Bringing OOP Best Practices to the World of Functional Programming

Elana Hashman

Computer Science Club – Fall 2016

```
$ whoami
ehashman
$ ldapsearch -x uid=ehashman
# extended LDIF
#
# LDAPv3
# base <dc=csclub, dc=uwaterloo, dc=ca> (default) with scope subtree
# filter: uid=ehashman
# requesting: ALL

# ehashman, People, csclub.uwaterloo.ca
dn: uid=ehashman,ou=People,dc=csclub,dc=uwaterloo,dc=ca
uid: ehashman
cn: Elana Hashman
loginShell: /bin/bash
uidNumber: 21685
gidNumber: 21685
homeDirectory: /users/ehashman
program: Alumni
term: f2010
...
term: w2018
objectClass: account
objectClass: member
objectClass: posixAccount
objectClass: shadowAccount
objectClass: top
```

DESIGN PATTERNS

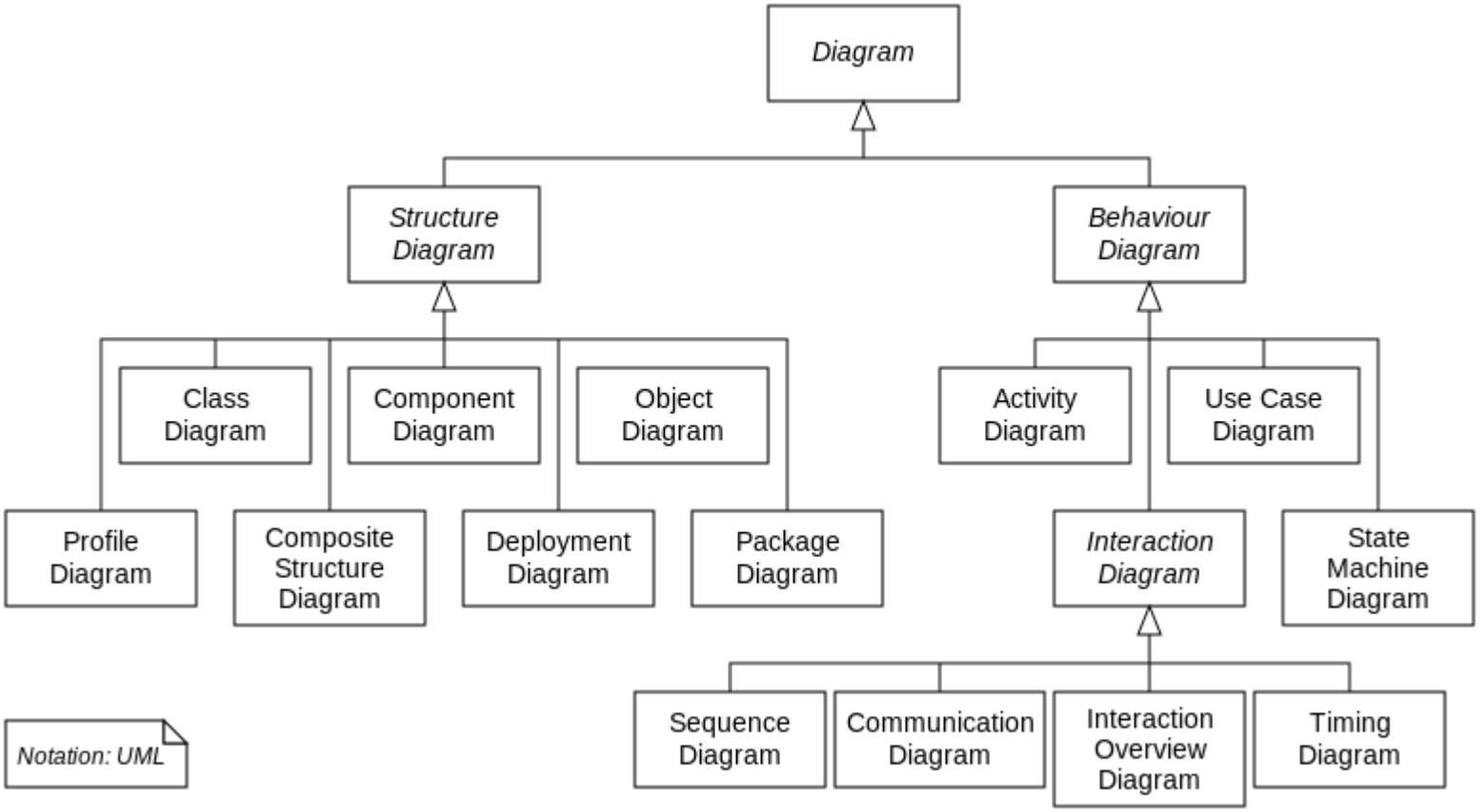
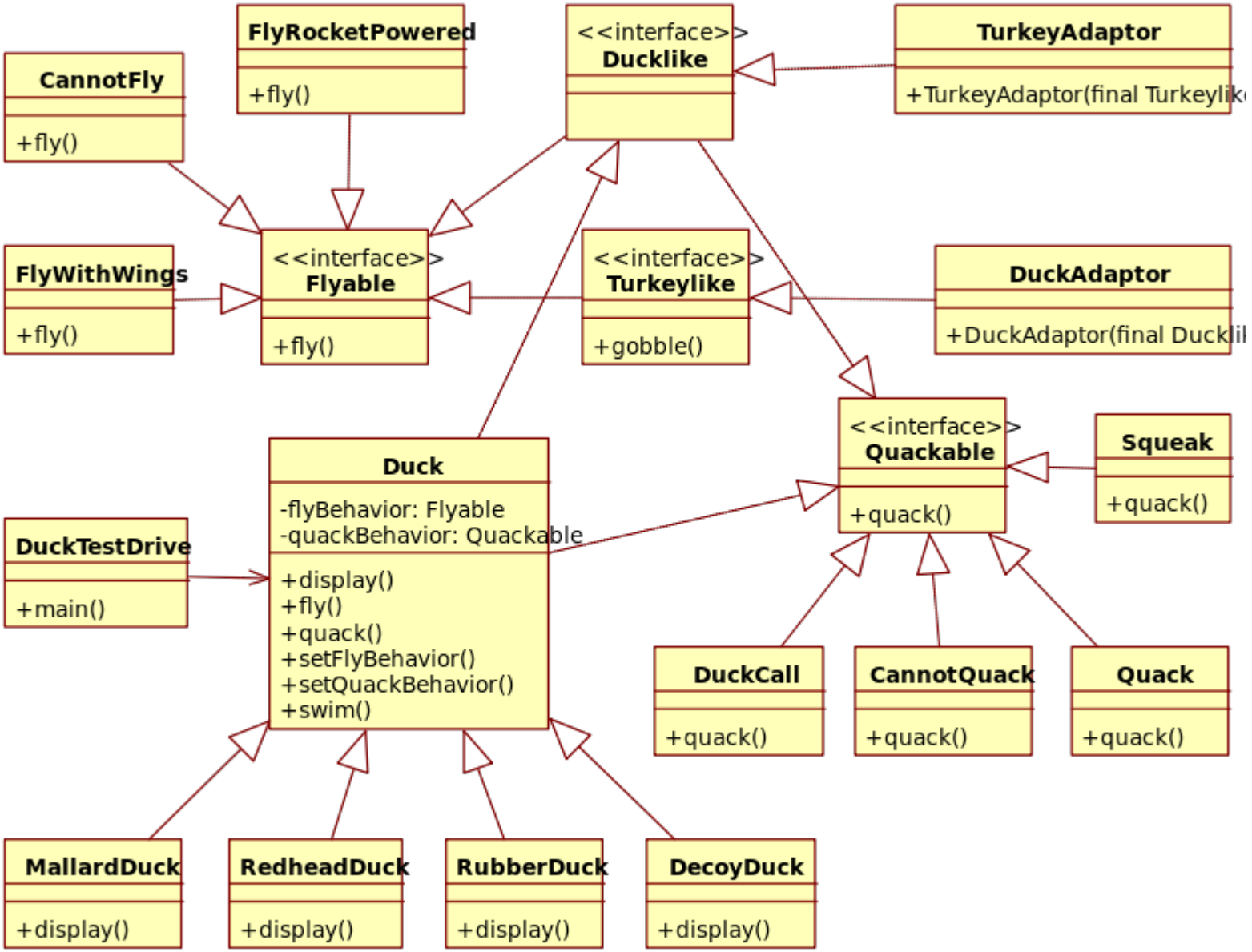


Diagram
Structure Diagram
Behaviour Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
Use Case Diagram
Profile Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Interaction Diagram
State Machine Diagram
Sequence Diagram
Communication Diagram
Interaction Overview Diagram
Timing Diagram
Notation: UML


FlyRocketPowered
+fly()
<<interface>>
Ducklike
TurkeyAdaptor
+TurkeyAdaptor(final Turkeylik
CannotFly
+fly()
FlyWithWings
+fly()
<<interface>>
Flyable
+fly()
<<interface>>
Turkeylike
+gobble()
DuckAdaptor
+DuckAdaptor(final Ducklil
<<interface>>
Quackable
+quack()
Squeak
+quack()
Duck
-flyBehavior: Flyable
-quackBehavior: Quackable
+display()
+fly()
+quack()
+setFlyBehavior()
+setQuackBehavior()
+swim()
DuckTestDrive
+main()
DuckCall
+quack()
CannotQuack
+quack()
Quack
+quack()
MallardDuck
+display()
RedheadDuck
+display()
RubberDuck
+display()
DecoyDuck
+display()

Functional programming?

Object-oriented programming?

Programming Paradigms

- **What is Object-Oriented Programming (OOP)?**
  - Program logic is organized around *objects*, which store data as *properties* and algorithms for the data as *methods*
  - Objects are described by *classes*, which can inherit from one another
  - Common recipes/best practices are codified as *design patterns*
- Examples
  - C++, Java, C#, Python

Programming Paradigms

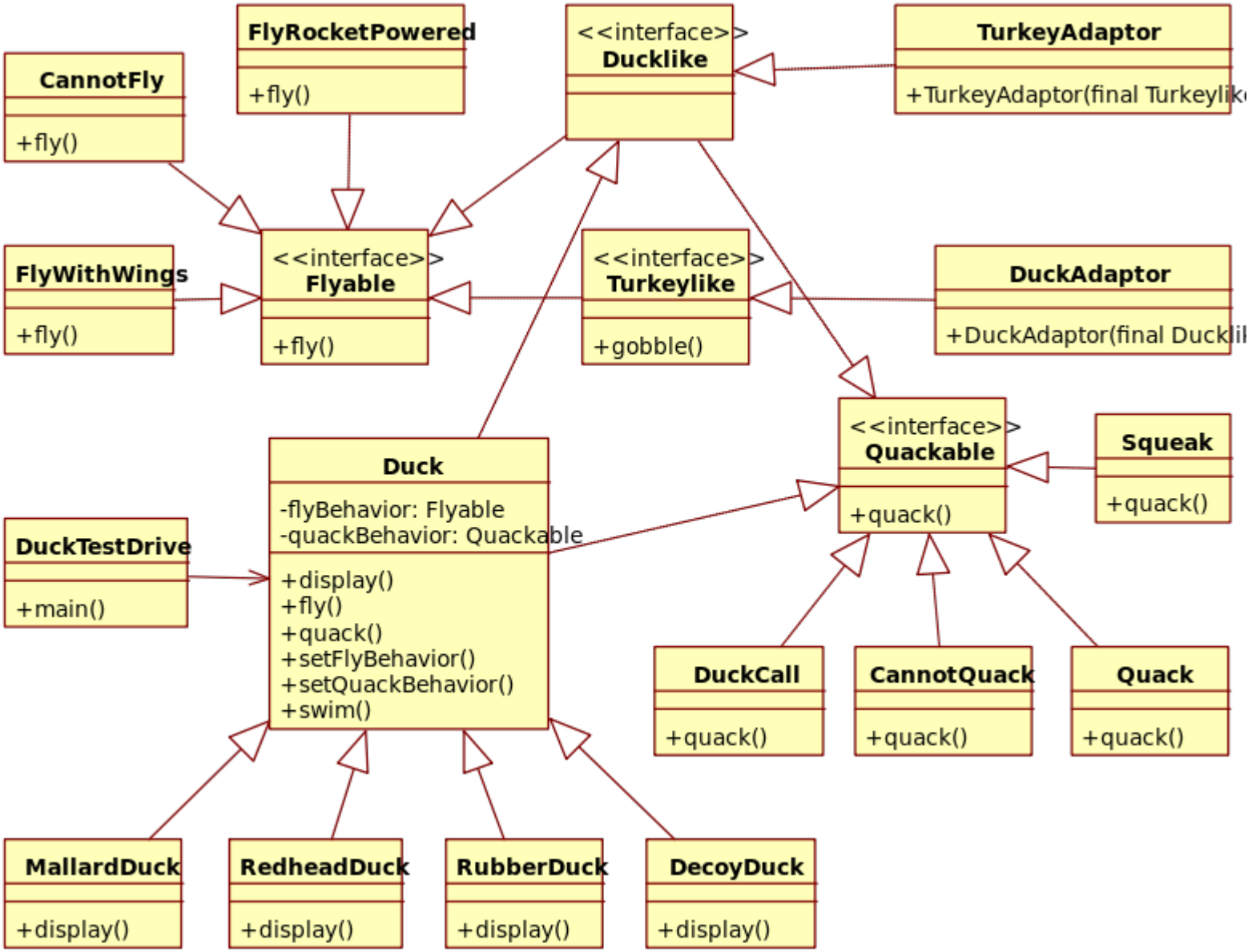
- **What is Functional Programming (FP)?**
  - Logic is codified and evaluated as pure functions, avoiding the mutation of state
  - Commonly characterized by the use of *first-order functions*, which can be passed around as arguments, and *higher-order functions*, which take functions as arguments
- Examples
  - Lisp (with various dialects, such as Racket), Haskell, OCaml, F#

The fateful interview

“Programming Tropes”

…

a.k.a. DESIGN PATTERNS


FlyRocketPowered
+fly()
<<interface>>
Ducklike
TurkeyAdaptor
+TurkeyAdaptor(final Turkeylik
CannotFly
+fly()
FlyWithWings
+fly()
<<interface>>
Flyable
+fly()
<<interface>>
Turkeylike
+gobble()
DuckAdaptor
+DuckAdaptor(final Ducklil
<<interface>>
Quackable
+quack()
Squeak
+quack()
Duck
-flyBehavior: Flyable
-quackBehavior: Quackable
+display()
+fly()
+quack()
+setFlyBehavior()
+setQuackBehavior()
+swim()
DuckTestDrive
+main()
DuckCall
+quack()
CannotQuack
+quack()
Quack
+quack()
MallardDuck
+display()
RedheadDuck
+display()
RubberDuck
+display()
DecoyDuck
+display()

What is “good code?”

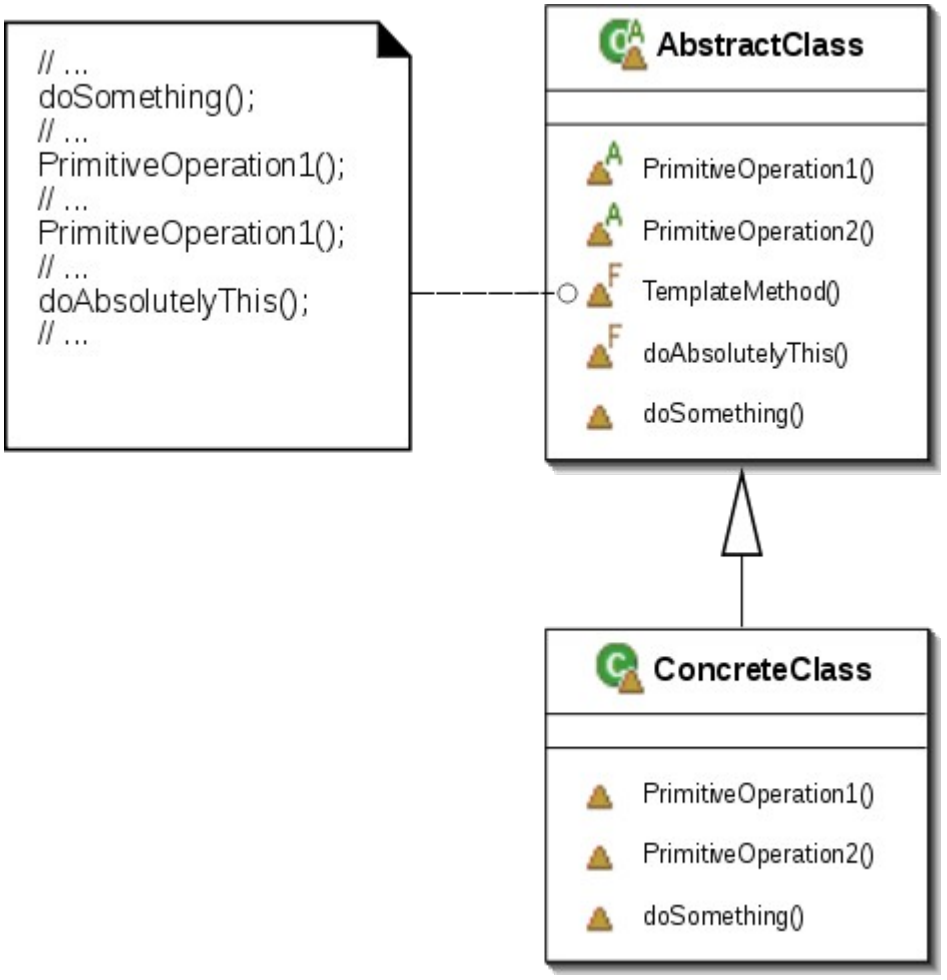
“I can't tell you what bad code looks like, but *I know it when I see it*.”

- Anonymous Programmer

Don't Repeat Yourself (DRY)

Ya Ain't Gonna Need It (YAGNI)

Keep It Simple, Senator (KISS)


// ...
doSomething();
// ...
PrimitiveOperation1();
// ...
PrimitiveOperation1();
// ...
doAbsolutelyThis();
// ...
AbstractClass
PrimitiveOperation1()
PrimitiveOperation2()
TemplateMethod()
doAbsolutelyThis()
doSomething()
ConcreteClass
PrimitiveOperation1()
PrimitiveOperation2()
doSomething()

```
(defn common-logic
  [data case-specific-logic]
  ...)

(defn data1-specific-logic [args] ...)

(defn data2-specific-logic [args] ...)

(for [[data fn] data-and-fns]
  (common-logic data fn))
```

I.

Different programming paradigms are not as different as you might think.

II.

Different programming paradigms
do have different design philosophies.

“It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.”

- Alan J. Perlis

III.

No paradigm is objectively "better," but each has advantages in certain situations.

A common critique of the “Gang of Four” 1995 *Design Patterns* book was that many of the patterns served as workarounds for language limitations of C++.

In 1996, Peter Norvig claims that 16 of the 23 patterns in the Gang of Four book are invisible or simplified in Lisp!

- Abstract Factory, Flyweight, Factory Method, State, Proxy, Chain of Responsibility → **first-class types**
- Command, Strategy, Template, Visitor → **first-class functions**
- Interpreter, Iterator → **macros**
- Mediator, Observer → **method combination**
- Builder → **multimethods**
- Façade → **modules**

Functional programming language features can **reduce the need** to use fully-implemented class-based design patterns.

Some Notable FP Language Features

Immutability

- **What is it?**
  - Data cannot change state after its creation
  - Functions cannot have “side effects”
- Improves ability to reason about code
- Simplifies unit testing
- Discourages the use of global mutable state, i.e. Singleton pattern

First-Class Functions

- **What are they?**
  - Functions can be passed as arguments
  - Lexical scoping allows for local bindings of values
- *Currying* gives us function Factories

```
((fn [a] (fn [b] (does-stuff a b))) "yo")

=> (fn [b] (does-stuff "yo" b)) ;; a := "yo"
```

- Treating functions like we treat data gives us programmable power over them

Macros and Pattern Matching

- **What are they?**
  - Macros: we can write code to generate code
  - Pattern Matching: match data according to patterns!
- Enables writing new grammar and evaluating it with ease, i.e. Interpreter pattern
- Brevity means devs can write more with less time, and the resulting code needs less maintenance
- Pattern match example...

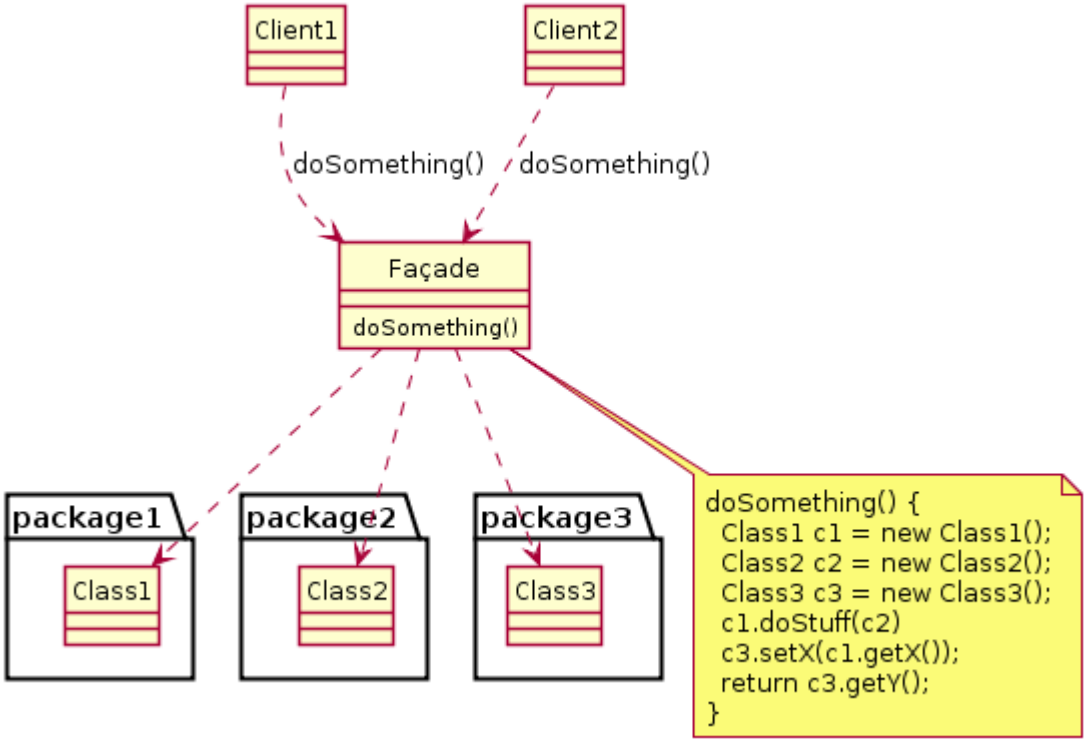
Macros and Pattern Matching

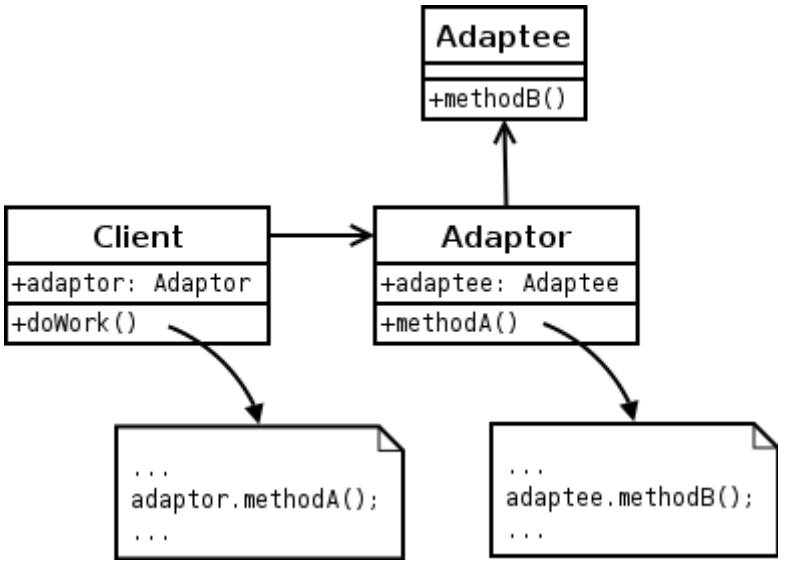
```
(defn message-origin
  [message]
  (match [message]
    [{:parsed {:metadata
               {:customer-id-string _}}}] :type1
    [{:parsed {:id _
               :type _
               :critical _
               :message _}}] :type2
    [{:parsed _}] :json
    :else :syslog))
```

Some Examples of Design Patterns

The Façade and Adapter Patterns

- **What are they?**
  - Hide an existing API by providing a new one on top





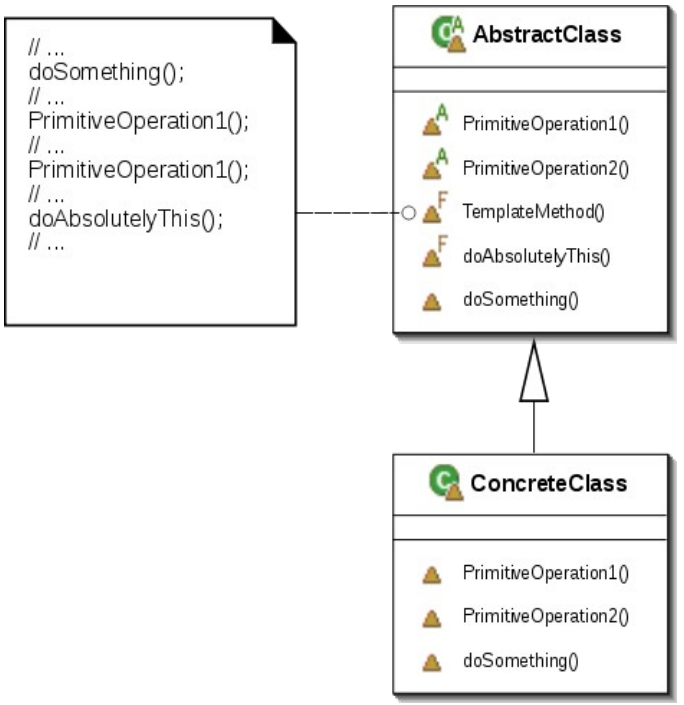
The Façade and Adapter Patterns

- **When to use them?**
  - Provide a unified or simplified interface for other code
  - *Technical debt wrangling:* standardize an interface so you can refactor the original code behind it
- **When not to use them?**
  - Too many layers of indirection from the original API can be fragile
- **How to use them with FP?**
  - Just like you would in the OOP world!
  - Write modules with public functions instead of classes

```clojure
(defn yucky-API
  [hot dog other-infos] ...)

(defn consumes-yucky-API []
  (yucky-API true true
             {:hot true
              :dog true
              :sundaes "???"}))

(defn nice-API
  "helpful docstring!"
  [other-infos]
  (let [{:keys [hot dog]} other-infos]
    (yucky-API hot dog other-infos)))
```

The Template Pattern

- **What is it?**
  - Defines the majority of an algorithm in an operation, deferring some steps to subclasses



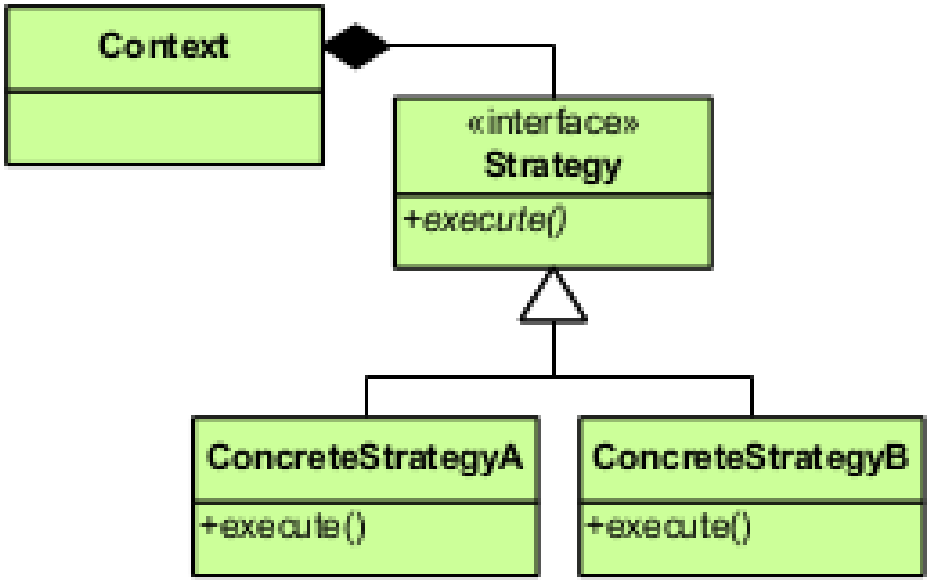
The Template Pattern

- **When to use it?**
  - Nearly identical data and data operations
  - Need to stub out a small amount of functionality
- **When not to use it?**
  - Use abstract base classes or similar very sparingly
  - In OOP land: prefer composition over inheritance
- **How to use it with FP?**
  - Pass in stub functions as arguments to common logic instead of implementing stubs on subclasses

```
(defn common-logic
  [data case-specific-logic]
  ...)

(defn data1-specific-logic [args] ...)

(defn data2-specific-logic [args] ...)

(for [[data fn] data-and-fns]
  (common-logic data fn))
```

The Strategy Pattern

- **What is it?**
  - Conditionally switch algorithms in a given context



The Strategy Pattern

- **When to use it?**
  - Encapsulates dispatching many variants of a similar algorithm
  - Feature-flagged functionality
- **When not to use it?**
  - When strategies fundamentally differ (e.g. return type)
  - Adds complexity and code branching
- **How to use it with FP?**
  - Pass in algorithm variants as first-order functions

```
(defn strategy1 [args] ...)

(defn strategy2 [args] ...)

(apply-strategy-a [strategy1 strategy2])

(apply-strategy-b
 (fn [cond] (if cond
              strategy1
              strategy2))
```

Summary

We covered some design patterns from all three categories!

Creational

- Factory
- Singleton

Structural

- Adapter
- Façade

Behavioural

- Interpreter
- Strategy
- Template

Conjecture: You can use functional programming languages to write industry software.

Evidence: My team!

~~join the party~~

THIS ADVICE PROVIDED WITH ABSOLUTELY NO WARRANTY

Thank you!

Thanks to: Nik Black, Fatema Boxwala,
Shane Wilton, Rackspace

Talk links, references, and resources can be found at
https://hashman.ca/osb-2016/

Talk References

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (1995)
  - Referred to as the "Gang of Four"
- Peter Norvig, "Design Patterns in Dynamic Languages" (1996)

Image Licenses

- **Public Domain**
  - Meta-UML Diagram
  - Adapter Pattern UML Diagram
  - Strategy Pattern UML Diagram
- **Creative Commons Share Alike 3.0 Unported License**
  - UML diagram of composition over inheritance
  - Template Method: UML Class Diagram
  - Facade Design Pattern in UML