause and effect are repeatable and predictable.

Best Practice
(Sense-Categorize-Respond)
Standard Operating Procedures; Automation

Disorder
Obvious
Cause and effect are repeatable and predictable.

Best Practice
(Sense-Categorize-Respond)
Standard Operating Procedures; Automation

Complicated
Cause and effect are separated over time and space.

Good Practice
(Sense-Analyze-Respond)
Predictive Planning; Rules; Expert Analysis

Complex
Cause and effect seen in retrospect and do not repeat.

Emergent Practice
(Probe-Sense-Respond)
Pattern Matching; Heuristics; Experimentation

Chaotic
Cause and effect are not usually perceivable.

Novel Practice
(Act-Sense-Respond)
Intervention; Crises Management; Rescue

Disorder

Obvious
Cause and effect are repeatable and predictable.

Best Practice
(Sense-Categorize-Respond)
Standard Operating Procedures; Automation
Architectural Mapping
Architectural Maps

Differ from traditional diagrams; they’re maps.

Are based on the “C4 Model” by Simon Brown.

Combine static structure with a planned and annotated route path.
Source: Simon Brown, “The C4 Model of Software Architecture”
System Context diagram for Internet Banking System

The system context diagram for the Internet Banking System.
Workspace last modified Tue Jan 14 2020 07:33:50 GMT-0500 (Eastern Standard Time)
Container diagram for Internet Banking System

The container diagram for the Internet Banking System.

Internet Banking System
- **Database** (Container: Oracle Database Server)
  - Stores user registration information, hashed authentication credentials, access logs, etc.

- **Web Application** (Container: Java and Spring MVC)
  - Delivers the static content and the Internet banking single page application.
  - Delivers to the customer’s web browser.

- **Single-Page Application** (Container: JavaScript and Angular)
  - Provides all of the Internet banking functionality to customers via their web browser.
  - Makes API calls to [JSON/HTTPS]

- **Mobile App** (Container: iOS or Android)
  - Provides a limited subset of the Internet banking functionality to customers via their mobile device.
  - Makes API calls to [JSON/HTTPS]

- **API Application** (Container: Java and Spring MVC)
  - Provides Internet banking functionality via a JSON/HTTPS API.
  - Sends e-mail using [SMTP]

- **E-mail System** (Software System)
  - The internal Microsoft Exchange e-mail system.
  - Sends e-mails to [SMTP]

- **Mainframe Banking System** (Software System)
  - Stores all of the core banking information about customers, accounts, transactions, etc.
Component diagram for Internet Banking System - API Application

- **Single-Page Application**
  - Container: Java/Spring
  - Provides all of the Internet banking functionality to customers via their web browser.
  - Makes API calls to [JSON HTTP]

- **Mobile App**
  - Container: Native
  - Provides a limited subset of the Internet banking functionality to customers via their mobile device.
  - Makes API calls to [JSON HTTP]

- **Sign In Controller**
  - Component: Spring/MVC/REST Controller
  - Allows users to sign in to the Internet Banking System.
  - Uses [JSON HTTP]

- **Reset Password Controller**
  - Component: Spring/MVC/REST Controller
  - Allows users to reset their passwords with a single use URL.
  - Uses [JSON HTTP]

- **Accounts Summary Controller**
  - Component: Spring/MVC/REST Controller
  - Provides customers with a summary of their bank accounts.
  - Uses [JSON HTTP]

- **Security Component**
  - Component: Spring/Thrift/REST
  - Provides functionality related to signing in, changing passwords, etc.

- **E-mail Component**
  - Component: Spring/Thrift/REST
  - Sends e-mails to users.

- **E-mail System**
  - Container: Microsoft Exchange e-mail system.
  - Sends e-mail using [JSON HTTP]

- **Mainframe Banking System Facade**
  - Component: Spring/Thrift/REST
  - A facade onto the mainframe banking system.

- **Database**
  - Container: Oracle Database Schema
  - Stores user registration information, hashed authentication credentials, account data, etc.

- **Mainframe Banking System**
  - Container: Oracle Database Schema
  - Stores all of the core banking information about customers, accounts, transactions, etc.
C4 Model + Journeys

C1 - System Landscape and/or Context
C2 - Container Diagrams (Stacks & Deploy)
C3 - Component Diagram (Logical Arch.)
Journeys - Data Flows, Extraction Plan

@laribee
Sum Up

Make architecture a social activity!

People are a part of your system, too. If they don’t understand your architecture, they may be operating from a position of chaos or complexity.

@laribee
Sum Up

Plot journeys in diagrams to previsualize risky changes or large scale refactorings.

Only diagram what you need to. Remember: diagrams, just like code, need maintenance.
Refactoring With Telemetry
Refactoring

Changing the structure of an application without changing its behavior.

Helps introduce new domain concepts and in moving from specific to general (abstraction).

A healthy part of a complete XP breakfast.

@laribee
TDD + Refactoring

Make structural changes to improve clarity and lessen cognitive load.

Refactor

Red

Write a new test to extend the behavior of the system.

Green

Write *just* enough code to make the new test pass. Ensure all tests pass.
Refactoring Problems

If you didn’t start with TDD, it can be difficult. Optimally backed by a “safety net” test suite. High-skill,-discipline, and -commitment.

Refactor to what?!
Refactor to what?!

Principles and heuristics such as SOLID.
Domain alignment à la DDD.
Lighter cognitive load, better Developer Experience.

@laribee
TDD + Refactoring

- **Red**: Write a new test to extend the behavior of the system.
- **Green**: Write *just* enough code to make the new test pass. Ensure all tests pass.
- **Refactor**: Make structural changes to improve clarity and lessen cognitive load.
A Different Loop

Find your refactoring target(s).
Targeting Change

Intuition- or consensus-based.

Archeological - “we have \( N \) ways”

Code smells - “Primitive Obsession”

Visualization or data-driven, e.g. cyclomatic complexity plus method size.
A Different Loop

- **Learn**: Use telemetry, discussion, and teaching to discuss proposed changes.
- **Refactor**: Execute refactoring experiments and runs until satisfied.
- **Target**: Find your refactoring target(s).
Refactor, Learn, Repeat

Aesthetic convergence - “how we do it.”
Temporary technical coaching; after several weeks the team no longer needs them.
SQALE and other code quality metrics add some nice data to the conversations.

@laribee
A Different Loop

Pushing units. Making bank.

Find your refactoring target(s).

Use telemetry, discussion, and teaching to discuss proposed changes.

Execute refactoring experiments and runs until satisfied.

Commit

Target

Learn

Refactor

@laribee
Tools, Techniques, Resources

**Refactoring**

“Refactoring (2nd Edition)” - Fowler  
“Refactoring to Patterns” - Kerievsky  
Smells to Refactoring Cheat Sheet

Jetbrains Resharper / Rider  
Visual Studio / IntelliJ  
Manual Refactoring As Needed

**Metrics**

NDepend / JArchitect  
SonarQube / Code Climate

**Other**

Mob Programming  
ProTip: line numbers on in IDE!
public decimal CalculateTotal()
{
    var subtotal = _items.Sum(i => i.Price);

    var electronicsDiscount = 0m;

    var electronicsItems = _items.Where(i => i.Type == ItemType.Electronics);
    var sameItem = electronicsItems.GroupBy(i => i.Sku);

    foreach (var item in sameItem)
    {
        if (item.Count() >= 4)
        {
            electronicsDiscount = item.First().Price;
            break;
        }
    }

    var expensiveApplianceDiscount = 0m;

    var applianceItem = _items.Where(i => i.Type == ItemType.Appliance);
    var qualifyingAppliances = applianceItem.Where(i => i.Price > 500.0m);

    if (qualifyingAppliances.Count() >= 2) expensiveApplianceDiscount = 100m;

    return subtotal - electronicsDiscount - expensiveApplianceDiscount;
}
public decimal CalculateTotal()
{
    var subtotal = _items.Sum(i => i.Price);

    var electronicsDiscount = 0m;
    var electronicsItems = from item in _items
                            where itemItemType == ItemType.Electronics
                            select item;
    foreach (var item in electronicsItems)
    {
        if (item.Count() > 1)
        {
            electronicsDiscount += item.Price;
        }
    }
    if (qualifyingAppliance != null)
    {
        return subtotal - electronicsDiscount;
    }
    return subtotal;
}
public decimal CalculateTotal()
{
    var subtotal = _items.Sum(i => i.Price);

    var electronicsDiscount = 0m;

    var electronicsItems = _items.Where(i => i.Type == ItemType.Electronics);
    var sameItem = electronicsItems.GroupBy(i => i.ItemId)
    .Where(g => g.Count() >= 4)
    .Select(g => g.First().ItemId);

    foreach (var item in sameItem)
    {
        if (item.Count() >= 4)
        {
            electronicsDiscount = item.First().Price * 0.1m;
            break;
        }
    }

    var expensiveApplianceDiscount = 0m;

    var applianceItem = _items.Where(i => i.Type == ItemType.Appliance)
    .Where(i => qualifyingAppliances.Contains(i.ItemId));

    if (qualifyingAppliances.Count() >= 2)
    {
        expensiveApplianceDiscount = applianceItem.Sum(i => i.Price) * 0.2m;
    }

    return subtotal - electronicsDiscount - expensiveApplianceDiscount;
}

private decimal CalculateElectronicsDiscount()
{
    var electronicsDiscount = 0m;

    var electronicsItems = _items.Where(i => i.Type == ItemType.Electronics);
    var applianceItem = _items.Where(i => i.Type == ItemType.Appliance);
    var qualifyingAppliances = applianceItem
        .Where(i => i.ItemId == sameItemgroupid)
        .Select(i => i.Price);

    if (qualifyingAppliances.Count() >= 2)
    {
        electronicsDiscount = qualifyingAppliances.Sum() * 0.1m;
    }

    return subtotal - electronicsDiscount - expensiveApplianceDiscount;
}
public decimal CalculateTotal()
{
    var subtotal = _items.Sum(i => i.Price);

    var electronicsDiscount = CalculateElectronicsDiscount();
    var expensiveApplianceDiscount = CalculateExpensiveApplianceDiscount();

    return subtotal - electronicsDiscount - expensiveApplianceDiscount;
}

private decimal CalculateElectronicsDiscount() {...}

private decimal CalculateExpensiveApplianceDiscount() {...}
namespace example
{
    public class Order
    {
        private readonly List<Item> _items = new List<Item>();
        private readonly IEnumerable<IDiscount> _discounts;

        public Order(IEnumerable<IDiscount> discounts = null)
        {
            _discounts = discounts;
        }

        public void AddItem(Item item)
        {
        }

        public decimal CalculateTotal()
        {
            var subtotal = _items.Sum(i => i.Price);
            var promotionalDiscounts = _discounts.Sum(d => d.Calculate(_items));

            return subtotal - promotionalDiscounts;
        }
    }

    public interface IDiscount
    {
        decimal Calculate(IEnumerable<Item> orderItems);
    }

    public class ExpensiveAppliancesPromotion : IDiscount
    {
        public decimal Calculate(IEnumerable<Item> orderItems)
        {
        }

        private static IEnumerable<Item> FilterQualifyingAppliances(IEnumerable<Item> orderItems)
        {
        }
    }

    public class BuyFourGetOneElectronicsPromotion : IDiscount
    {
    }
}
public Order(IEnumerable<IDiscount> discounts = null)
{
    _discounts = discounts;
}

public void AddItem(Item item)
{
}

public decimal CalculateTotal()
{
    var subtotal = _items.Sum(i => i.Price);
    var promotionalDiscounts = _discounts.Sum(d => d.Calculate(_items));
    return subtotal - promotionalDiscounts;
}

public interface IDiscount
{
    decimal Calculate(IEnumerable<Item> orderItems);
}

public class ExpensiveAppliancesPromotion : IDiscount
{
    public decimal Calculate(IEnumerable<Item> orderItems)
    {
        // implementation...
    }

    private static IEnumerable<Item> FilterQualifyingAppliances(IEnumerable<Item> orderItems)
    {
        // implementation...
    }
}

public class BuyFourGetOneElectronicsPromotion : IDiscount
{
    public decimal Calculate(IEnumerable<Item> orderItems)
    {
        // implementation...
    }

    private static IEnumerable<IGrouping<string, Item>> FilterQualifyingElectronics(IEnumerable<Item> orderItems)
    {
        // implementation...
    }
}
Sum Up

Use metrics like cyclomatic complexity, method size (LoC), and SQALE to spark insight and observe relationships.

Return to safety (git checkout .) when things get wonky; they will.
Sum Up

Use an automated refactoring tool!

Do this in a group. I like mob programming for this kind of session.

You will speed up and find refactoring you can apply to other areas of your codebase.
RWT - Debt %

Metric: Percentage Debt (Metric)
Value at January 13th 2020 11:57 (v1.0)
2.1 %
Thank You!

Code Freeze 2020
David Laribee | dave@nerdnoir.com