Refactoring towards Design Patterns

Benjamin Eberlei, Qafoo GmbH

October 27, 2016

Motivation

- Neglecting design leads to <u>underengineering</u>
- Over-focusing on Design-Pattern leads to <u>overengineering</u>

Refactoring

- small changes to internal code structure

Refactoring

- small changes to internal code structure
- external code structure keeps the old behavior
    - Method/Function

Refactoring

- small changes to internal code structure
- external code structure keeps the old behavior
    - Method/Function
    - Class

Refactoring

- small changes to internal code structure
- external code structure keeps the old behavior
  - Method/Function
  - Class
  - System/API

Refactoring

- small changes to internal code structure
- external code structure keeps the old behavior
  - Method/Function
  - Class
  - System/API
- Tests increase reliability and speed of refactorings

Refactoring

- small changes to internal code structure
- external code structure keeps the old behavior
  - Method/Function
  - Class
  - System/API
- Tests increase reliability and speed of refactorings
- No tests are fine: refactorings can be performed very mechanically/automatically

Steps

1. Make it work
2. Make it nice

Goals

- Simpler to understand, change
- Reusable
- Less dependencies
- (Unit-) Testable

Design Patterns

A software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. (Wikipedia)

Refactoring and Patterns

There is a natural relation between patterns and refactoring. Patterns are where you want to be; refactorings are ways to get there from somewhere else. *(*Martin Fowler, Refactoring p. 107*)*

Refactoring and Patterns

Refactoring towards Patterns to avoid both <u>under-</u> and <u>overengineering</u>.

Refactoring Basics: Extract Method

- Identify lines to extract from a method/function
- Create new, empty method without arguments
- Copy lines over to new method
- Find all variables declared outside method, define as argument
- Find all variables used after method, define as return value
- Identify instance variables that can be turned into argument

Failsafes

- Version Control: Every successful step is one commit
- IDEs: Automate extract method using tools (PHPStorm, ...)
- ”Scientist”: Keep old code and compare result of old vs new
- Tests: Verify old logic still works

Code Smell: Construction Spread Everywhere!

- **Problem:** All parts of your app create and configure objects
- Complicates the reuse of objects
- Gravitates application towards use of Singletons
- Prevents exchange of code at runtime (dynamic binding)

Factory

A factory creates an object for you.

- Getting control over object creation

Factory

A factory creates an object for you.

- Getting control over object creation
- Most important issue for every code-base

Factory

A factory creates an object for you.

- Getting control over object creation
- Most important issue for every code-base
- Actually 4 patterns
  - Factory
  - Factory Method
  - Abstract Factory
  - Builder

Refactoring: Move creation knowledge to Factory

1. Extract creation logic into Factory Method

Refactoring: Move creation knowledge to Factory

1. Extract creation logic into Factory Method
2. Introduce Lazy Initialization

Refactoring: Move creation knowledge to Factory

1. Extract creation logic into Factory Method
2. Introduce Lazy Initialization
3. Introduce Setter for ”Dependency Injection”

Refactoring: Move creation knowledge to Factory

1. Extract creation logic into Factory Method
2. Introduce Lazy Initialization
3. Introduce Setter for ”Dependency Injection”
4. Extract Factory method into class

Refactoring: Move creation knowledge to Factory

1. Extract creation logic into Factory Method
2. Introduce Lazy Initialization
3. Introduce Setter for ”Dependency Injection”
4. Extract Factory method into class
5. Invert dependency graph

Code Smell: Singleton

- **Problem:** To fix scattered object creation, Singleton Pattern is used
- Shared global state that is causing side effects
- Reduced testability

Refactoring: Inline Singleton

- Extract Method: Usage of Singleton into Factory Method

Refactoring: Inline Singleton

- Extract Method: Usage of Singleton into Factory Method
- Introduce Lazy Initialization

Refactoring: Inline Singleton

- Extract Method: Usage of Singleton into Factory Method
- Introduce Lazy Initialization
- Introduce Setter for ”Dependency Injection”

Code Smell: God Object

- **Problem:** Class/Methods too large with multiple responsibilities that cannot be untangled
- Prevents reuse of individual parts
- High complexity
- Usually grows larger because of Feature Envy

Facade

A facade provides a simplified interface to a larger body of code.

Facade

A facade provides a simplified interface to a larger body of code.

- Make code reusable (business logic, ..)
- Integrate third party code (libraries)

Facade

A facade provides a simplified interface to a larger body of code.

- Make code reusable (business logic, ..)
- Integrate third party code (libraries)
- Avoid hard dependencies on technical details

Facade

A facade provides a simplified interface to a larger body of code.

- Make code reusable (business logic, ..)
- Integrate third party code (libraries)
- Avoid hard dependencies on technical details
- Strongly Related to the Adapter/Bridge patterns

Refactoring: Compose Methods/Classes

1. Identify lines that should be composed into new method/class

Refactoring: Compose Methods/Classes

1. Identify lines that should be composed into new method/class
2. Perform Extract Method

Refactoring: Compose Methods/Classes

1. Identify lines that should be composed into new method/class
2. Perform Extract Method
3. Identify Dependencies used in Extracted Method

Refactoring: Compose Methods/Classes

1. Identify lines that should be composed into new method/class
2. Perform Extract Method
3. Identify Dependencies used in Extracted Method
4. Extract Class including dependencies

Refactoring: Compose Methods/Classes

1. Identify lines that should be composed into new method/class
2. Perform Extract Method
3. Identify Dependencies used in Extracted Method
4. Extract Class including dependencies
5. Move method to new class

Refactoring: Compose Methods/Classes

1. Identify lines that should be composed into new method/class
2. Perform Extract Method
3. Identify Dependencies used in Extracted Method
4. Extract Class including dependencies
5. Move method to new class
6. Integrate into factory

Code Smell: Primitive Obsession

- **Problem:** Using primitive types of language and libraries everywhere
- Internals and assumptions of classes are shared throughout application
- Logic has to be re-implemented everywhere
- Prevents changing the internals
- Leaky abstraction increases the required mental model of developers

Replace Type Code with Class

1. Identify Primitive variables and Logic

Replace Type Code with Class

1. Identify Primitive variables and Logic
2. Extract into new Method + Class

https://qafoo.com/newsletter

Qafoo
passion for software quality

THANK YOU

Rent a quality expert
qafoo.com