Improving Object-Oriented Programming by Integrating Language Features to Support Immutability

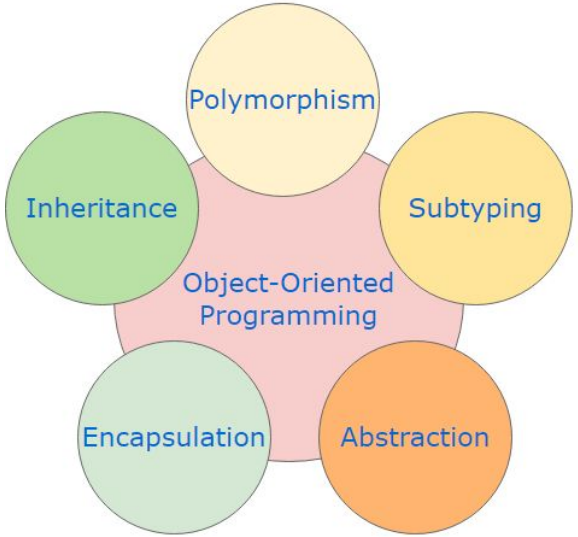
By William Flageol
Concordia University

Supervised by:
Yann-Gaël Guéhéneuc, Concordia University
Mourad Badri, Université du Québec à Trois-Rivières
Stefan Monnier, Université de Montréal

2023-05-29

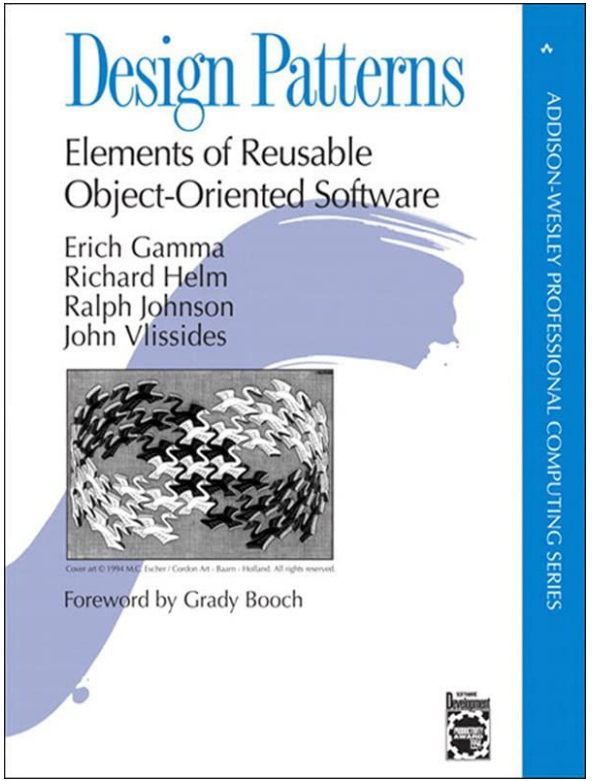
Object-Oriented Programming

- De-facto general programming paradigm
- Principles and practices
  - SOLID [63]
  - GRASP
  - GoF design patterns [31]
- Other paradigms
  - Functional programming
  - Meta-programming



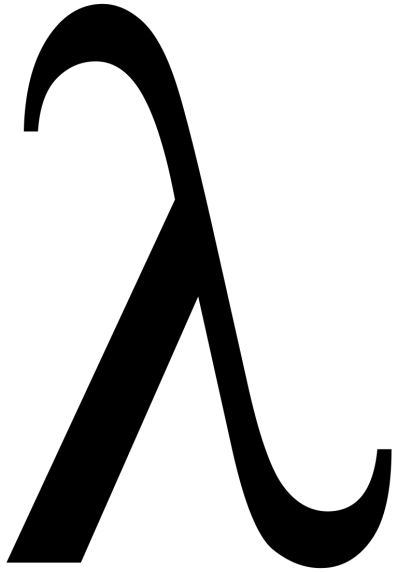
About OOP Design Patterns

- Generic solutions to recurring problems
  - Require design patterns to solve
  - Added complexity
- Can OOP be improved by adding language features that solve the underlying issue behind design patterns?
  - Improve pattern implementations
  - Replace design patterns

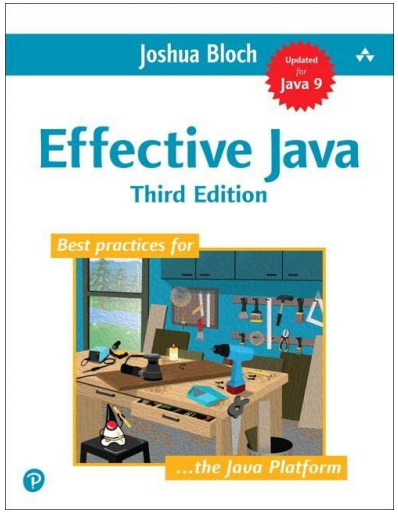


About Immutability

- Core property of Functional Programming
- Advantages include [1, 11]
  - More consistent behaviour
  - Makes code easier to understand
  - Easier record keeping
- Advantages tend to be rhetorical
  - No empirical data
- Research focuses on enforcement
  - [13, 21, 31, 51, 85, 91, 92, 95]
- Industry focuses on support
  - C#, Java, Rust, Kotlin, JavaScript, etc.



Rules for transitive immutability in OOP



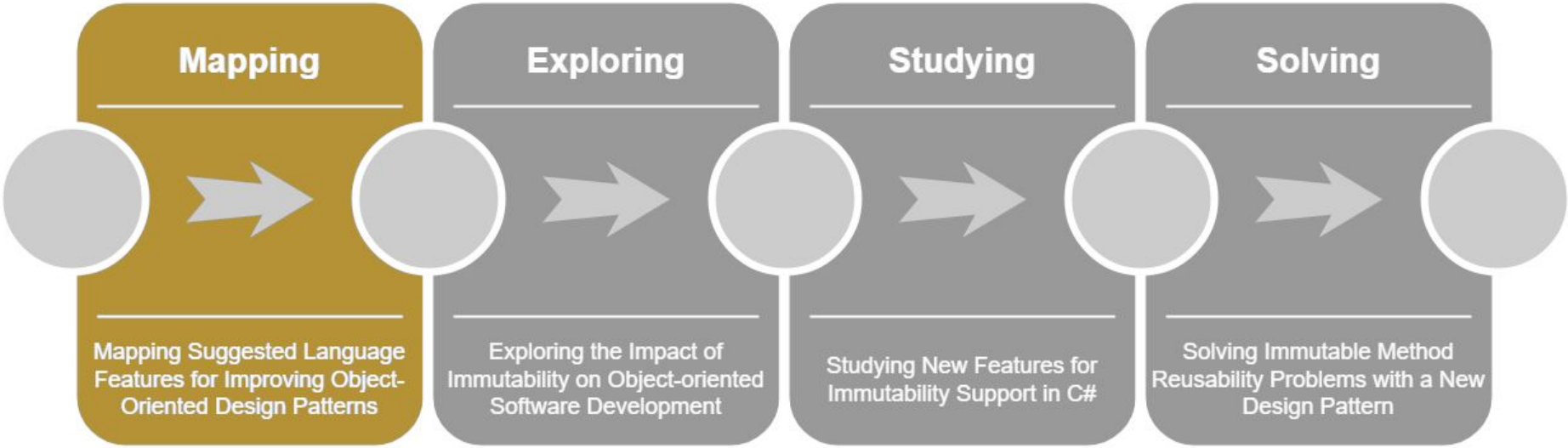
- “Don’t provide methods that modify the object’s state.”
- “Ensure that the class can’t be extended.”
- “Make all fields final.”
- “Make all fields private.”
- “Ensure exclusive access to any mutable components.”

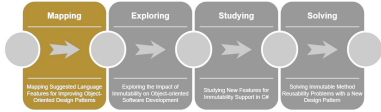
Presented by Bloch [11] in *Effective Java*

Thesis Statement

- OOP can be improved by adding language features that solve the underlying issue behind design patterns
- Among many possibilities, we focus on immutability
  - Can increase understandability and granularity of the code
  - Immutability features can improve maintainability, reduce code duplication, and improve scalability

Thesis Flow


Mapping
Mapping Suggested Language Features for Improving Object-Oriented Design Patterns
Exploring
Exploring the Impact of Immutability on Object-oriented Software Development
Studying
Studying New Features for Immutability Support in C#
Solving
Solving Immutable Method Reusability Problems with a New Design Pattern



An Example: Improving the Factory Method

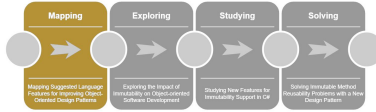
C++ Example [31]

```
class Document {
  virtual void Open() const = 0;
  virtual void Close() const = 0;

  void Save() {
    // ...
  }

  void Revert() {
    // ...
  }
}

class MyDocument : public Document
{
  void Open() const override {
    // ...
  }

  void Close() const override {
    // ...
  }
}
```

```
class Application {
  virtual Document*
CreateDocument() const = 0;

  Document* NewDocument() {
    // ...
  }

  Document* OpenDocument() {
    // ...
  }
}

class MyApplication : public
Application {
  Document* CreateDocument()
const override {
    return new MyDocument();
  }
}
```

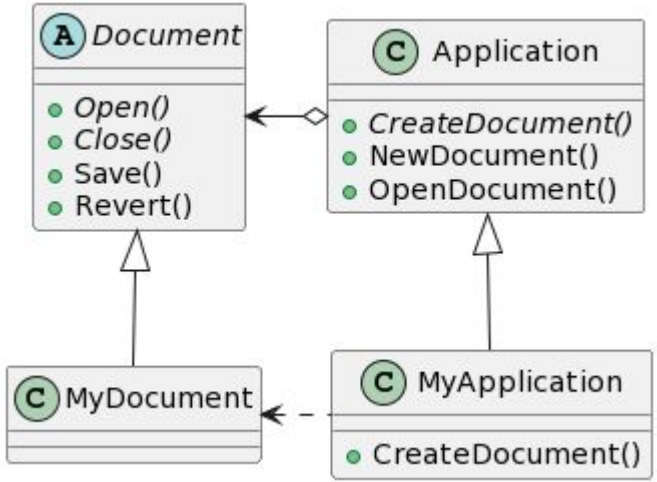
Smalltalk Example [31]

```
"In class MyApplication"

documentClass

    ^ MyDocument

createDocument

    ^ documentClass new
```



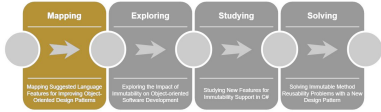
An Example: Improving the Factory Method

C++ Implementation [31]



Smalltalk Example [31]

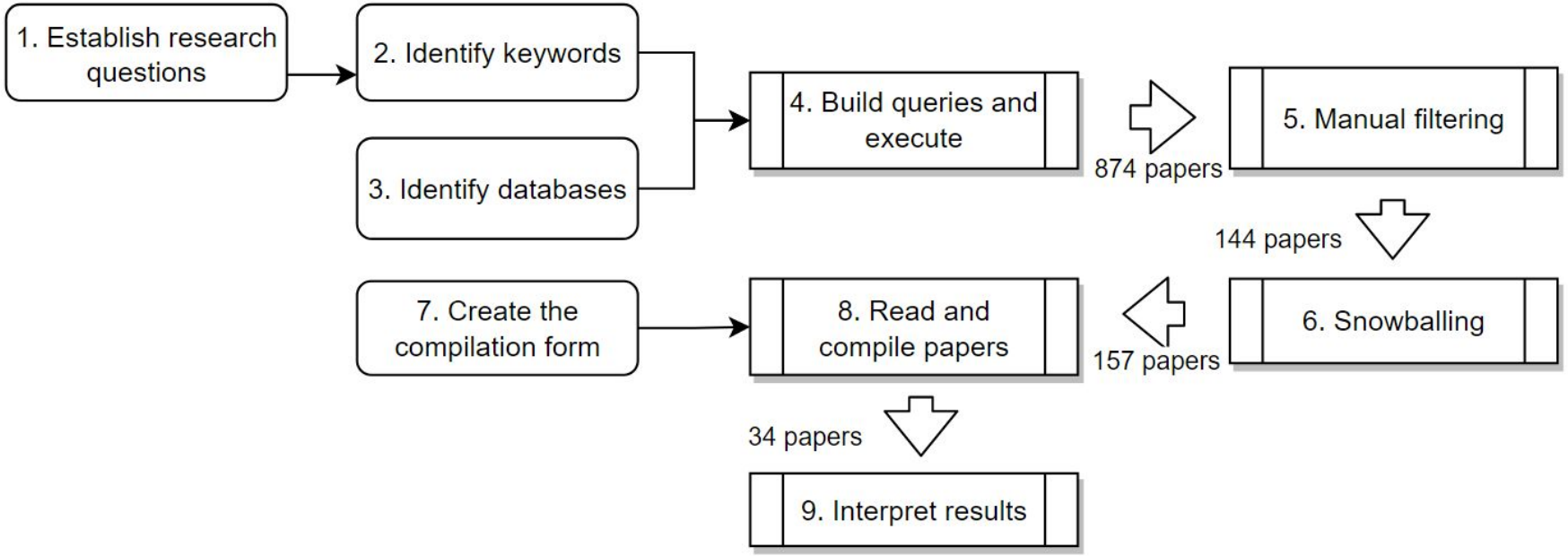
```
"In class MyApplication"

documentClass

    ^ MyDocument

createDocument

    ^ documentClass new
```

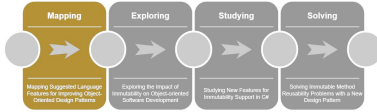


Research Questions

- What language features have been suggested to improve design pattern implementations?
  - Which design patterns?
  - What measures?
  - Empirical experiments?

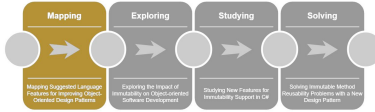
Methodology





Summary of Language Features

Paradigm	Language Feature	# papers	Language
Functional Programming	Case Classes	1	Scala
Functional Programming	Closures	2	Haskell
Functional Programming	Immutability	1	Scala
Meta-Programming	AOP Annotations	2	Java
Meta-Programming	AOP Mixins	2	Java
Meta-Programming	AOP Join-point	17	Java
Meta-Programming	Layer Objects	1	Pseudo-Java
Meta-Programming	Pattern Keywords	2	Java
Meta-programming	Reflection	3	Java
Object-Oriented Programming	Chameleon Objects	1	N/A
Object-Oriented Programming	Class Extension	1	Java
Object-Oriented Programming	Default Implementation	1	N/A
Object-Oriented Programming	Extended Initialization	1	N/A
Object-Oriented Programming	Mixins	3	Java, Scala
Object-Oriented Programming	Multiple Inheritance	1	C++
Object-Oriented Programming	Object Interaction Styles	1	N/A
Object-Oriented Programming	Subclassing members in a subclass	1	N/A
Reactive Programming	Signals	1	Scala



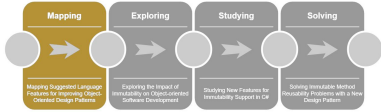
Summary of Language Features

Paradigm	Language Feature	# papers	Language
Functional Programming	Case Classes	1	Scala
Functional Programming	Closures	2	Haskell
Functional Programming	Immutability	1	Scala
Meta-Programming	AOP Annotations	2	Java
Meta-Programming	AOP Mixins	2	Java
Meta-Programming	AOP Join-point	17	Java
Meta-Programming	Layer Objects	1	Pseudo-Java
Meta-Programming	Pattern Keywords	2	Java
Meta-programming	Reflection	3	Java
Object-Oriented Programming	Chameleon Objects	1	N/A
Object-Oriented Programming	Class Extension	1	Java
Object-Oriented Programming	Default Implementation	1	N/A
Object-Oriented Programming	Extended Initialization	1	N/A
Object-Oriented Programming	Mixins	3	Java, Scala
Object-Oriented Programming	Multiple Inheritance	1	C++
Object-Oriented Programming	Object Interaction Styles	1	N/A
Object-Oriented Programming	Subclassing members in a subclass	1	N/A
Reactive Programming	Signals	1	Scala

Layer Objects

Pseudo-Java Layer Objects [84]

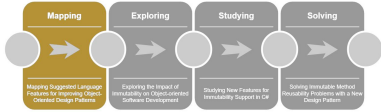
```
class FieldTile {
     def enter(player) {
          // Do something when a player enters.
     }
}

class BurningTileDecorator {
     def damage = 15;
     def FieldTile.enter(player) {
          player.health -= thisLayer.damage;
          proceed(player);
     }
}

// Usage example
def decorator = new BurningTileDecorator();
fieldTile.activate(decorator);
```



Layer Objects

Pseudo-Java Layer Objects [84]

```
class FieldTile {
     def enter(player) {
          // Do something when a player enters.
     }
}

class BurningTileDecorator {
     def damage = 15;
     def FieldTile.enter(player) {
          player.health -= thisLayer.damage;
          proceed(player);
     }
}

// Usage example
def decorator = new BurningTileDecorator();
fieldTile.activate(decorator);
```



Layer Objects

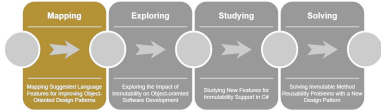
Pseudo-Java Layer Objects [84]

```
class FieldTile {
     def enter(player) {
          // Do something when a player enters.
     }
}

class BurningTileDecorator {
     def damage = 15;
     def FieldTile.enter(player) {
          player.health -= thisLayer.damage;
          proceed(player);
     }
}

// Usage example
def decorator = new BurningTileDecorator();
fieldTile.activate(decorator);
```

Layer Objects

Pseudo-Java Layer Objects [84]

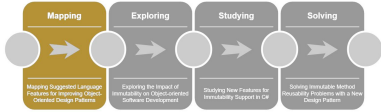
```
class FieldTile {
     def enter(player) {
          // Do something when a player enters.
     }
}

class BurningTileDecorator {
     def damage = 15;
     def FieldTile.enter(player) {
          player.health -= thisLayer.damage;
          proceed(player);
     }
}

// Usage example
def decorator = new BurningTileDecorator();
fieldTile.activate(decorator);
```



Signals

Scala Signals [77]

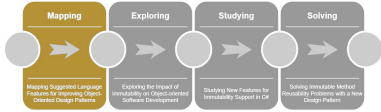
```
val a = Var(1)
val b = Var(2)
val s = Signal { a() + b() }

println(s.getVal()) // 3
a() = 4
println(s.getVal()) // 6
```



Signals

Scala Signals [77]

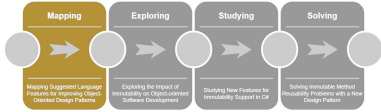
```
val a = Var(1)
val b = Var(2)
val s = Signal { a() + b() }

println(s.getVal()) // 3
a() = 4
println(s.getVal()) // 6
```



Signals

Scala Signals [77]

```
val a = Var(1)
val b = Var(2)
val s = Signal { a() + b() }

println(s.getVal()) // 3
a() = 4
println(s.getVal()) // 6
```

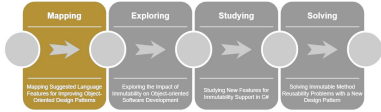


Immutability

- Property of Functional Programming
- Affects design pattern implementations
  - Combined with Closures, makes the Command pattern obsolete
- Examples also include a different State pattern implementation [76]
- Has other potential advantages for software engineering

Immutable Command Example

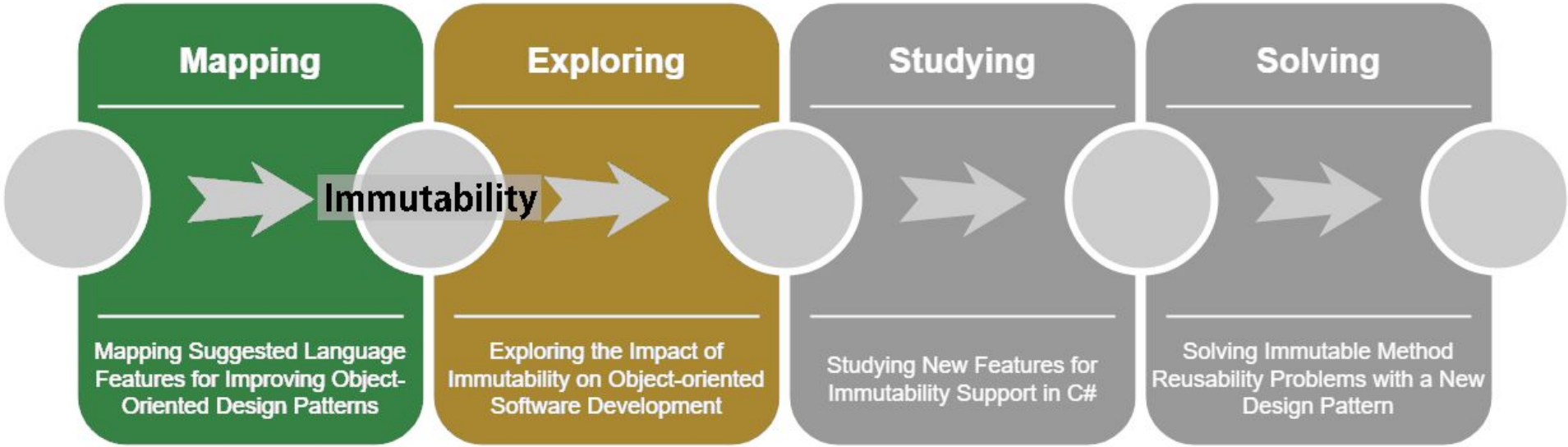
```
Action command = ()
     => Console.WriteLine("Hello World!");
// ...
command();
```



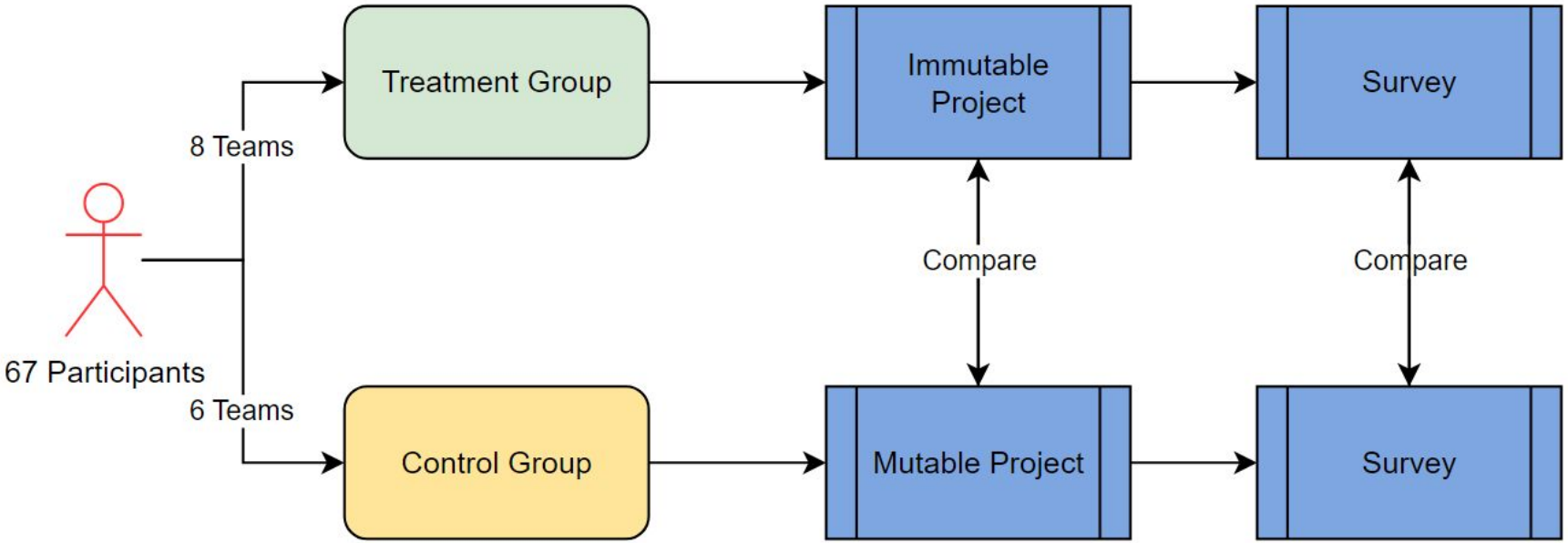
Answer to Research Questions

- Catalogue of 18 language features
- Observer, Visitor, and Decorator
- Maintainability and understandability
  - Chidamber and Kemerer [19]
  - AOP papers focus on concern diffusion
- Mostly descriptive studies
  - Case studies are in-vitro
  - Only one experiment

Thesis Flow

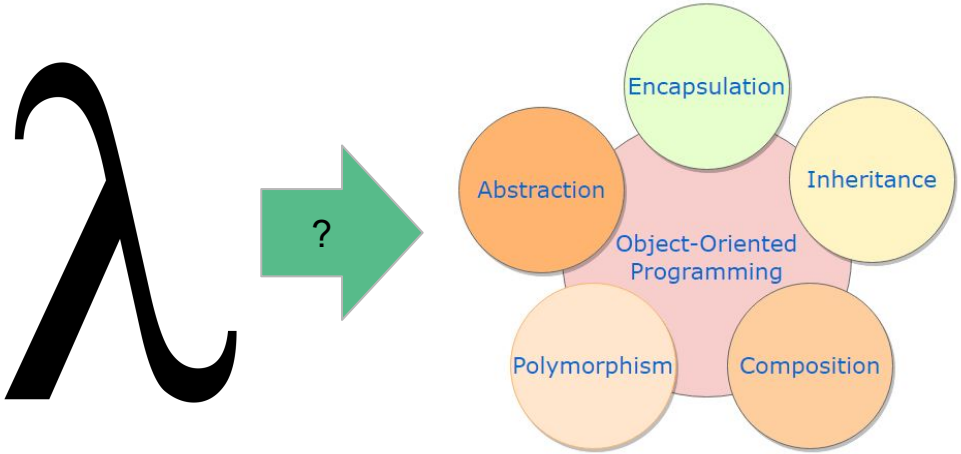

Mapping
Exploring
Studying
Solving
Immutability
Mapping Suggested Language Features for Improving Object-Oriented Design Patterns
Exploring the Impact of Immutability on Object-oriented Software Development
Studying New Features for Immutability Support in C#
Solving Immutable Method Reusability Problems with a New Design Pattern

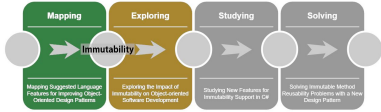
A Multi-Method Exploratory Study



Research Questions

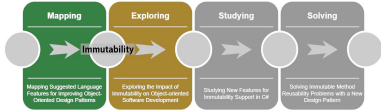
- What is the impact of immutability on object-oriented development?
  - Quantitative
  - Qualitative
  - Comments





Answer to Research Questions

- Quantitative
  - Shorter, more granular methods
  - No significant negative impact of immutability
  - No noticeable impact on performance
- Qualitative
  - Lower workload, lower difficulty, less complex code
- Participants were divided about immutability
  - High learning curve and lack of language support
  - Easier communication among teammates and more understandable programs

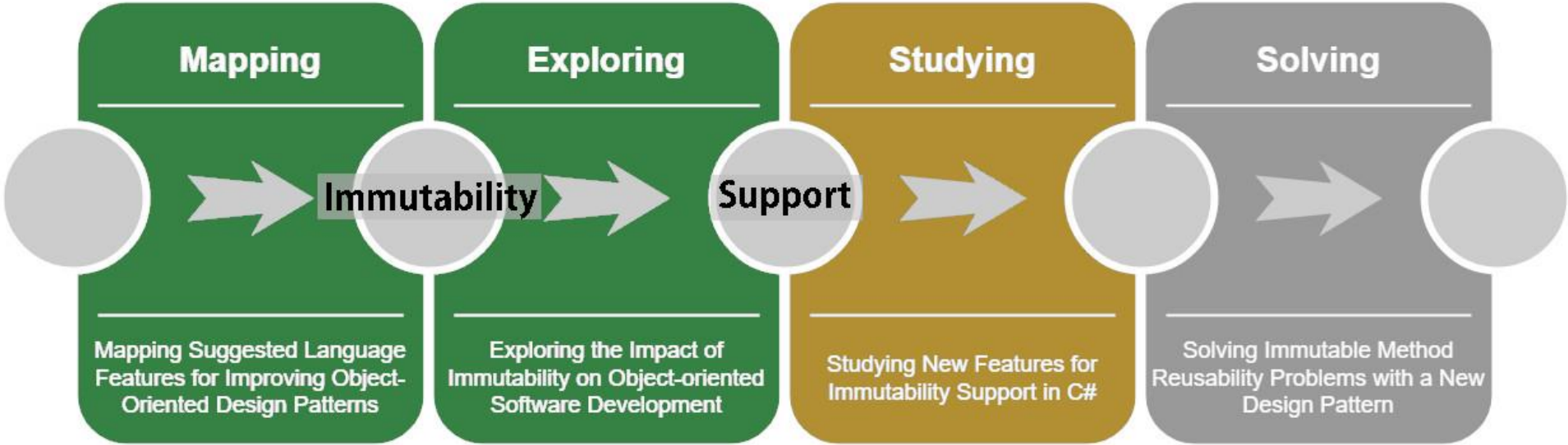


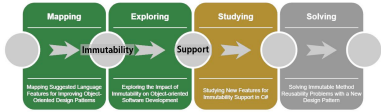
Answer to Main Research Question

What is the impact of immutability on object-oriented development?

- No significant disadvantage observed of using immutability
- Advantages outweigh any disadvantage

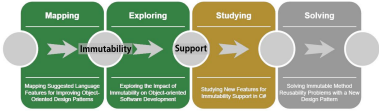
Thesis Flow


Mapping
Exploring
Studying
Solving
Immutability
Support
Mapping Suggested Language Features for Improving Object-Oriented Design Patterns
Exploring the Impact of Immutability on Object-oriented Software Development
Studying New Features for Immutability Support in C#
Solving Immutable Method Reusability Problems with a New Design Pattern

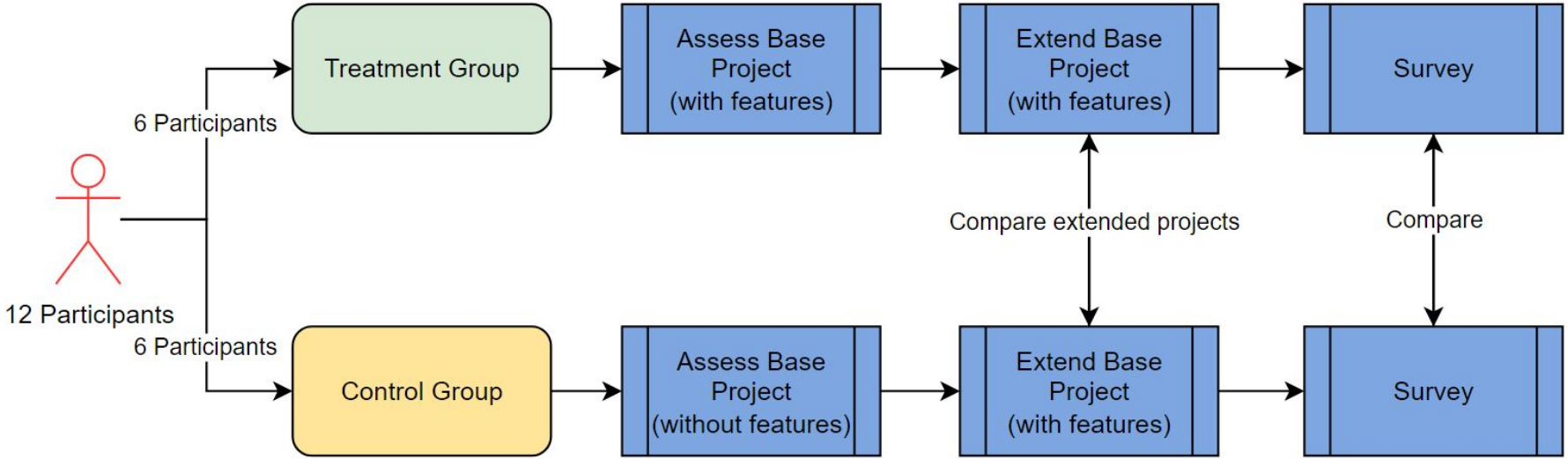


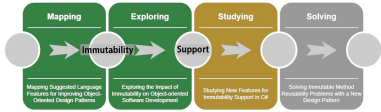
A Multi-Method Empirical Study

- Study language features for immutability support in OOP
- Empirical study on a set of features recently added to C#
  - Record Types and Record Updating
  - Pattern Matching
  - Multiple Values Return



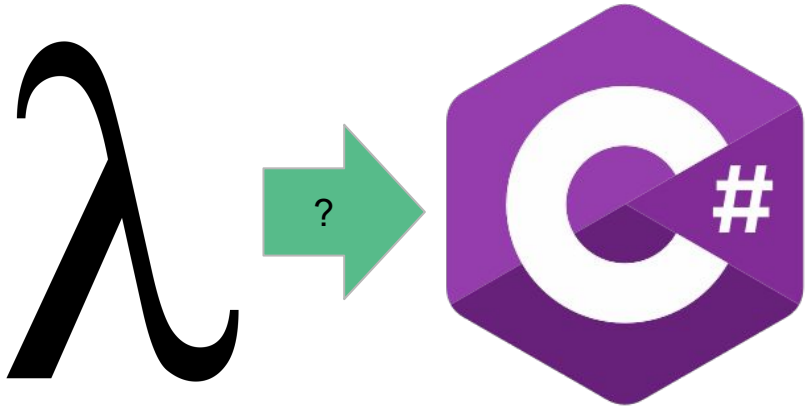
A Multi-Method Empirical Study

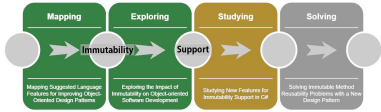




Research Questions

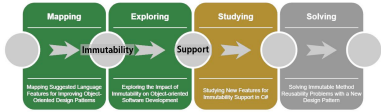
- Do the recently added immutability-related features have a positive impact on writing immutable code in C#?
  - Quantitative
  - Qualitative
  - Differences in code





Answer to Research Questions

- Quantitative
  - Greater maintainability and quality
  - Pattern Matching was particularly effective
- Qualitative
  - No significant difference
- Both groups adopted similar approaches
  - No use of the Visitor pattern

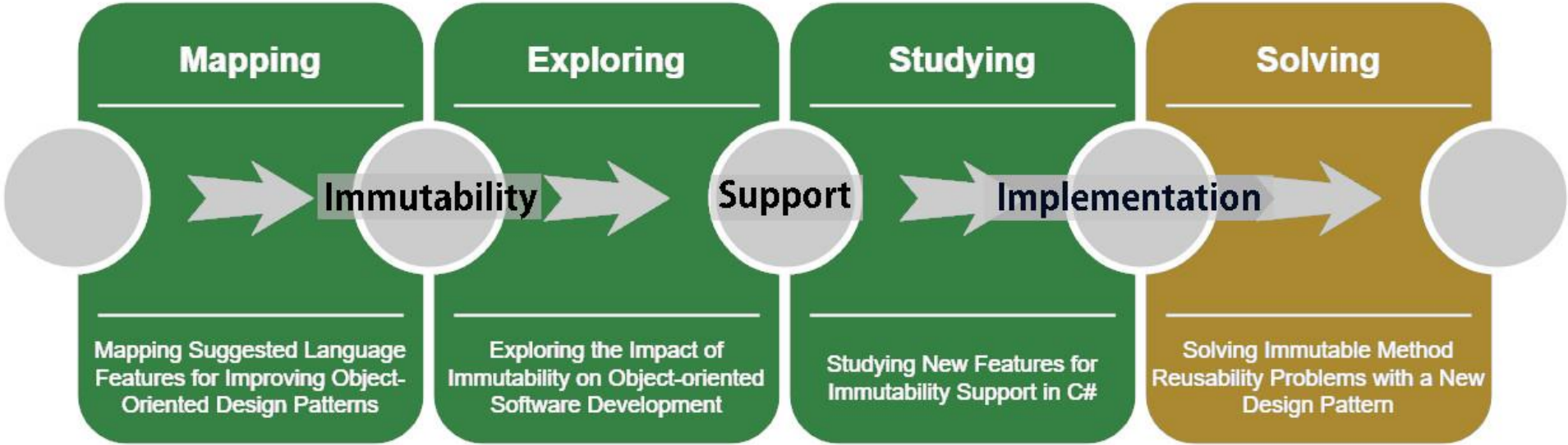


Answer to Main Research Question

Do the recently added immutability-related features have a positive impact on writing immutable code in C#?

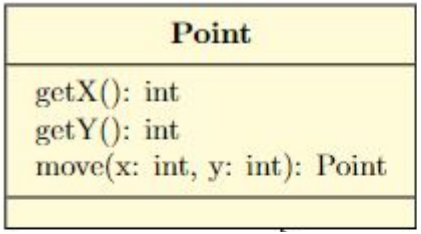
- Supports the approach naturally used by developers
- Improves the quality of immutable code

Thesis Flow


Mapping
Exploring
Studying
Solving
Immutability
Support
Implementation
Mapping Suggested Language Features for Improving Object-Oriented Design Patterns
Exploring the Impact of Immutability on Object-oriented Software Development
Studying New Features for Immutability Support in C#
Solving Immutable Method Reusability Problems with a New Design Pattern

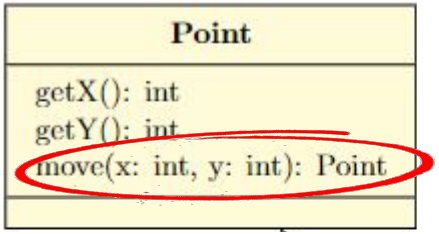
Non-destructive mutators

- Immutable version of mutator
  - Creates a new object
  - Does not modify the original



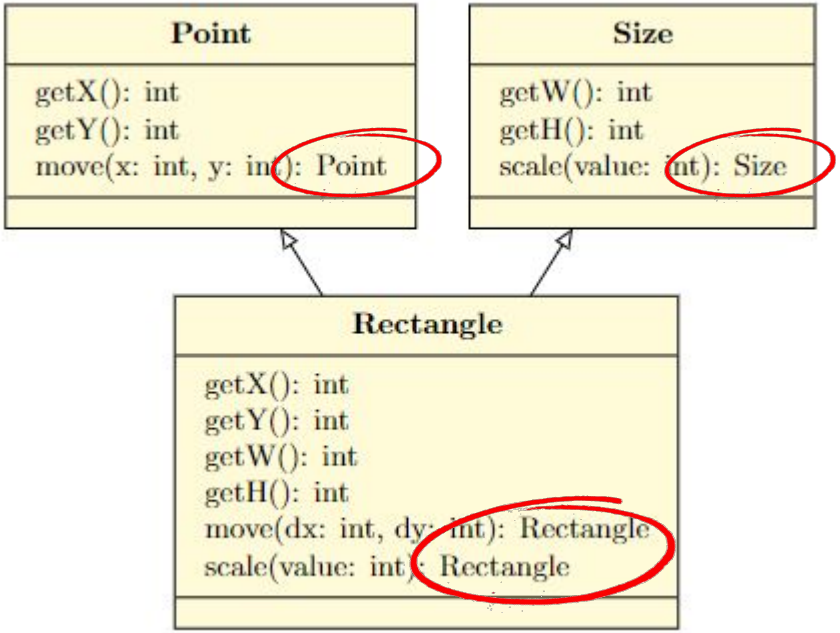
Non-destructive mutators

- Immutable version of mutator
  - Creates a new object
  - Does not modify the original



Non-destructive mutators

- Immutable version of mutator
  - Creates a new object
  - Does not modify the original
  - **What about polymorphism?**



Main Research Question

- Is it possible to reuse non-destructive mutators via polymorphism when combining immutability and OOP subtyping?
  - Inheritance vs. composition
  - Return type polymorphism

Problem 1: Inheritance, Composition, and Immutability

- "Is-a" relationship between Rectangle, Point, and Size
  - Use composition instead?
- Multiple inheritance?
  - Unsupported in Java/C#

Composition Problem

```
Rectangle r = new Rectangle(1, 3, 2, 2);
Point p = r.getPosition().move(1, 2);

Rectangle r2
     = new Rectangle(p.x, p.y, r.w, r.h);
```

Problem 2: Return Type Polymorphism

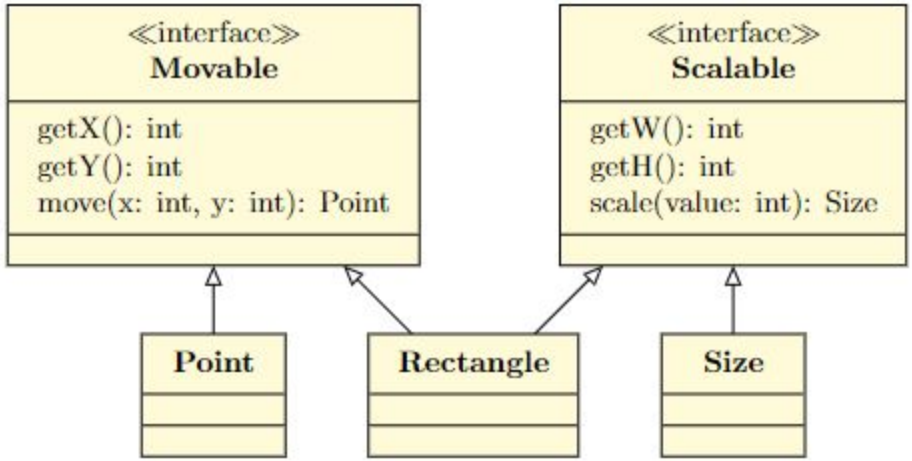
- Methods “move” and “scale” must return a new object of the correct type
- Using generic programming:

Generic Method Definition

```
static T move<T>(T movable, int x, int y) { ... }
```

- We still need a way to create the new object
  - Cannot call the constructor of a generic type

Solution: The Immutable Factory Method



Addressing Problem 1: Subtyping

- We use subtyping with interfaces instead of inheritance
  - No multiple inheritance in Java
  - Must define reusable methods
  - Java 8 supports default methods in interfaces

Default Method Implementation

```
interface Movable {
    int getX();
    int getY();

    default Point move(int dx, int dy) {
        return new Point(this.getX() + x, this.getY() + y);
    }
}
```

Addressing Problem 1: Subtyping

- We use subtyping with interfaces instead of inheritance
  - No multiple inheritance in Java
  - Must define reusable methods
  - Java 8 supports default methods in interfaces

Default Method Implementation

```
interface Movable {
    int getX();
    int getY();

    default Point move(int dx, int dy) {
        return new Point(this.getX() + x, this.getY() + y);
    }
}
```

Addressing Problem 1: Subtyping

- We use subtyping with interfaces instead of inheritance
  - No multiple inheritance in Java
  - Must define reusable methods
  - Java 8 supports default methods in interfaces

Default Method Implementation

```
interface Movable {
    int getX();
    int getY();

    default Point move(int dx, int dy) {
        return new Point(this.getX() + x, this.getY() + y);
    }
}
```

Addressing Problem 2: Return Type Polymorphism

- Cannot call a constructor on a generic type

Factory Method Definition

```
T updateMovable(int x, int y);
```

Factory Method Implementation

```
final class Point implements Movable<Point> {
    // ...

    @Override
    public Point updateMovable(int x, int y) {
        return new Point(x, y);
    }
}
```

Addressing Problem 2: Return Type Polymorphism

- Cannot call a constructor on a generic type

Factory Method Definition

```
T updateMovable(int x, int y);
```

Factory Method Implementation

```
final class Point implements Movable<Point> {
    // ...

    @Override
    public Point updateMovable(int x, int y) {
        return new Point(x, y);
    }
}
```

Rectangle Implementation of the Factory Method

```
final class Rectangle
    implements Movable<Rectangle>, Scalable<Rectangle> {
    // ...

    @Override
    public Rectangle updateMovable(int x, int y) {
        return new Rectangle(x, y, this.getW(), this.getH());
    }

    @Override
    public Rectangle updateScalable(int w, int h) {
        return new Rectangle(this.getX(), this.getY(), w, h);
    }
}
```

Rectangle Implementation of the Factory Method

```
final class Rectangle
    implements Movable<Rectangle>, Scalable<Rectangle> {
    // ...

    @Override
    public Rectangle updateMovable(int x, int y) {
        return new Rectangle(x, y, this.getW(), this.getH());
    }

    @Override
    public Rectangle updateScalable(int w, int h) {
        return new Rectangle(this.getX(), this.getY(), w, h);
    }
}
```

Client Code

```
Point p = new Point(2, 2);
Point p2 = p.move(1, 2); // 3, 4

Size s = new Size(4, 6);
Size s2 = s.scale(3); // 12, 18

Rectangle r = new Rectangle(2, 2, 3, 4);
Rectangle r2 = r.move(1, 2); // 3, 4, 3, 4
Rectangle r3 = r.scale(5); // 2, 2, 12 18
```

Design Pattern Drawbacks

- Added complexity
  - Default Methods
  - Generic Programming
  - Factory Method
- “Boilerplate” code
  - UpdateMovable/UpdateScalable
  - Must be updated if classes change

Clojure Variant

- Dynamically-typed language
- Functional updating support
- Idiomatic solution:

Clojure Move Implementation

```
(defn make-point [x y]
    {:x x :y y})

(defn make-rectangle [x y w h]
    {:x x :y y :w w :h h})

(defn move [point dx dy]
    (assoc point
           :x (+ (point :x) dx)
           :y (+ (point :y) dy)))
```

Clojure Variant

- Dynamically-typed language
- Functional updating support
- Idiomatic solution:

Clojure Move Implementation

```
(defn make-point [x y]
    {:x x :y y})

(defn make-rectangle [x y w h]
    {:x x :y y :w w :h h})

(defn move [point dx dy]
    (assoc point
           :x (+ (point :x) dx)
           :y (+ (point :y) dy)))
```

Clojure Variant

- Dynamically-typed language
- Functional updating support
- Idiomatic solution:

Clojure Move Implementation

```
(defn make-point [x y]
    {:x x :y y})

(defn make-rectangle [x y w h]
    {:x x :y y :w w :h h})

(defn move [point dx dy]
    (assoc point
           :x (+ (point :x) dx)
           :y (+ (point :y) dy)))
```

Key Features

- Functional updating
  - Construct the new object of the proper type
  - Available in: Kotlin, Rust, C#
- Dynamic typing
  - Simplified method definitions
  - Alternative is to use generic programming

Functional Updating in Clojure

```
(assoc point
       :x (+ (point :x) dx)
       :y (+ (point :y) dy))
```

Generic Default Method in Java

```
default T move(int dx, int dy) {
    return updateMovable(this.getX() + x, this.getY() + y);
}
```

Common Lisp Variant

- Dynamically-typed OOP language (using CLOS)
- No native functional updating support

Common Lisp Types

```
(defclass point ()
  ((x :reader x :initarg :x)
   (y :reader y :initarg :y)))

(defclass size ()
  ((w :reader w :initarg :w)
   (h :reader h :initarg :h)))

(defclass rectangle (point size) ())
```

Common Lisp Variant

- Dynamically-typed OOP language (using CLOS)
- No native functional updating support

Common Lisp Types

```
(defclass point ()
  ((x :reader x :initarg :x)
   (y :reader y :initarg :y)))

(defclass size ()
  ((w :reader w :initarg :w)
   (h :reader h :initarg :h)))

(defclass rectangle (point size) ())
```

CLOS Movable Methods Implementation

```
(defgeneric update-movable (movable x y))

(defmethod update-movable ((movable point) x y)
  (make-instance 'point :x x :y y))

(defmethod update-movable ((movable rectangle) x y)
  (make-instance 'rectangle
                 :x x :y y
                 :w (w movable) :h (h movable)))

(defun move (movable dx dy)
  (update-movable movable
        (+ dx (x movable))
        (+ dy (y movable))))
```

CLOS Movable Methods Implementation

```
(defgeneric update-movable (movable x y))

(defmethod update-movable ((movable point) x y)
  (make-instance 'point :x x :y y))

(defmethod update-movable ((movable rectangle) x y)
  (make-instance 'rectangle
                 :x x :y y
                 :w (w movable) :h (h movable)))

(defun move (movable dx dy)
  (update-movable movable
        (+ dx (x movable))
        (+ dy (y movable))))
```

CLOS Movable Methods Implementation

```
(defgeneric update-movable (movable x y))

(defmethod update-movable ((movable point) x y)
  (make-instance 'point :x x :y y))

(defmethod update-movable ((movable rectangle) x y)
  (make-instance 'rectangle
                 :x x :y y
                 :w (w movable) :h (h movable)))

(defun move (movable dx dy)
  (update-movable movable
        (+ dx (x movable))
        (+ dy (y movable))))
```

Extending Common Lisp

- Meta-programming support
  - Allows the use of macros to extend the language
- Let us add functional updating support to the language!
  - The macro code would be distributed in a library
  - Invisible to the user

Common Lisp Mutable Implementation

Move and Scale Mutable Implementations

```
(defun move (movable dx dy)
  (with-slots (x y) movable
    (incf x dx)
    (incf y dy)))

(defun scale (scalable scale)
  (with-slots (w h) scalable
    (setf w (* scale w))
    (setf h (* scale h))))
```

Common Lisp Mutable Implementation

Move and Scale Mutable Implementations

```
(defun move (movable dx dy)
  (with-slots (x y) movable
    (incf x dx)
    (incf y dy)))

(defun scale (scalable scale)
  (with-slots (w h) scalable
    (setf w (* scale w))
    (setf h (* scale h))))
```

Extended Common Lisp Implementation

- No more need for the Factory Method
  - Can directly implement the move and scale methods
- Idiomatic code
  - Same as if using “with-slots”

Move and Scale Implementations

```
(defun move (movable dx dy)
  (with-new (x y) movable
    (incf x dx)
    (incf y dy)))

(defun scale (scalable scale)
  (with-new (w h) scalable
    (setf w (* scale w))
    (setf h (* scale h))))
```

Extended Common Lisp Implementation

- No more need for the Factory Method
  - Can directly implement the move and scale methods
- Idiomatic code
  - Same as if using “with-slots”

Move and Scale Implementations

```
(defun move (movable dx dy)
  (with-new (x y) movable
    (incf x dx)
    (incf y dy)))

(defun scale (scalable scale)
  (with-new (w h) scalable
    (setf w (* scale w))
    (setf h (* scale h))))
```

Extended Common Lisp Implementation

- No more need for the Factory Method
  - Can directly implement the move and scale methods
- Idiomatic code
  - Same as if using “with-slots”

Move and Scale Implementations

```
(defun move (movable dx dy)
  (with-new (x y) movable
    (incf x dx)
    (incf y dy)))

(defun scale (scalable scale)
  (with-new (w h) scalable
    (setf w (* scale w))
    (setf h (* scale h))))
```

Idiomatic Mutable Implementations

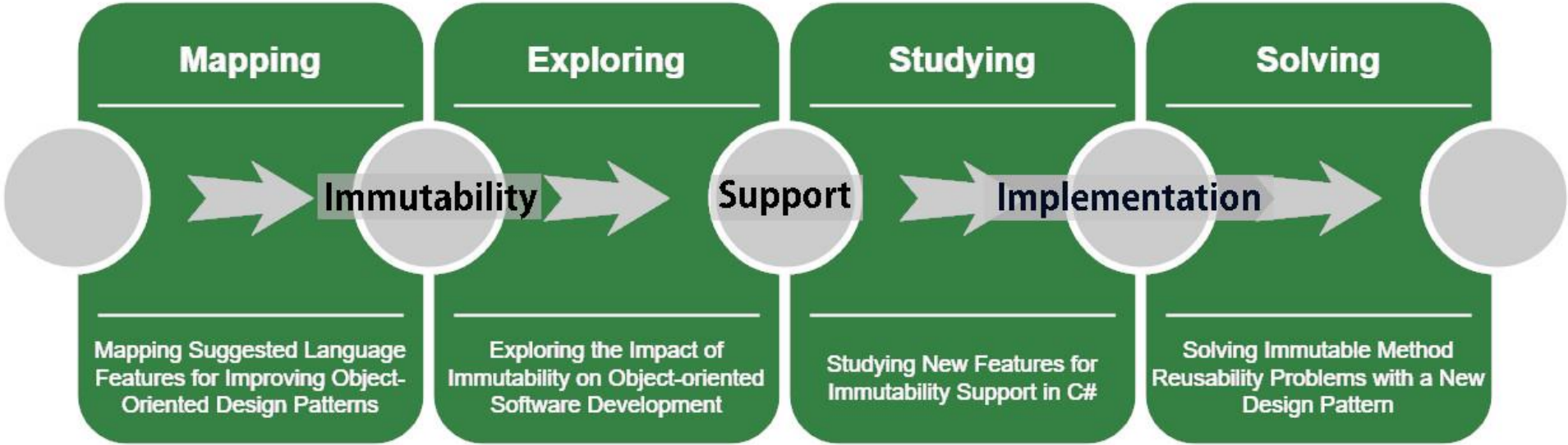
```
(defun move (movable dx dy)
  (with-slots (x y) movable
    (incf x dx)
    (incf y dy)))

(defun scale (scalable scale)
  (with-slots (w h) scalable
    (setf w (* scale w))
    (setf h (* scale h))))
```

Answer to Main Research Question

Is it possible to re-use non-destructive mutators via polymorphism when combining immutability and OOP subtyping?

- New design pattern to circumvent the problem in any OOP language
- Key features
  - Functional Updating
  - Dynamic Typing

Thesis Flow


Mapping
Exploring
Studying
Solving
Immutability
Support
Implementation
Mapping Suggested Language Features for Improving Object-Oriented Design Patterns
Exploring the Impact of Immutability on Object-oriented Software Development
Studying New Features for Immutability Support in C#
Solving Immutable Method Reusability Problems with a New Design Pattern

Contributions

- A catalogue of 18 language features suggested to improve OOP design pattern implementations
- An exploratory study on the impact of immutability on OOP
- An empirical study on the impact of adding immutability-related language features to C#
- A new design pattern to solve a problem that emerges with the combination of OOP and immutability
- An extension to Common Lisp that adds functional updating to the language

Threats to Validity

- Internal Validity
  - Student participants
  - Potential bias towards immutability
- External Validity
  - Hawthorne effect
- Construct Validity
  - Structure of the C# study
  - Overall difficulty of the studies
- Conclusion Validity
  - Avoided strong conclusions concerning statistical results

Thesis Statement

- OOP can be improved by adding language features that solve the underlying issue behind design patterns
- Among many possibilities, we focused on immutability
  - Can increase understandability and granularity of the code
  - Immutability features can improve maintainability, reduce code duplication, and improve scalability

Short Term: Language Features to Improve OOP

- Aspect-Oriented Programming
- Case Classes
- Chameleon Objects
- Class Extension
- Closures
- Default Implementation
- Immutability
- Layer Objects
- Mixins
- Multiple Inheritance
- Object Interaction Styles
- Pattern Keywords
- Reflection
- Signals
- Subclassing members in a subclass

Short Term: Language Features to Improve OOP

- Aspect-Oriented Programming
- Case Classes
- Chameleon Objects
- Class Extension
- Closures
- Default Implementation
- Immutability
- Layer Objects
- Mixins
- Multiple Inheritance
- Object Interaction Styles
- Pattern Keywords
- Reflection
- Signals
- Subclassing members in a subclass

Short Term: Language Features to Improve OOP

- Aspect-Oriented Programming
- Case Classes
- Chameleon Objects
- Class Extension
- Closures
- Default Implementation
- Immutability
- Layer Objects
- Mixins
- Multiple Inheritance
- Object Interaction Styles
- Pattern Keywords
- Reflection
- Signals
- Subclassing members in a subclass

Short Term: Rust Ownership System

- Isolate mutating parts of a program
- Studying how this system interacts with structural design patterns
  - Composite pattern seems interesting

Short Term: Pattern Matching, Multimethods, and Visitor

- Pattern Matching can replace Visitor in some situations
- Multimethod, or multiple dispatch, can also replace the Visitor
- We want to study the interactions between Pattern Matching, Multimethods, and the Visitor design pattern

Mid Term: Individual Design Pattern Studies

- Each feature in our catalogue maps to specific design patterns
- Focus on specific design patterns and which features impact them
  - Improve pattern by combining features
  - Solve the underlying problem of the pattern
- Categorize patterns by most “solvable”

Mid Term: Replications

- Quasi-replication of our exploratory study
  - Impact of immutability on code granularity and understandability
  - Compare the workload between mutable and immutable software development
- Replicate the C# study with only professional developers
  - Study could consider other languages or features, such as C# LINQ or Java Streams

Long Term: A New Paradigm?

- Combination of FP and OOP?
- Can a pattern-less language exist?
  - Resolve all underlying issues?
  - Architectural patterns vs. code patterns
  - Formalization vs Obsolescence
- What about generative machine learning?
  - Impact on programming languages
  - Impact on design patterns

Publications

- William Flageol, Éloi Menaud, Yann-Gaël Guéhéneuc, Mourad Badri, Stefan Monnier, "**A mapping study of language features improving object-oriented design patterns**", Information and Software Technology, Volume 160, 2023.

- William Flageol, Yann-Gaël Guéhéneuc, Mourad Badri, and Stefan Monnier, "**A Multimethod Exploratory Study on the Impact of Immutability on Object-oriented Software Development**", under revision.

- William Flageol, Yann-Gaël Guéhéneuc, Mourad Badri, and Stefan Monnier, "**A Multimethod Empirical Study on New Features for Immutability Support in C#**", submitted to Journal of Systems and Software.

- William Flageol, Yann-Gaël Guéhéneuc, Mourad Badri, and Stefan Monnier, "**Design Pattern for Reusing Immutable Methods in Object-Oriented Languages**", EuroPLoP '23: 28th European Conference on Pattern Languages of Programs, July 05–09, 2023, Kloster Irsee, Germany.

Pilot Study

- We performed a pilot study prior to the experiment
  - The goal was to assess the feasibility of the experiment
- Participants included two recently graduated Ph.D. students
- The initial structure of the experiment required the participant to develop the full program by themselves following Bloch's transitive immutability rules
- The feedback from the pilot study indicated this was too big a task to ask of volunteer participants
- We redesigned the experiment so that the participant would evaluate a base program and extend it instead

Experiment

- There are two base programs, one per group
  - The treatment group program uses the new features
  - The control group program does not
  - Other than these differences, the two base programs are similar in functionality
- The base programs are file system simulators
  - Allows creating a hierarchy of folders and files
- We ask the participants to extend the base programs by adding the following functionality:
  - Collect operation
  - Undo operation
  - Duplicate operation
- At the end, we asked the participants to fill out a survey about their experience
- Interview with the professional developers

Chameleon Objects

- OOP feature which allows changing the class of an object at run-time
  - Already exists in some languages
  - Common Lisp, Perl
- Can implement State by changing classes when state changes
- Examples also include implementing Factory Method

Chameleon Objects in Common Lisp

```
(change-class target-object target-class)
```

Chameleon Objects in Perl

```
bless $targetObject, 'Package::TargetClass';
```

Pattern Keywords

- Some studies introduce design patterns directly as new keywords
- Can be done without the help of third party-tools in languages with meta-programming support
  - MzScheme implementation of the Visitor as a keyword by Krishnamurthi et al.
- Examples include implementing Decorator, Observer, Singleton and Visitor
- Whether design patterns should be language features is debatable

Java Singleton Keyword

```
public singleton class A
{
      instantiate A as s1;

      public static void main(String[] args)
      {
            instantiate A as s2, s3;
      }
}
```

Case Classes

- Feature of Functional Programming
  - Also known as Pattern Matching
- Shares functionality with object polymorphism
- Can be used to implement multimethods
- Examples include a Visitor implementation

Scala Case Classes

```
// Visitor structure for a Binary Tree.
trait Tree {
      def accept[R] (v :TreeVisitor[R]):R
}
case class Empty extends Tree {
      def accept[R] (v :TreeVisitor[R]):R = v.empty
}
case class Fork (x :int,l : Tree,r: Tree) extends Tree {
      def accept[R] (v :TreeVisitor[R]):R = v.fork (x,l,r) }

trait TreeVisitor[R] {
      def empty :R
      def fork (x : int,l :Tree,r: Tree):R
}

// Concrete implementation of visitor to calculate
// the depth of the Tree.
def depth = new CaseTree [External,int] {
      def Empty = 0
      def Fork (x : int,l :R[TreeVisitor],r:R[TreeVisitor])
            = 1+max (l.accept (this),r.accept (this))
}
```

Aspect-Oriented Programming

- Introduced in 1997 by Kiczales et al.
- Paradigm extension to procedural programming
  - Increases modularity by encapsulating cross-cutting concerns into Aspects
- Three main ways to implement :
  - The Join-Point model
  - Annotations
  - Mixins
- Presents full implementations of the 23 GoF patterns

Join-Point model

```
pointcut setter(): target(Point) &&
     (call(void setX(int)) ||
      call(void setY(int)));
```

Annotations

```
[NotifyPropertyChanged]
public class Person
{
     public string FirstName { get; set;
}
     public string LastName { get; set; }
     public Address Address { get; set; }
}
```

Closures

- Main feature of Functional Programming and lambda-calculus
  - Encapsulates behaviour and data
- Difference with OOP classes
  - Can only encapsulate one function
  - Data usually cannot be accessed from outside
- Supported by most modern languages
  - C++, C#, Java, Python, JavaScript, Kotlin, etc.
- Examples include implementing Command, Composite, Iterator, Visitor, and Builder

Lambda Syntax Example

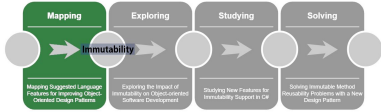
```
var i = 42
var closure = (argument) => {
     // Some code which can
     // use the i variable.
}
```

Mixins

- Extension of OOP
- Unlike inheritance, do not enforce a “is-a” relationship
- Examples include implementations of Decorator, Proxy, Chain of Responsibility, and Strategy

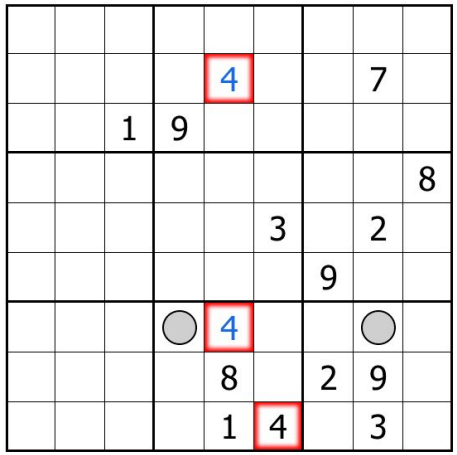
Mixins in Java

```
class Component { Operation(); }

class ConcreteComponent implements Component { ... };

class DecoratorMixA implements Component
            needs Component
      Operation() { ... Component.Operation() };

class DecoratorMixB implements Component
            needs Component
      Operation() { ... Component.Operation() };

// Usage
class Client {
      main() {
            ConcreteComponent cc =
                  new ConcreteComponent with DecoratorMixA;
            extend cc with DecoratorMixB;
            cc.Operation();
      }
}
```



Project

- Development of a Sudoku solver
- Two phases
  - Application core
    - SOLID and GRASP principles
  - Add new functionality
    - GoF design patterns
- Both groups had identical requirements



Databases for the Search Query

- We use multiple online databases to search for primary studies
  - The databases we used are part of an online tool called Engineering Village, hosted by Elsevier
  - They contain studies from many scientific journal databases, including ACM and IEEE
- We restrain our search to studies published after 1995

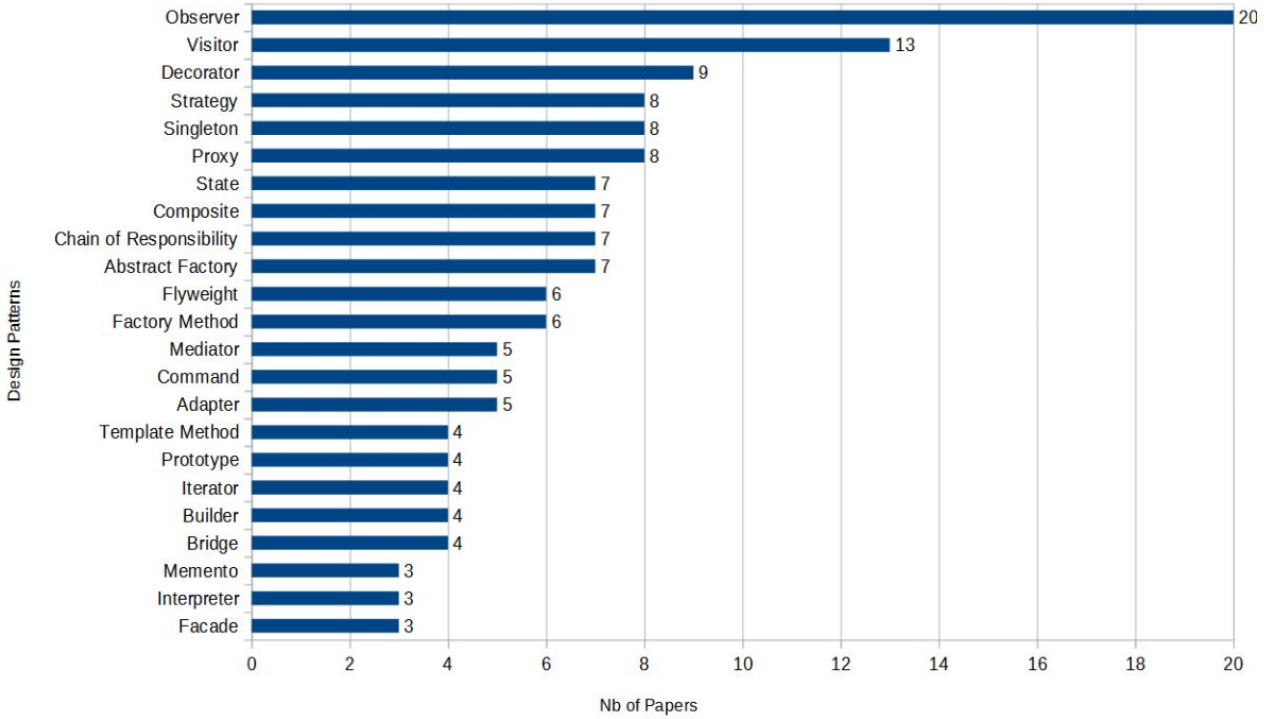
Database Query

Query	Comments
((oop OR object) AND (design pattern OR design patterns OR anti-pattern OR anti-patterns OR paradigm) AND (weakness* OR disadvantage* OR improve* OR enhance* OR refine* OR help OR better OR expand*) WN AB)	 Must be linked to design patterns Initially, we intended the scope to be broader, but this shouldn't impact the final results Must be about improving or finding weaknesses These keywords are located in abstract text
NOT ((teach* OR learn* OR test* OR modeling OR automated OR automation OR tool* OR mobile OR optimiz* OR simulation OR mining OR medical OR bio* OR hardware OR hdl OR parallel* OR api OR find* OR sql) WN KY)	These keywords are excluded to avoid papers about learning, automating, optimizing tools, or other non-related subjects encountered during the first passes of the query. Excluded keywords must not appear in the keyword field.
NOT ((pattern recognition OR pattern identification OR pattern detection OR object recognition OR object detection OR feature extraction OR computer vision OR pattern clustering OR learning (artificial intelligence) OR image segmentation OR pattern classification OR data mining OR computer aided design OR formal specification OR image classification OR user interfaces OR computer simulation OR internet OR image processing OR codes (symbols) OR image enhancement OR embedded systems OR image reconstruction OR cameras OR distributed computer systems OR product design OR artificial intelligence OR iterative methods OR neural nets OR neural networks OR object tracking OR genetic algorithms OR classification (of information) OR learning systems OR mathematical models OR middleware OR internet of things OR cloud computing OR decision making OR electroencephalography OR virtual reality OR risk management OR health care OR distributed object management OR query processing OR knowledge based systems) WN KY)	Finally, a large list of expressions was excluded from the keywords because they tended to be attached to papers about concepts unrelated to this research.

Results by Publication Venue

Publication Venue [# Papers]
Conference on Pattern Languages of Programs (PLoP) [4 papers]
European Conference on Object-Oriented Programming (ECOOP) [2 papers]
Special Interest Group on Programming Languages (SIGPLAN) [2 papers]
Symposium on Applied Computing (SAC) [2 papers]
Computer Languages, Systems & Structures [1 paper]
International Conference on Advanced Communication Control and Computing Technologies (ICACCCT) [1 paper]
International Conference on Computer Science and Information Technology (CSIT) [1 paper]
International Conference on Informatics and Systems (INFOS) [1 paper]
International Conference on Objects, Components, Models and Patterns [1 paper]
International Conference on Software Engineering Advances (ICSEA) [1 paper]
International Conference on Software Technology and Engineering (ICSE) [1 paper]
International Conference on Systems, Programming, Languages and Applications: Software for Humanity [1 paper]
International Journal of Software Engineering and Knowledge Engineering [1 paper]
International Journal of Software Innovation [1 paper]
International Symposium on Foundations of Software Engineering (FSE) [1 paper]
International Workshop on Context-Oriented Programming [1 paper]
Journal of Object Technology [1 paper]
Journal of Software [1 paper]
Journal of Systems and Software [1 paper]
Journal of the Brazilian Computer Society [1 paper]
Lecture Notes in Computer Science [1 paper]
Second Eastern European Regional Conference on the Engineering of Computer Based Systems [1 paper]
Software Engineering and Knowledge Engineering (SEKE) [1 paper]
Software Technology and Engineering Practice (STEP) [1 paper]
Software: Practice and Experience [1 paper]
Workshop on Aspects, components, and patterns for infrastructure software (ACP4IS) [1 paper]
Workshop on Generic Programming (WGP) [1 paper]
Workshop on Reuse in Object-Oriented Information Systems Design [1 paper]

Number of papers by design pattern



Measures

- Top measures used to evaluate the improvements by the suggested design patterns include
  - Depth of inheritance (DIT)
  - Coupling and cohesion (CBC, LCOO)
  - Concern diffusion (CDC, CDLoC, CDO)
  - Code size (LoC, NoA, WoC)
  - Reuseability

Programs Summary

Team #	Group	Language	Grade
1	Immutable	TypeScript	100
2	Immutable	C#	100
3	Immutable	C#	93
4	Immutable	C#	100
6	Immutable	C#	100
7	Immutable	C#	96
9	Immutable	C#	67
11	Immutable	C#	100
5	Mutable	Unity C#	90
8	Mutable	C#	100
10	Mutable	C#	88
12	Mutable	C#	100
13	Mutable	Java	85
14	Mutable	C#	56
Average (All)			91.07
Average (Mutable)			86.5
Average (Immutable)			94.5

Statistical Tests

- Since the dataset was not normally distributed, we opted to perform a Mann Whitney U test to test if the groups were significantly different
  - We used Cliff's Delta to measure effect sizes
  - Because of the large amount of measures, we cannot compare to the typical p-value threshold of 0.05
  - A Bonferroni correction would give a threshold of 0.006

Experiments and Participation

- Out of the 34 primary studies
  - 10 were case studies
  - 1 was a controlled experiment
  - 23 were descriptive studies
- The case studies were all done in-vitro
  - It is difficult to find projects developed by third parties using new tools or methods
- The experiment was done with 38 undergraduate students
- In general, industry practitioners are not involved in these efforts

Data Collection

- We used SciTools Understand to collect quantitative measures on the programs
- We collected measures related to many categories
  - Class Complexity
  - Method Size
  - Class interactions
  - Class Size
  - Class Cohesion and Nesting
  - Program Size

Survey

- We had the participants fill out a survey to collect data about their experience with the project
- The survey asked the following questions:
  - Auto-evaluate your expertise in OOP (1 to 5)
  - Auto-evaluate your participation in the project within their team (1 to 5)
  - Give your impression on the workload, difficulty, and complexity of the resulting program for each of the following (1 to 5):
    - Phase 1 implementation
    - Phase 2 implementation
    - SOLID/GRASP principle implementations
    - GoF patterns implementations
  - Would you consider using immutability for future projects? (yes or no)
- The participants also had the option to leave some comments

Participants

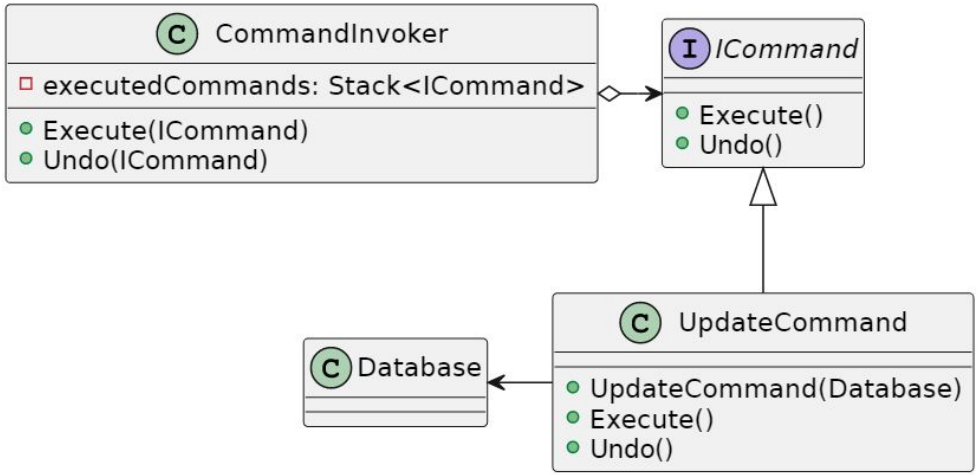
- The experiment was part of a B.Sc. software engineering class on advanced OOP
  - The participants all had a similar academic background, with experience mainly in Java and C#
  - The main focus of the class was design patterns
  - About 9 hours out of 45 were dedicated to immutability
  - No other emphasis was put on immutability during the class
  - All participants were taught the same material
- None of the participants reported any prior experience with immutability
- Participation was voluntary
  - We gave incentives in the form of extra credits (10% of the total grade for the assignment)
  - Out of the 84 students, 67 chose to participate

Program Measures

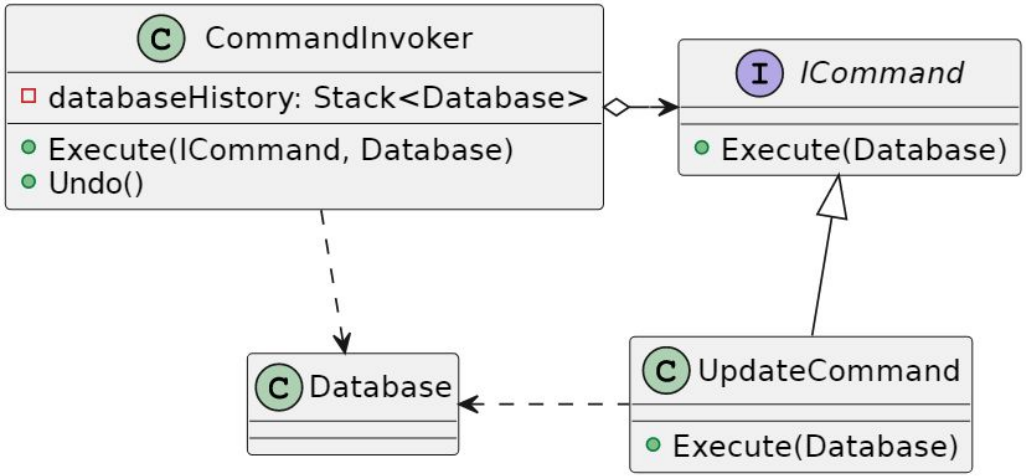
Measure	Avg	Avg (I)	Avg (M)
AvgCycl (Avg)	1.4038	1.4624	1.3257
AvgCycl (Max)	5.2143	5.5000	4.8333
AvgCycl* (Avg)	1.3454	1.3864	1.2908
AvgCycl* (Max)	4.4286	4.3750	4.5000
AvgCyclStrict (Avg)	1.6905	1.9074	1.4014
AvgCyclStrict (Max)	7.5000	9.0000	5.5000
AvgEss (Avg)	0.9466	0.9699	0.9156
AvgEss (Max)	1.7143	1.5000	2.0000
NbClassBase (Avg)	1.2545	1.2736	1.2290
NbClassBase (Max)	2.0000	2.0000	2.0000
CBO (Avg)	6.9152	7.2760	6.4943
CBO (Max)	27.3846	26.5714	28.3333
CBO* (Avg)	3.0608	2.5215	3.6899
CBO* (Max)	12.1538	9.5714	15.1667
NOC (Avg)	0.1297	0.1292	0.1303
NOC (Max)	2.0714	2.1250	2.0000
NbClassMethods (Avg)	0.4382	0.4003	0.4886
NbClassMethods (Max)	4.7143	4.8750	4.5000
NbClassAttr (Avg)	0.3465	0.4195	0.2492
NbClassAttr (Max)	2.2143	2.5000	1.8333
NbInstMethods (Avg)	6.1109	6.2317	5.9499
NbInstMethods (Max)	25.5714	27.1250	23.5000
NbInstAttr (Avg)	1.6174	1.3972	1.9111
NbInstAttr (Max)	7.1429	7.3750	6.8333
NbMethods (Avg)	6.5654	6.6605	6.4385
NbMethods (Max)	27.0714	28.7500	24.8333
NbMethodsAll (Avg)	94.5030	112.6830	70.2629
NbMethodsAll (Max)	414.9286	513.2500	283.8333
NbMethodsProt (Avg)	0.1027	0.0936	0.1148
NbMethodsProt (Max)	1.7143	1.5000	2.0000
NbMethodsPub (Avg)	5.2550	5.1922	5.3388
NbMethodsPub (Max)	22.0714	24.0000	19.5000
SumEss (Avg)	6.4359	6.4865	6.3685
SumEss (Max)	31.2143	31.0000	31.5000
MaxCycl (Avg)	2.8764	2.8819	2.8689
MaxCycl (Max)	10.8571	11.1250	10.5000
MaxCycl* (Avg)	2.6594	2.6064	2.7301

measure	Avg	Avg (I)	Avg (M)
MaxCycl* (Max)	9.5714	9.6250	9.5000
MaxCyclStrict (Avg)	3.4061	3.5976	3.1826
MaxCyclStrict (Max)	14.9231	16.4286	13.1667
MaxEss (Avg)	1.3097	1.3222	1.2931
MaxEss (Max)	6.0714	5.7500	6.5000
DIT (Avg)	0.3526	0.2785	0.4513
DIT (Max)	1.3571	1.2500	1.5000
MaxNesting (Avg)	1.0591	0.9046	1.2652
MaxNesting (Max)	3.7857	3.6250	4.0000
LCOM% (Avg)	31.3982	31.4723	31.3118
LCOM% (Max)	84.0769	84.8571	83.1667
LCOM*% (Avg)	27.5726	26.4119	28.9269
LCOM*% (Max)	79.6923	78.1429	81.5000
SumCycl (Avg)	10.4729	10.2071	10.8273
SumCycl (Max)	54.4286	51.1250	58.8333
SumCycl* (Avg)	10.0892	9.7026	10.6047
SumCycl* (Max)	53.3571	50.6250	57.0000
SumCyclStrict (Avg)	11.4045	11.1294	11.7714
SumCyclStrict (Max)	59.4286	55.0000	65.3333
FnLines (Avg)	6.4768	6.0209	7.0846
FnLines (Max)	76.5714	79.5000	72.6667
FnLinesCode (Avg)	5.8576	5.6029	6.1973
FnLinesCode (Max)	62.2857	65.3750	58.1667
FnLinesComm (Avg)	0.2342	0.1670	0.3237
FnLinesComm (Max)	10.1429	11.3750	8.5000
LoC	2029.2858	1945.7500	2140.6667
LoCBlank	265.8571	226.0000	319.0000
LoCCode	1555.6428	1536.7500	1580.8334
LoCDecl	469.3077	448.1429	494.0000
LoCExec	583.6923	606.1429	557.5000
LoCComm	167.0000	118.7500	231.3333
NbSemicolons	672.0769	675.0000	668.6667
NbStmt	958.4286	943.7500	978.0000
NbDecl	483.3571	465.3750	507.3333
NbExe	521.5000	527.8750	513.0000
RatioComm	0.1044	0.1078	0.0998

Mutable Representation of a Command Invoker



Immutable Representation of a Command Invoker



Measures on Participants

Participants	67
Teams	14
Participants (mutable group)	29
Teams (mutable group)	6
Participants (immutable group)	37
Teams (immutable group)	8
Survey respondents	50
Survey respondents (mutable group)	20
Survey respondents (immutable group)	30

Mann Whitney U test results

Measure	P-Value	Cliff's Delta
AvgCycl (Avg)	0.8465	-0.0833
AvgCycl (Max)	1.0000	-0.0208
AvgCycl* (Avg)	0.5613	0.2083
AvgCycl* (Max)	0.7399	0.1250
AvgCyclStrict (Avg)	0.8972	-0.0625
AvgCyclStrict (Max)	0.8902	-0.0625
AvgEss (Avg)	0.8401	-0.0833
AvgEss (Max)	0.6637	0.1458
CBO (Avg)	0.5203	-0.2381
CBO (Max)	0.7172	0.1429
CBO* (Avg)	0.8303	0.0952
CBO* (Max)	0.3848	0.3095
DIT (Avg)	0.7466	0.1250
DIT (Max)	0.5813	0.1875
FnLines (Avg)	0.3329	0.3333
FnLines (Max)	0.9485	0.0417
FnLinesCode (Avg)	0.7469	0.1250
FnLinesCode (Max)	0.8463	0.0833
FnLinesComm (Avg)	0.0612	0.6250
FnLinesComm (Max)	0.4772	0.2500
LCOM*% (Avg)	0.9431	0.0476
LCOM*% (Max)	0.7730	0.1190
LCOM% (Avg)	0.9431	-0.0476
LCOM% (Max)	0.8296	-0.0952
LoC	0.5613	0.2083
LoCBlank	0.1962	0.4375
LoCCode	0.7469	0.1250
LoCComm	0.4777	0.2500
LoCDecl	0.7210	0.1429
LoCExec	1.0000	0.0000
MaxCycl (Avg)	0.7960	0.1042
MaxCycl (Max)	0.2369	0.3958
MaxCycl* (Avg)	0.4381	0.2708
MaxCycl* (Max)	0.3254	0.3333
MaxCyclStrict (Avg)	1.0000	-0.0000
MaxCyclStrict (Max)	0.7188	0.1429
MaxEss (Avg)	0.8465	-0.0833

Measure	P-Value	Cliff's Delta
MaxEss (Avg)	0.8465	-0.0833
MaxEss (Max)	0.3972	0.2917
MaxNesting (Avg)	0.1752	0.4583
MaxNesting (Max)	1.0000	0.0208
NOC (Avg)	0.9482	0.0417
NOC (Max)	0.8950	0.0625
NbClassAttr (Avg)	0.0926	-0.5625
NbClassAttr (Max)	0.0789	-0.5625
NbClassBase (Avg)	0.6510	-0.1667
NbClassBase (Max)	1.0000	0.0000
NbClassMethods (Avg)	1.0000	0.0000
NbClassMethods (Max)	1.0000	0.0208
NbDecl	0.4772	0.2500
NbExe	0.7469	0.1250
NbInstAttr (Avg)	0.8465	-0.0833
NbInstAttr (Max)	1.0000	-0.0208
NbInstMethods (Avg)	0.6514	-0.1667
NbInstMethods (Max)	0.5181	0.2292
NbMethods (Avg)	0.8465	-0.0833
NbMethods (Max)	0.8465	0.0833
NbMethodsAll (Avg)	0.8465	-0.0833
NbMethodsAll (Max)	0.4777	-0.2500
NbMethodsProtect (Avg)	0.9450	0.0417
NbMethodsProtect (Max)	0.9448	0.0417
NbMethodsPublic (Avg)	0.9485	0.0417
NbMethodsPublic (Max)	0.9483	0.0417
NbSemicolons	0.7210	0.1429
NbStmt	0.4777	0.2500
RatioComm	0.8465	0.0833
SumCycl (Avg)	1.0000	0.0000
SumCycl (Max)	0.3656	0.3125
SumCycl* (Avg)	0.9485	0.0417
SumCycl* (Max)	0.4772	0.2500
SumCyclStrict (Avg)	1.0000	0.0000
SumCyclStrict (Max)	0.2195	0.4167
SumEss (Avg)	0.6514	-0.1667
SumEss (Max)	0.6982	0.1458

Survey Answers Average

measure	Average (All)	Average (Mutable)	Average (Immutable)
Expertise (self-evaluation)	3.78	3.55	3.93
Contribution (self-evaluation)	3.92	3.8	4
Workload (Phase 1)	3.14	3.25	3.07
Workload (SOLID/GRASP)	3.2	3.3	3.13
Workload (Phase 2)	2.68	2.7	2.67
Workload (GoF)	2.82	2.95	2.73
Difficulty (Phase 1)	3.06	3.3	2.9
Difficulty (SOLID)	3.18	3.3	3.1
Difficulty (Phase 2)	2.64	2.85	2.5
Difficulty (GoF)	2.84	2.95	2.77
Complexity (Phase 1)	3.16	3.3	3.07
Complexity (SOLID/GRASP)	3.04	3.15	2.97
Complexity (Phase 2)	2.96	3.15	2.83
Complexity (GoF)	2.88	3.15	2.7
Would use immutability?	58%	65%	53.33%

Mann-Whitney U test results on survey

measure	P-Value	Cliff's Delta
Workload (Phase 1)	0.3065	0.16
Workload (SOLID/GRASP)	0.2708	0.17
Workload (Phase 2)	0.8242	0.04
Workload (GoF)	0.4589	0.12
Difficulty (Phase 1)	0.1386	0.24
Difficulty (SOLID)	0.4891	0.11
Difficulty (Phase 2)	0.3112	0.17
Difficulty (GoF)	0.5834	0.09
Complexity (Phase 1)	0.5137	0.11
Complexity (SOLID/GRASP)	0.5509	0.1
Complexity (Phase 2)	0.3145	0.16
Complexity (GoF)	0.1571	0.23

Survey

- The survey was divided into 5 mains sections:
  - General questions about the participant's industry experience and knowledge of C#
  - Appraisal of the base program (workload, difficulty, complexity)
  - Appraisal of the extended program (workload, difficulty, complexity)
  - NASA Task Load Index (mental demand, physical demand, temporal demand, performance, effort, frustration)
  - Questions about the new features (treatment group only)

Program Measures

ID	Group	MI	CC	CBO	NLoC	NLoCE
1	Control	92	74	25	375	106
2	Control	91	99	27	513	170
3	Control	92	81	26	381	103
4	Control	92	76	26	359	102
5	Control	92	74	25	375	104
6	Treatment	93	60	25	339	89
7	Treatment	93	59	25	346	90
8	Treatment	93	63	26	368	86
9	Treatment	93	66	26	355	98
10	Treatment	94	58	25	331	84

Group	MI	CC	CBO	NLoC	NLoCE
Control	93	61	25	318	78
Treatment	95	48	25	276	59

Mann Whitney U test results

Measure	P-Value	Effect Size
Maintainability Index	0.0074	-1.00
Cyclomatic Complexity	0.0119	1.00
Coupling Between Objects	0.4884	0.28
NLoC	0.0211	0.92
NLoCExec	0.0122	1.00

Appraisal of the Base Program

- Student Answers

Measure	Control Avg (SD)	Treatment Avg (SD)
Workload	3.8 (0.74)	3.6 (0.8)
Difficulty	4.0 (0.63)	3.4 (0.49)
Complexity	3.6 (0.8)	3.4 (1.02)

- Professional Developers Answers

Measure	Control	Treatment
Workload	1	2
Difficulty	1	2
Complexity	1	3

Appraisal of the Extended Program

- Student Answers

Measure	Control Avg (SD)	Treatment Avg (SD)
Workload	3.8 (0.75)	3.4 (0.49)
Difficulty	3.2 (0.98)	3.2 (0.75)
Complexity	3.2 (0.75)	2.2 (0.4)

- Professional Developers Answers

Measure	Control	Treatment
Workload	1	1
Difficulty	2	1
Complexity	2	2

NASA Task Load Index Answers

- Student Answers

Measure	Control	Treatment
Mental Demand	5.4 (1.2)	6.0 (1.10)
Physical Demand	3.6 (2.33)	4.4 (1.85)
Temporal Demand	5.4 (1.36)	4.6 (2.33)
Performance	5.2 (2.4)	5.6 (1.36)
Effort	4.8 (1.72)	5.8 (1.17)
Frustration	5.0 (1.79)	5.6 (2.33)

- Professional Developers

Measure	Control	Treatment
Mental Demand	3	2
Physical Demand	1	1
Temporal Demand	2	2
Performance	7	7
Effort	4	3
Frustration	1	3

Participants

- 12 participants
  - 10 graduate students
  - Two professional developers
- Participants are split into two groups
  - The treatment group is given a short training on the four C# features
  - The control group is not given any information concerning the new features
  - The professional developer in the control group is given specific instructions not to use the new features

Record Types

- Data structure similar to class or struct
- Structural equality
- In C#, not required to be immutable
- Allows the use of Record Updating

Record Type Example

```
record MyRecord(string Field1, int Field2)
{
    void SomeMethod(string arg)
    {
        Field1 = arg;
        return Field2;
    }
}
```

Record Updating

- Also called non-destructive mutation or functional updating
- Allows duplicating a record while changing some values of the duplicate
- Does not affect the original record
- Allows creating “setters” in an immutable context

Record Updating Example

```
var entry1 = new MyRecord("Content", 0);
var entry2 = entry with { Field2 = 42 };
```

Pattern Matching

- Allows code to branch based on the type of an object
- Similar syntax to switch statement
- Feature present in most FP languages
- FP equivalent to object polymorphism
- Usually combined with Records

Pattern Matching Example

```
// If-expression example
if (someVariable is int i)
{
    return i + 2;
}
else
{
    return 0;
}

// Switch-expression example
return (someVariable switch
{
    int i => i + 2;
    string s => 1;
    _ => throw new
        InvalidOperationException();
});
```

Multiple Values Return

- In FP, it is frequent to return multiple values from a single function call
- Pure functions cannot have side effects
  - They must return every updated object
- If a pure function updates more than one object at a type, it must return multiple values

MVR Example

```
(int, string) MakeTuple()
{
    return (42, "Content");
}

// ...

(int, string) tuple = MakeTuple();
// Use tuple.Item1 and tuple.Item2
// to access the values.

// or alternatively

(int i, string s) = MakeTuple();
// Use i and s as normal variables.
```

Kotlin Variant

- Statically-typed language
- Similar to Java
- Functional updating support

Kotlin Movable Trait

```
interface Movable<T> {
    val x: Int
    val y: Int

    fun updateMovable(newX: Int, newY: Int) : T

    fun move(moveX: Int, moveY: Int) : T {
        return updateMovable(moveX + x, moveY + y)
    }
}
```

Kotlin Movable Implementations

```
data class Point(override val x: Int, override val y: Int)
        : Movable<Point> {

    override fun updateMovable(newX: Int, newY: Int): Point {
        return Point(x, y)
    }
}

data class Rectangle(override val x: Int, override val y: Int,
                     override val w: Int, override val h: Int)
                : Movable<Rectangle> {
    override fun updateMovable(newX: Int, newY: Int): Rectangle {
        return copy(x = newX, y = newY)
    }
}
```

With-New Macro

Clone Method

```
(defun clone-object (instance)
  (let* ((class (class-of instance))
         (clone (allocate-instance class)))
    (dolist (slot-name
             (mapcar #'closer-mop:slot-definition-name
                     (closer-mop:class-slots class)))
      (when (slot-boundp instance slot-name)
        (setf (slot-value clone slot-name)
              (slot-value instance slot-name))))
    clone))
```

With-New Macro

```
(defmacro with-new (slots instance &body body)
  (let ((instance-sym (gensym)))
    `(let ((,instance-sym (clone-object ,instance)))
       (with-slots ,slots ,instance-sym
         ,@body
         ,instance-sym))))
```