Refactoring

How to improve your code

ARTEM TABALIN

What is refactoring?

Definition. Goals. When to Refactor. Pitfalls.

What is Refactoring?

*“**Refactoring** is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.”*

- Martin Fowler

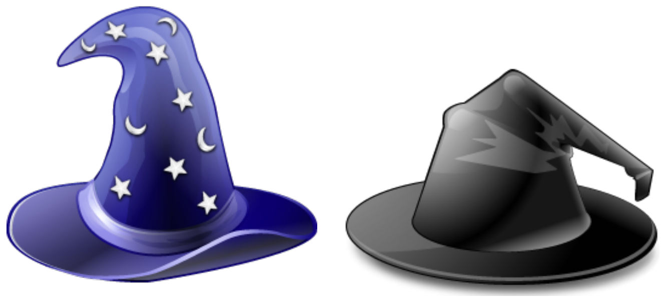
Is it just cleaning up code?

The technique of cleaning up code in more organized and controlled way

The Two Hats

Two distinct activities

1. adding functions
2. refactoring



These two activities should NOT take place at the same time

Goals

- Prevent code quality reduction
- Make code clear
- Help to find bugs
- **Help to program faster**

When to Refactor

The Rule of Three

Three strikes and you refactor

1. You do something the first time
2. You do similar thing again with regret
3. On the third time – start refactoring

When to Refactor

- When adding new functionality
    - understanding the code
    - prepare code for adding new functionality
- When fixing a bug
    - error is a sign for refactoring
- Code reviews

Problems with Refactoring

Database

- difficult to change since tightly coupled to scheme
- existing data migration

Changing Interface

- deprecate old interface
- just call new method in old method
- don’t publish premature interfaces

When You Shouldn’t Refactor

Low code quality

- easier rewrite than refactor
- code should work correctly before refactoring

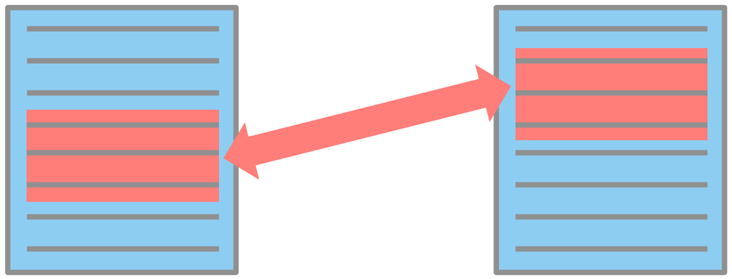
Deadline is too close

- postpone refactoring = technical debt

Smelling Code

Signs of low quality code

Code Duplication



- The same code in two methods

  Extract Method

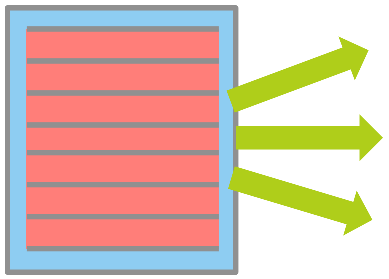
- The same code in two subclasses

  Extract Method + Pull Up Field / Method

- The same code in two different classes

  Extract Class

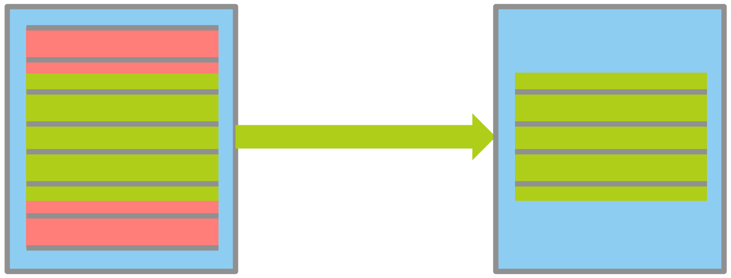
Long Method



- Sections in method
- Wish to comment part of code
- Long loops and conditionals

Extract Method

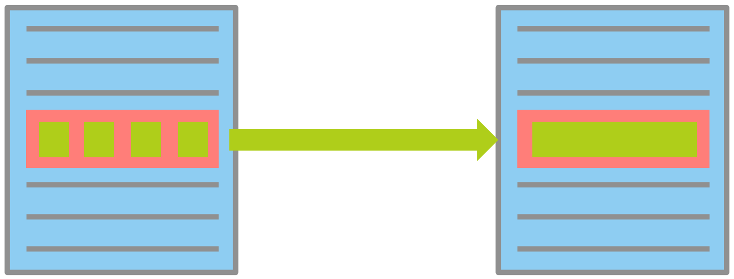
Large Class



- To many methods
- To many fields

Extract Class / Extract Subclass

Long Parameter List



- Method can calculate parameter itself

  Replace Parameter with Method

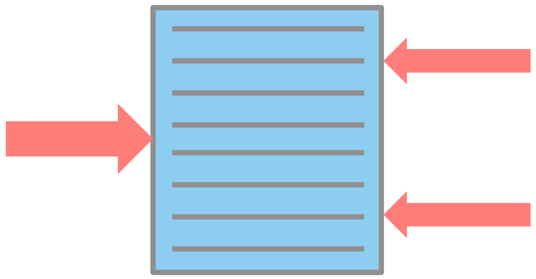
- Parameters are fields of single object

  Preserve Whole Object

- Several related parameters

  Introduce Parameter Object

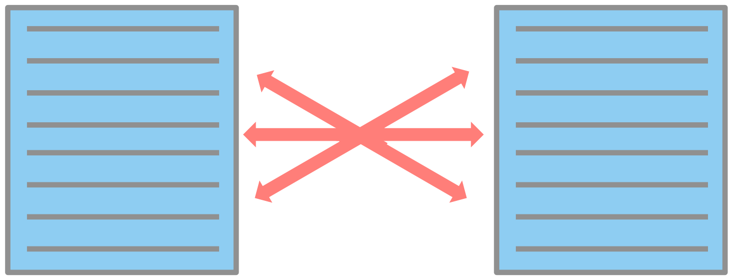
Divergent Change



- Class is changed for different reasons

  Extract Class

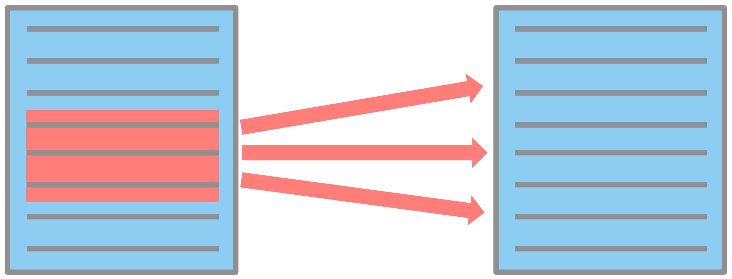
Shotgun Surgery



- Lots of little changes to lot of different classes

  Move Method / Move Field

Feature Envy



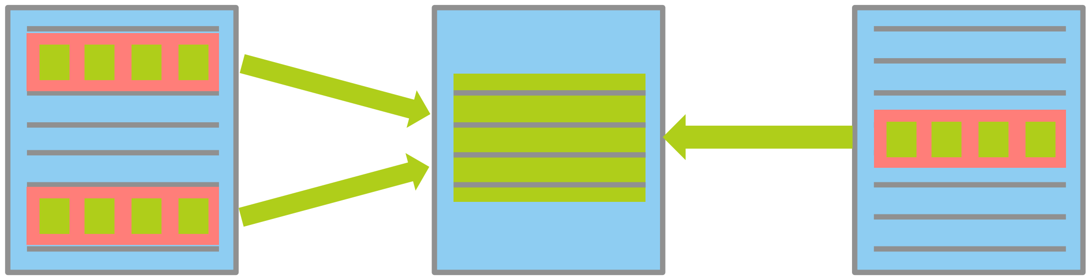
- Method is interested in another class too much

  Move Method [+ Extract Method]

Exceptions

- Strategy
- Visitor
- Data Access Layer

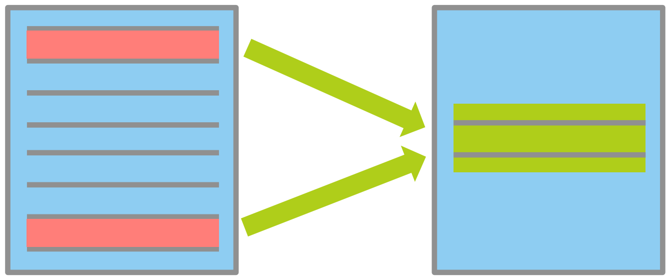
Data Clumps



- Few data items together in lots of places

  Extract Class / Introduce Parameter Object

Primitive Obsession



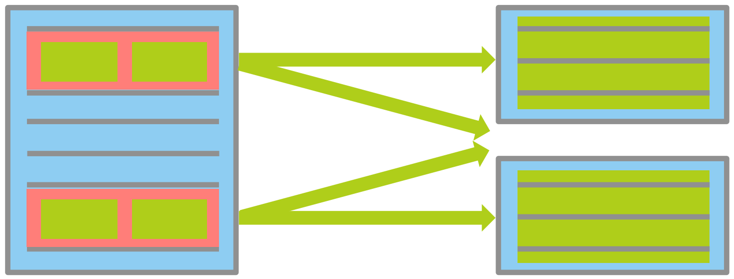
- Avoiding small classes and using primitives instead

  range, money, phone, array instead of object

  Replace Data Value with Object

  Extract Class / Introduce Parameter Object

Switch Statements



- Switch statements all over the class

  Replace Type Code with Subclasses
  Replace Conditional with Polymorphism

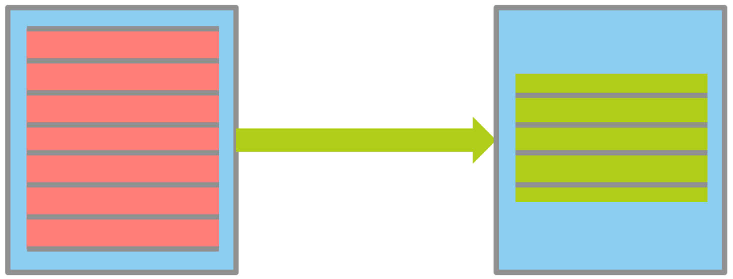
- Few switch statements

  Replace Type Code with Strategy / State

- Null special cases

  Introduce Null Object

Speculative Generality



We need this functionality someday

- Unnecessary base class

  Collapse Hierarchy

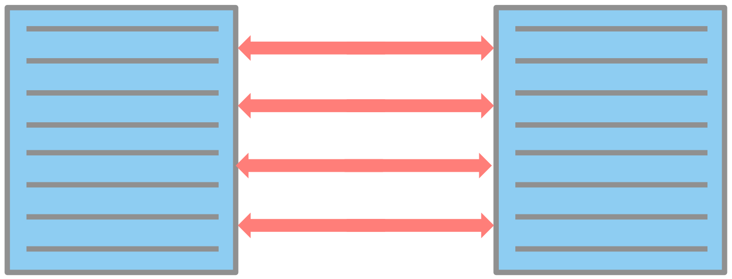
- Unnecessary delegation

  Inline Class

- Strange abstract method name

  Rename Method

Inappropriate Intimacy



Classes too coupled

- Use fields and methods of each other
  Change Bidirectional Association with Unidirectional
- Have common interests
  Extract Class
- High coupling due to inheritance
  Replace Inheritance with Delegation

Other Smells

- Lazy Class

  Inline Class

- Temporary Field

  Extract Class

- Message Chains `getThis().getThat().getObject().method()`

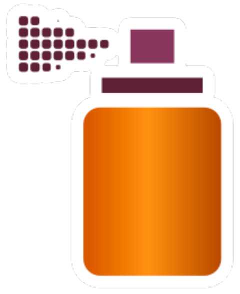
  Hide Delegate

- Refused Bequest

  Replace Inheritance with Delegation

Comments

Used like a **deodorant**



Indicators of bad code

If you need a comment try to refactor

- Extract Method
- Rename Method
- Introduce Assertion

Catalog of Refactorings

Composing Methods

Extract Method. Inline Method. Temporary Variables. Replace Method with Method Object.

Extract Method

Turn the fragment of code into a method

When

- Method is too long
- Code needs an explanatory comment

Why

- Increase reusability
- Higher level methods become more readable
- Easier to override

Extract Method

```
public void PrintClient(Client client) {
    PrintBanner();
    // print client details
    Console.Write("Name: " + client.Name);
    Console.Write("Address: " + client.Address);
}
```

```
public void PrintClient(Client client) {
    PrintBanner();
    PrintClientDetails(client);
}

private void PrintClientDetails(Client client) {
    Console.Write("Name: " + client.Name);
    Console.Write("Address: " + client.Address);
}
```

Inline Method

Put the method body to callers and remove it

When

- Method body is as clear as its name
- Calling method is badly factored

Why

- Reduce redundant delegation
- Restructure calling method

Inline Method

```
public int GetRating() {
    return MoreThanFiveLateDeliveries() ? 2 : 1;
}

private bool MoreThanFiveLateDeliveries() {
    return _numberOfLateDeliveries > 5;
}
```

↓

```
public int GetRating() {
    return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Replace Temp with Query

Extract expression to a method and replace temp

When

- Temp holds an expression result for other methods
- Temp prevents from another refactoring

Why

- Cleaner code
- Shorter method
- The result is available to all methods

Replace Temp with Query

```
public double GetPrice() {
    double basePrice = _quantity * _itemPrice;

    return (basePrice > 1000)
        ? basePrice * 0.95
        : basePrice * 0.98;
}
```

```
public double GetPrice() {
    return (BasePrice > 1000)
        ? BasePrice * 0.95
        : BasePrice * 0.98;
}

private double BasePrice {
    get { return _quantity * _itemPrice; }
}
```

Introduce Explaining Variable

Assign a part of expression to explaining variable

When

- Complex condition
- Complex algorithm

Why

- Improve code readability

Introduce Explaining Variable

```
if ((platform.toUpperCase().indexOf("MAC") > -1) &&
    (browser.toUpperCase().indexOf("IE") > -1) &&
    isInitialized() && resize > 0) {
    // do something
}
```

```
var isMac = platform.toUpperCase().indexOf("MAC") > -1;
var isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;
var isResized = resize > 0;
if (isMac && isIEBrowser && isInitialized() && isResized){
    // do something
}
```

Replace Method with Method Object

Transform long method into object

When

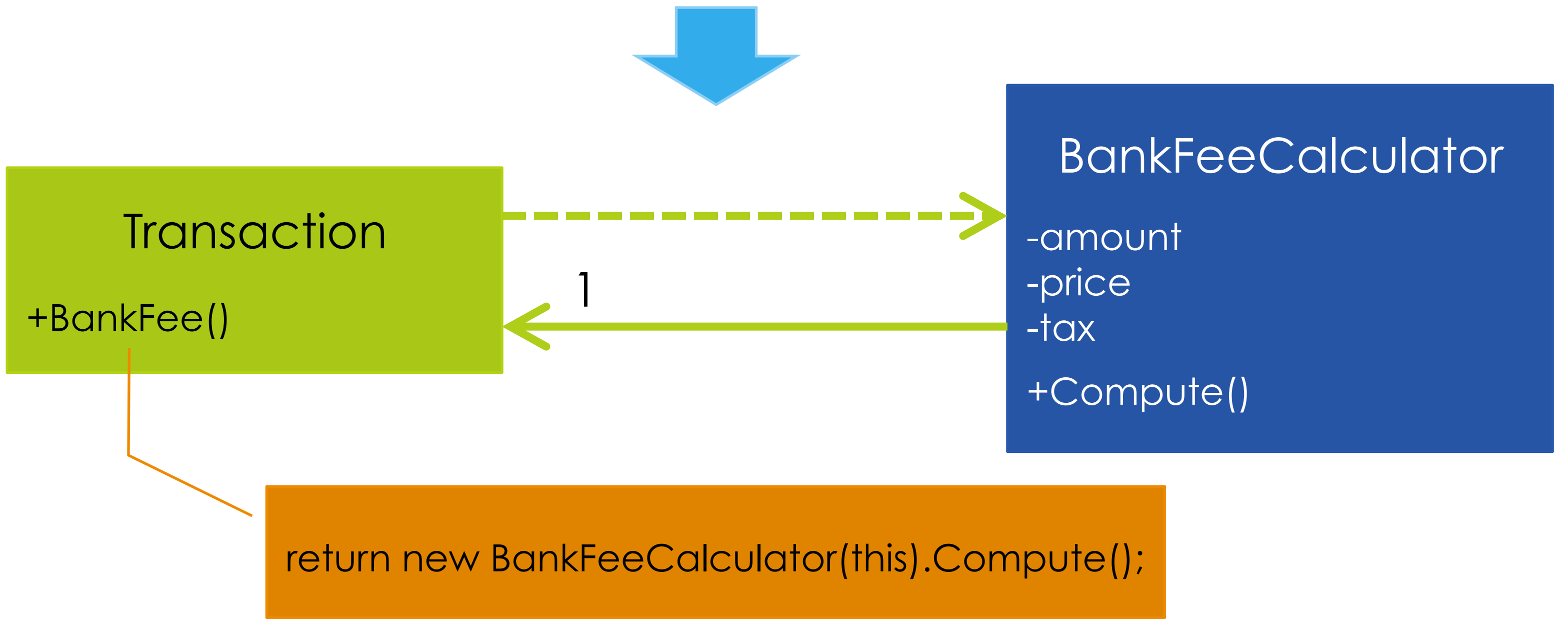
- Long method with lots of temp variables

Why

- Improve readability
- Increase reusability

Replace Method with Method Object

```
class Transaction {
    public double BankFee() {
        double amount;
        double price;
        double tax;
        // long computations
    }
```



```
return new BankFeeCalculator(this).Compute();
```

Moving Features Between Objects

Move Method. Move Field. Extract Class. Inline Class. Hide Delegation.

Move Method

Move the method to the class that uses it most

When

- Method references another object too much
- Two classes are too coupled
- Class is overcomplicated

Why

- Lower coupling
- Simplify design

Move Method

```
class Account {
  private AccountType _type;
  public double BankCharge() {
    double result = FIXED_FEE;
    if(HasOverdraft())
      result += OverdraftFee();
    return result;
  }
  private double OverdraftFee() {
    if(_type.IsPremium()) {
      // premium account
    } else {
      // standard account
    }
  }
```

```
class Account {
  public double BankCharge() {
    double result = FIXED_FEE;
    if(HasOverdraft())
      result += _type.OverdraftFee();
    return result;
  }

class AccountType {
  public double OverdraftFee() {
    if(IsPremium()) {
      // premium account
    } else {
      // standard account
    }
  }
```

Move Field

Move the field to class that uses it most

When

- Another class uses a field more than its owner
- Performing Extract Class refactoring

Why

- Lower coupling
- Simplify design

Move Field

```
class Account {

  private AccountType _type;

  private double _interestRate;

  public double InterestRate {
    get { return _interestRate; }
  }

  double Interest(int days) {
    return InterestRate * days/365;
  }
```

```
class Account {
  double Interest(int days) {
    return _type.InterestRate * days/365;
  }


class AccountType {

  private double _interestRate;

  public double InterestRate {
    get { return _interestRate; }
  }
```

Extract Class

Create new class and move fields and methods

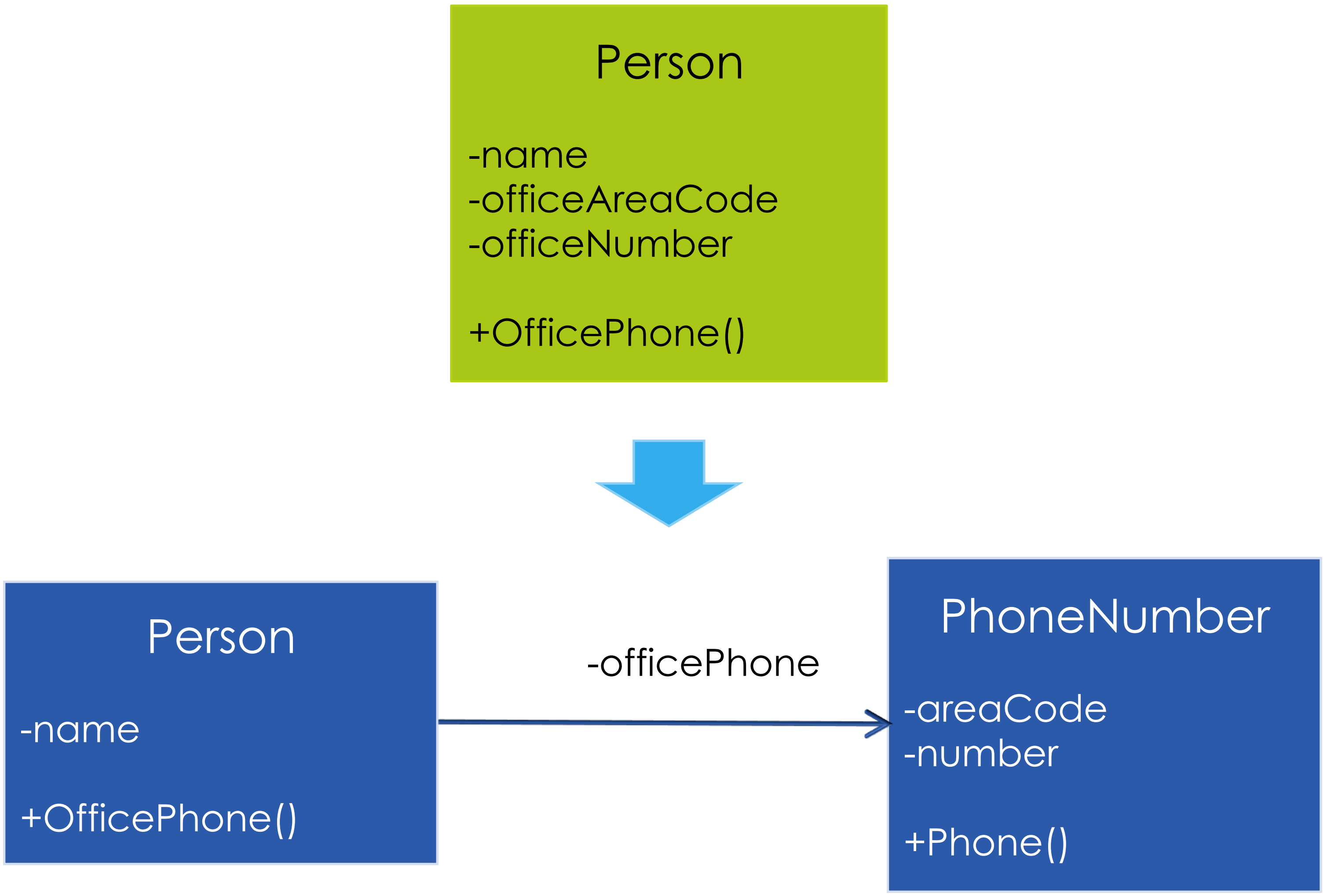
When

- Class is too big
- Single Responsibility Principle violation
- Data or methods dependent on each other

Why

- Simplify design

Extract Class



Inline Class

Move fields & methods from class and remove it

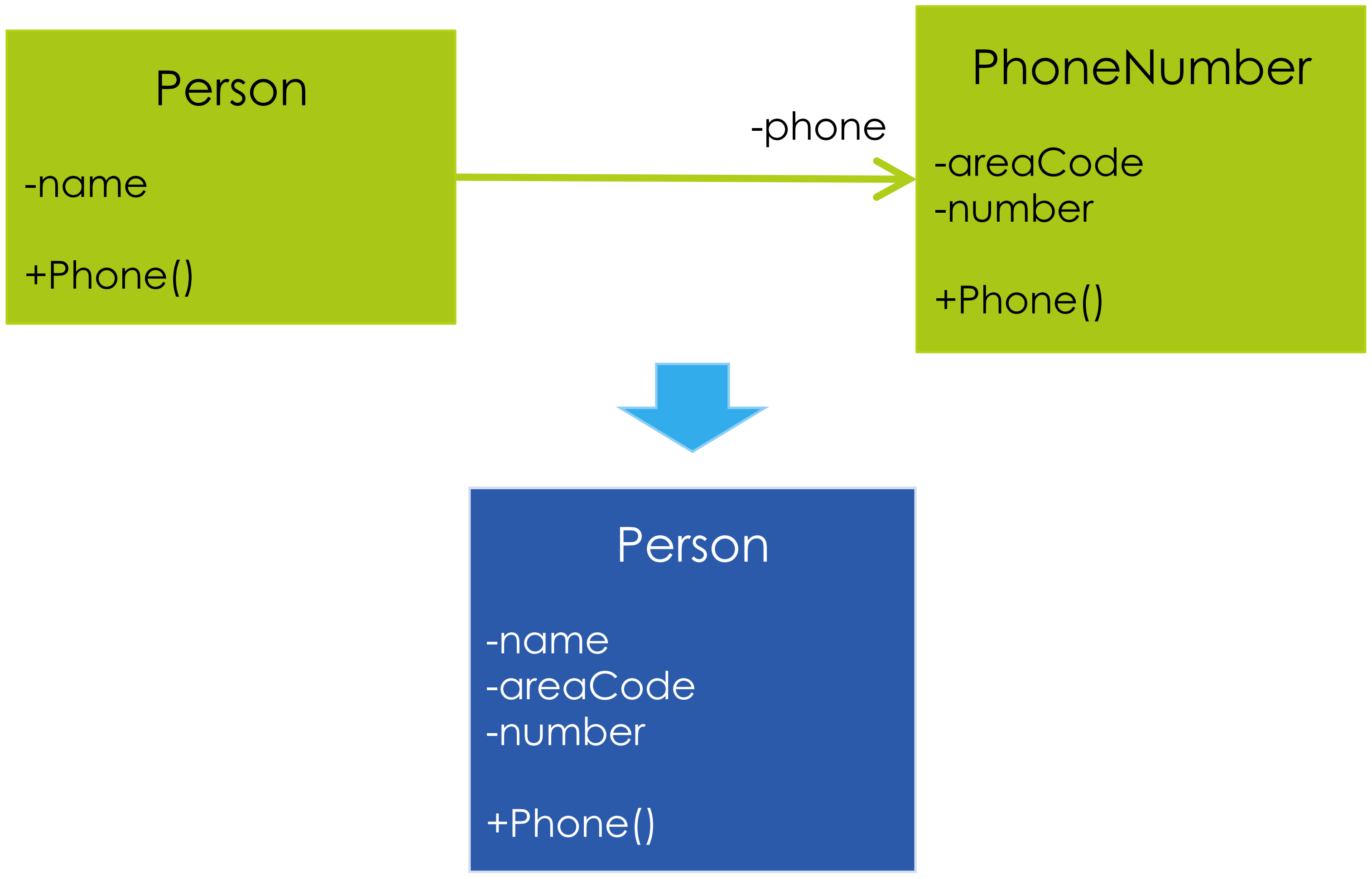
When

- Class is useless

Why

- Simplify design

Inline Class



Hide Delegate

Create method on the server to hide the delegate

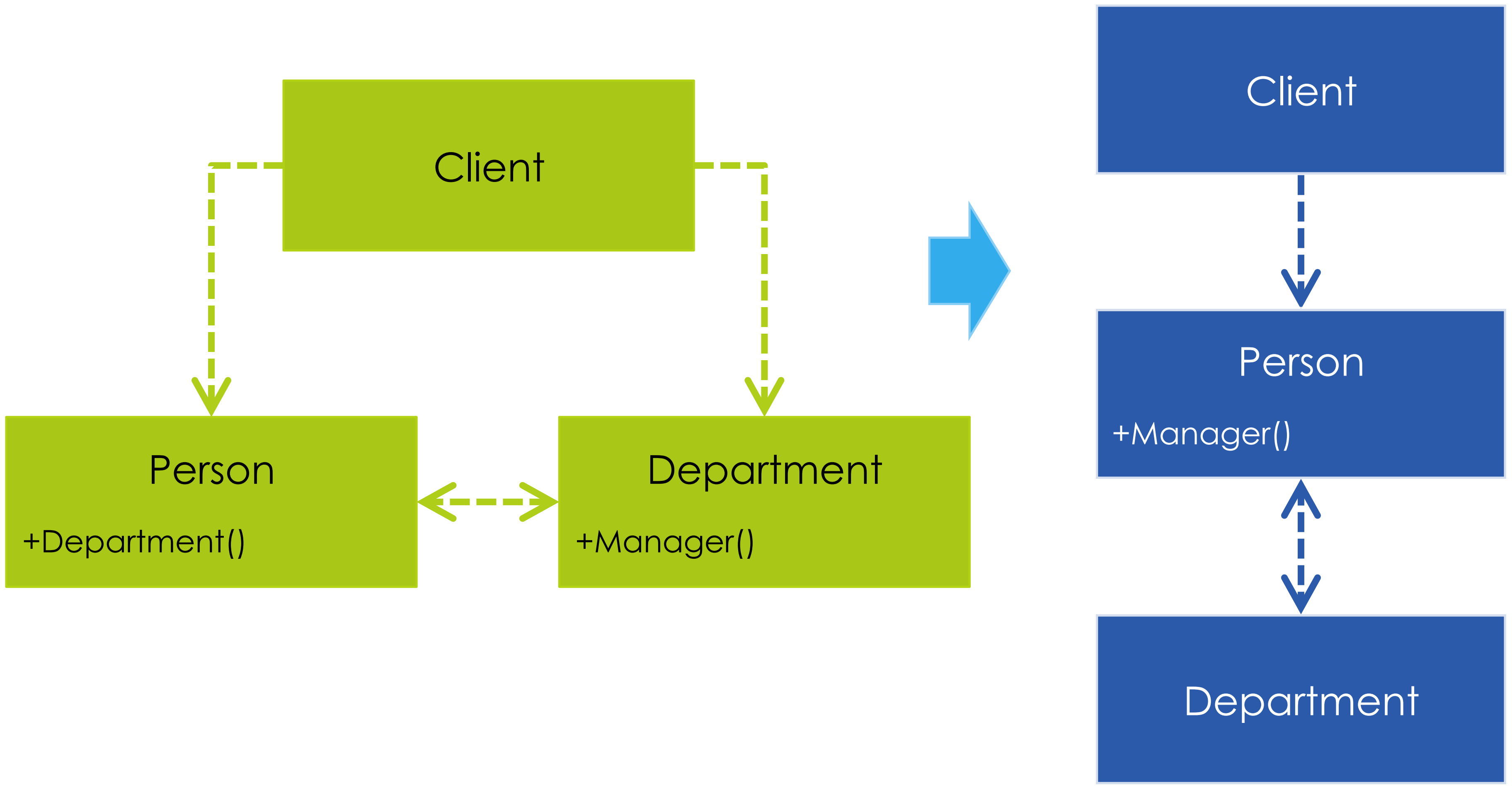
When

- object.getAnotherObject().method()

Why

- Lower coupling
- Strengthen encapsulation

Hide Delegate



Organizing Data

Self Encapsulate Field. Replacing Value with Object.
Replace Magic Number with Constant.
Replace Type Code with Subclasses/State/Strategy.

Self Encapsulate Field

Create and use getter and setter to access field

When

- Provide access to the field from outside
- Override property in a child class

Why

- Strengthen encapsulation
- Higher flexibility

Self Encapsulate Field

```
private double _interestRate;

public double Interest(int days) {
  return _interestRate * days/365;
}
```

```
private double _interestRate;

public double InterestRate {
  get { return _interestRate; }
  set { _interestRate = value; }
}

public double Interest(int days) {
  return InterestRate * days/365;
}
```

Replace Data Value with Object

Turn the data item to an object

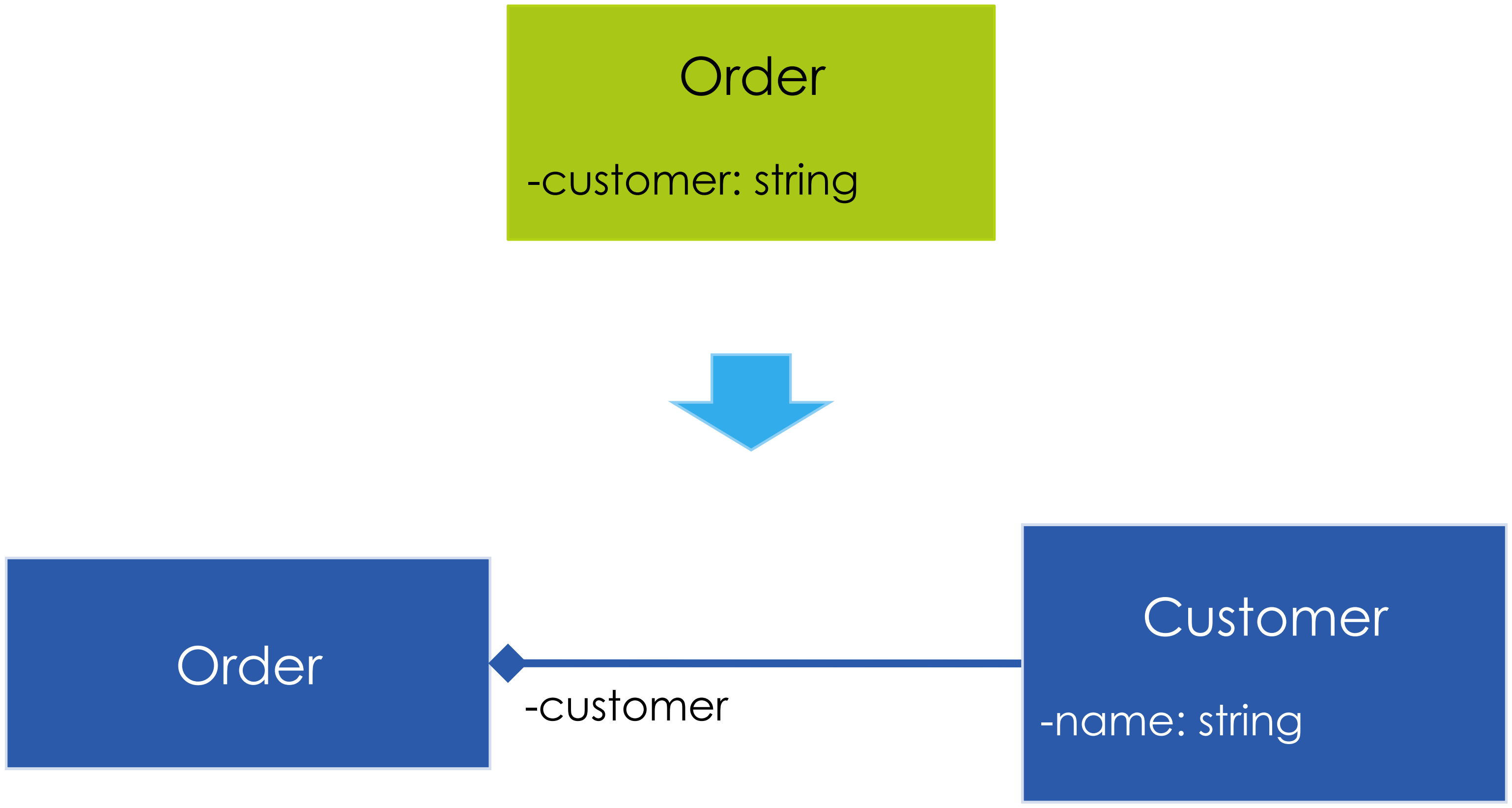
When

- Data item has dedicated methods
- Data items are used together in lots of places

Why

- Higher flexibility

Replace Data Value with Object



Replace Magic Number with Constant

Replace magic value with named constant

When

- There is a magic value in the code

Why

- Improve code readability

Replace Magic Number with Constant

```
double PotentialEnergy(double mass, double height) {
  return mass * 9.81 * height;
}
```

```
static const double GRAVITATIONAL_CONSTANT = 9.81;
double PotentialEnergy(double mass, double height) {
  return mass * GRAVITATIONAL_CONSTANT * height;
}
```

Replace Type Code with Subclasses

Replace the type code with subclasses

When

- There is a type code influencing on class behavior

Why

- Increase extensibility

Replace Type Code with Subclasses

```
class Employee {
  private EmployeeType _type;
  Employee(EmployeeType type) {
    _type = type;
  }
  public double Bonus {
    get {
      switch(_type) {
        case EmployeeType.Engineer:
          return 0.0;
        case EmployeeType.Salesman:
          return 0.25;
      }
    }
  }
}
```

```
abstract class Employee {
  abstract EmployeeType Type { get; }
  abstract double Bonus { get; }
}

class Engineer: Employee {

  public override EmployeeType Type {
    get {
      return EmployeeType.Engineer;
    }
  }

  public override double Bonus {
    get {
      return 0.0;
    }
  }
}
```

Replace Type Code with State/Strategy

Replace the type code with state/strategy classes

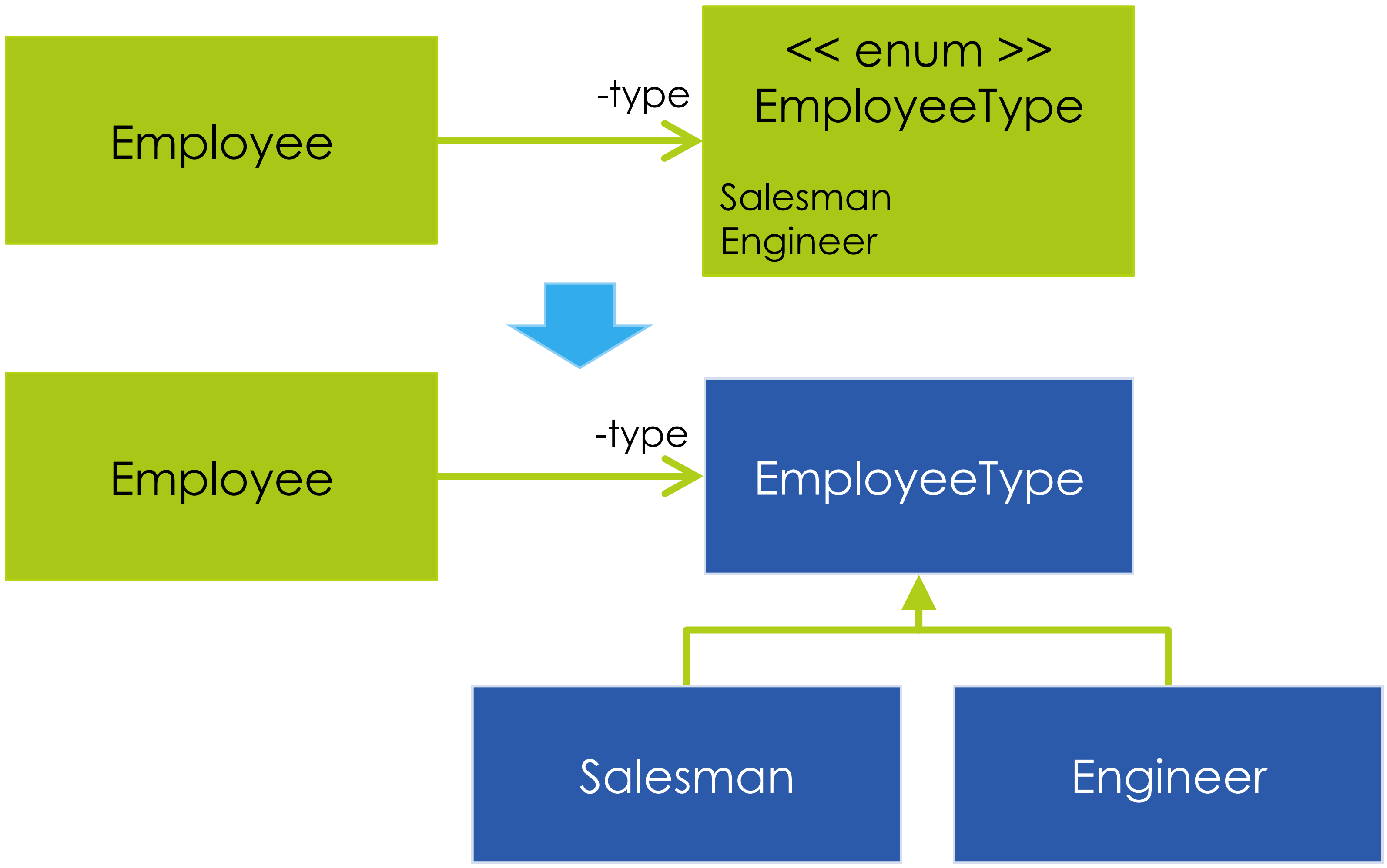
When

- There is a type code influencing on class behavior
- Type code may change during object life

Why

- Increase extensibility

Replace Type Code with State/Strategy



Simplifying Conditional Expressions

Decompose Conditional Expression.
Replace Conditional with Polymorphism.
Guard Clauses. Introduce Null Object.

Decompose Conditional

Extract methods from condition and branches

When

- Condition with complex expression and branches

Why

- Improve code readability

Decompose Conditional

```
if (date.Before(SUMMER_START) || date.After(SUMMER_END))
  charge = quantity * _winterRate + _winterServiceCharge;
else
  charge = quantity * _summerRate;
```

```
if (NotSummer(date))
  charge = WinterCharge(quantity);
else
  charge = SummerCharge(quantity);
```

Replace Nested Conditional with Guard Clauses

Use guard clauses for all special cases

When

- Conditional expression with special case branch

Why

- Improve code readability

Replace Nested Conditional with Guard Clauses

```
double PayAmount() {
  double result;
  if(_isDead)
    result = DeadAmount();
  else {
    if(_isSeparated)
      result = SeparatedAmount();
    else {
      if(_isRetired)
        result = RetiredAmount();
      else
        result = NormalAmount();
    }
  }
  return result;
}
```

```
double PayAmount() {
  if(_isDead)
    return DeadAmount();
  if(_isSeparated)
    return SeparatedAmount();
  if(_isRetired)
    return RetiredAmount();

  return NormalAmount();
}
```

Replace Conditional with Polymorphism

Move every condition branch to subclass method

When

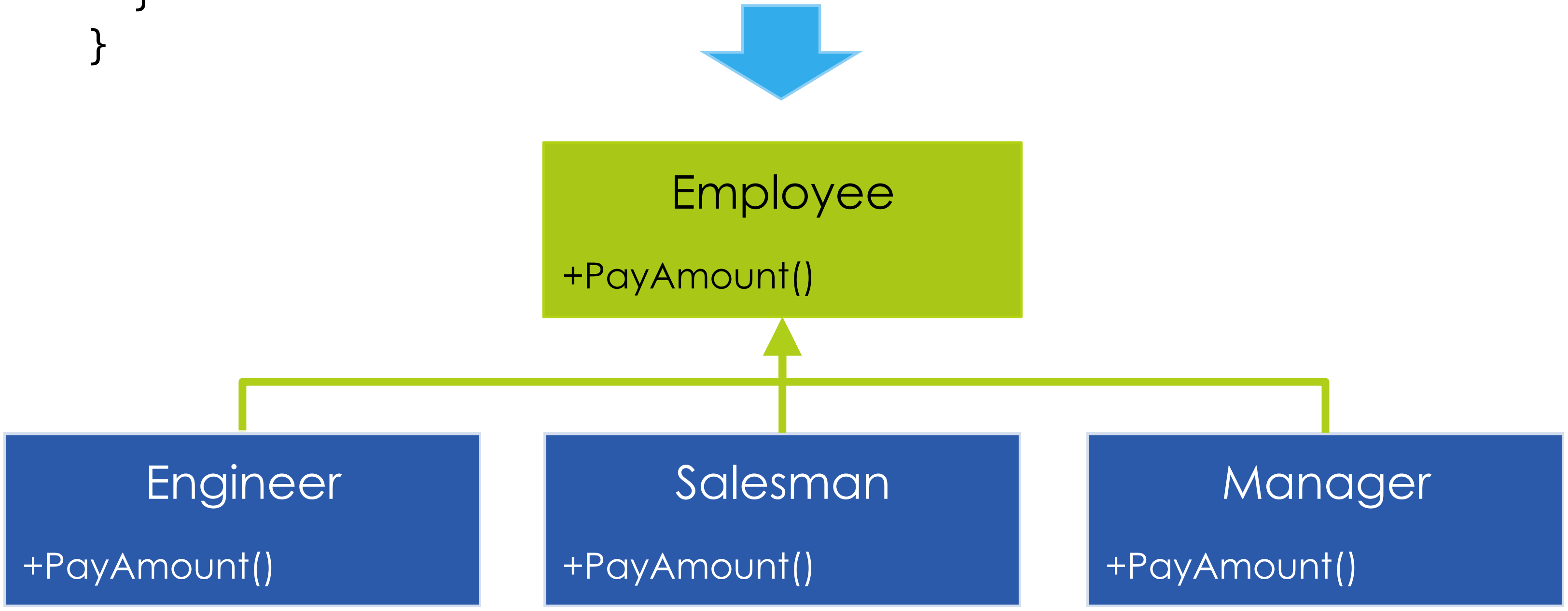
- There are conditions depending on object type

Why

- Increase extensibility
- Improve code readability

Replace Conditional with Polymorphism

```
class Employee {
  public double PayAmount() {
    switch (_type) {
      case EmployeeType.Engineer: return _salary;
      case EmployeeType.Salesman: return _salary + _commission;
      case EmployeeType.Manager: return _salary + _bonus;
      default: throw new WrongEmployeeTypeException();
    }
  }
```



Introduce Null Object

Replace null value with special class

When

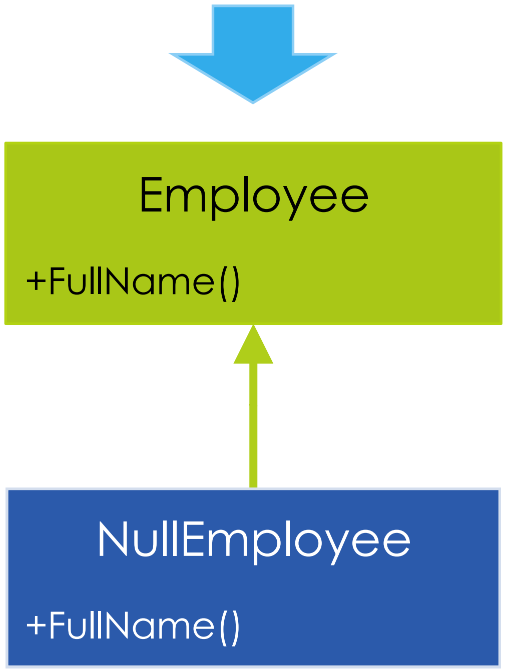
- There are lots of null checks

Why

- Reduce conditionals

Introduce Null Object

```
if(employee == null)
  name = NAME_PLACEHOLDER;
else
  name = employee.FullName;
```



Making Method Calls Simpler

Rename Method. Separate Query from Modifier. Preserve Whole Object. Introduce Parameter Object. Error Handling Refactorings.

Rename Method

Change method name

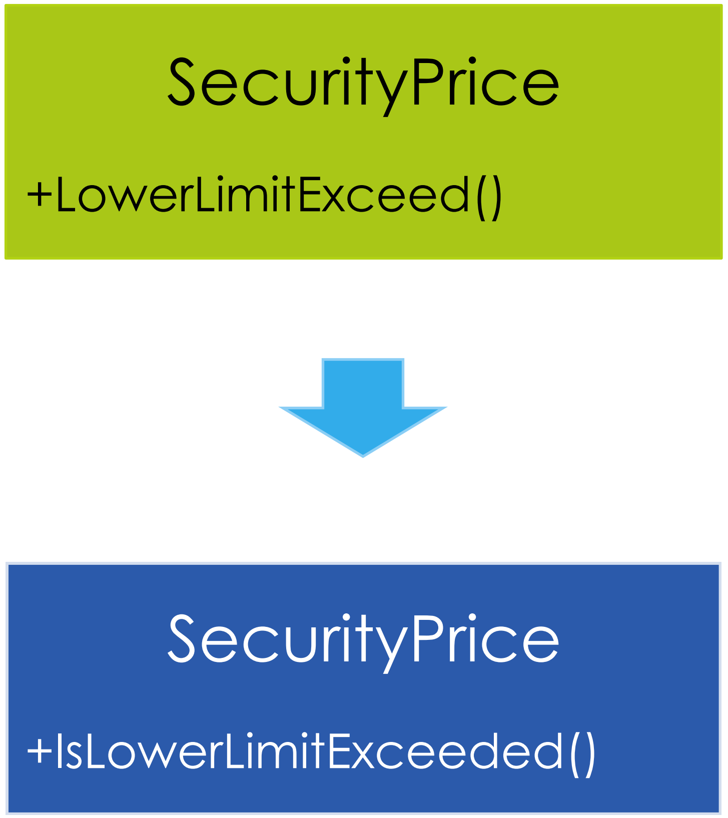
When

- Method name doesn't show its intention

Why

- Improve code readability

Rename Method



Separate Query from Modifier

Create methods for query and for modification

When

- Method returning value modifies object state

Why

- Simplify interface

Separate Query from Modifier

```
public List<Employee> FindRetired(List<Employee> employees) {
  var result = new List<Employee>();
  foreach(var emp in employees) {
    if(emp.IsRetired) {
      AddBonus(emp);
      result.Add(emp);
    }
  }
  return result;
}
```

```
public List<Employee> FindRetired(List<Employee> employees) {
  return employees.Where(emp => emp.IsRetired).ToList();
}

public List<Employee> AddBonusToRetired(List<Employee> employees) {
  foreach(var emp in employees) {
    if(emp.IsRetired)
      AddBonus(emp);
  }
}
```

Preserve Whole Object

Send the whole object to the method

When

- Method has several object field values as params

Why

- Simplify interface

Preserve Whole Object

```
DateTime start = Period.Start;
DateTime end = Period.End;
List<Event> events = schedule.FindEvents(start, end);
```

```
List<Event> events = schedule.FindEvents(Period);
```

Introduce Parameter Object

Replace method parameters with an object

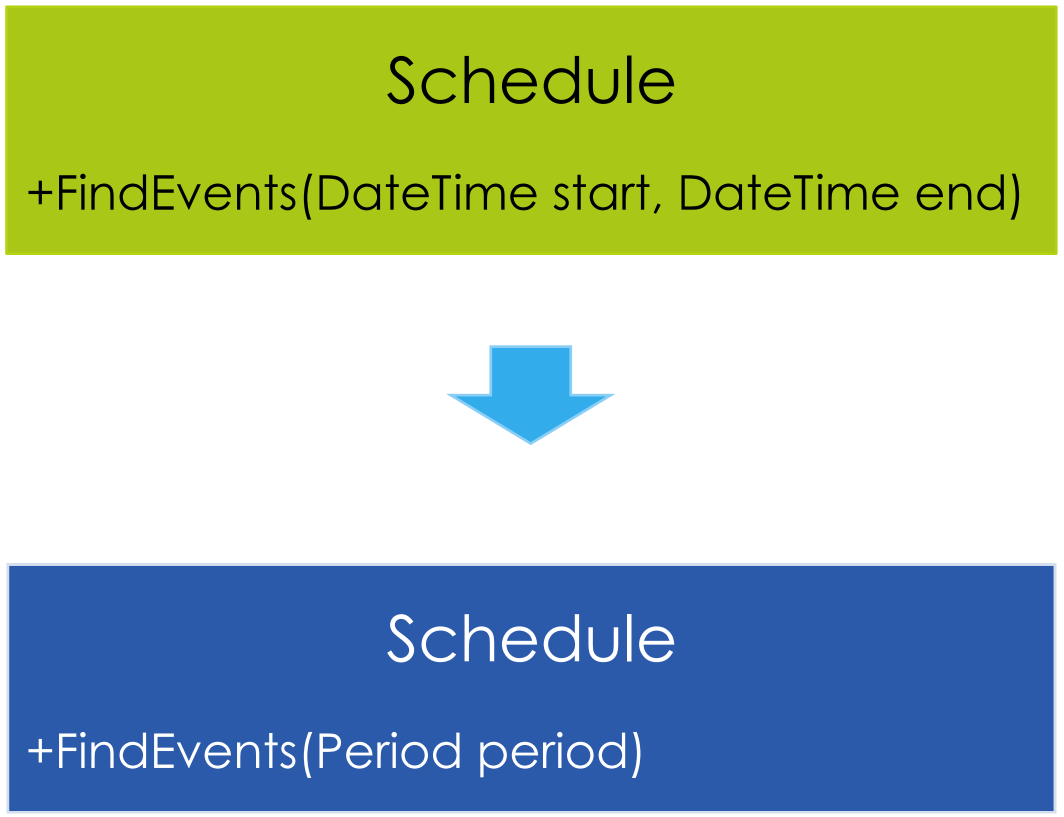
When

- Method accepts several related parameters

Why

- Simplify interface

Introduce Parameter Object


Schedule
+FindEvents(DateTime start, DateTime end)
Schedule
+FindEvents(Period period)

Replace Error Code with Exception

Throw exception instead of returning error code

When

- Method returns error code

Why

- Simplify interface

Replace Error Code with Exception

```
int Withdraw(int amount) {
  if (amount > _balance) {
    return -1;
  }
  else {
    _balance -= amount;
    return 0;
  }
}
```

→

```
void Withdraw(int amount) {
  if (amount > _balance)
    throw new BalanceException();

  _balance -= amount;
}
```

Replace Exception with Test

Put conditional instead of throwing exception

When

- Exception is used instead of conditional test

Why

- Improve code readability

Replace Exception with Test

```
public double ValueForPeriod(int periodIndex) {
  try {
    return _values[periodIndex];
  }
  catch (IndexOutOfRangeException e) {
    return 0;
  }
}
```

```
public double ValueForPeriod(int periodIndex) {
  if (periodIndex >= _values.length)
        return 0;
  return _values[periodIndex];
}
```

Generalization Refactorings

Pull Up and Push Down Method/Field.
Replace Inheritance with Delegation.
Extract Subclass/Superclass.

Pull Up Field/Method

Move method/field into the superclass

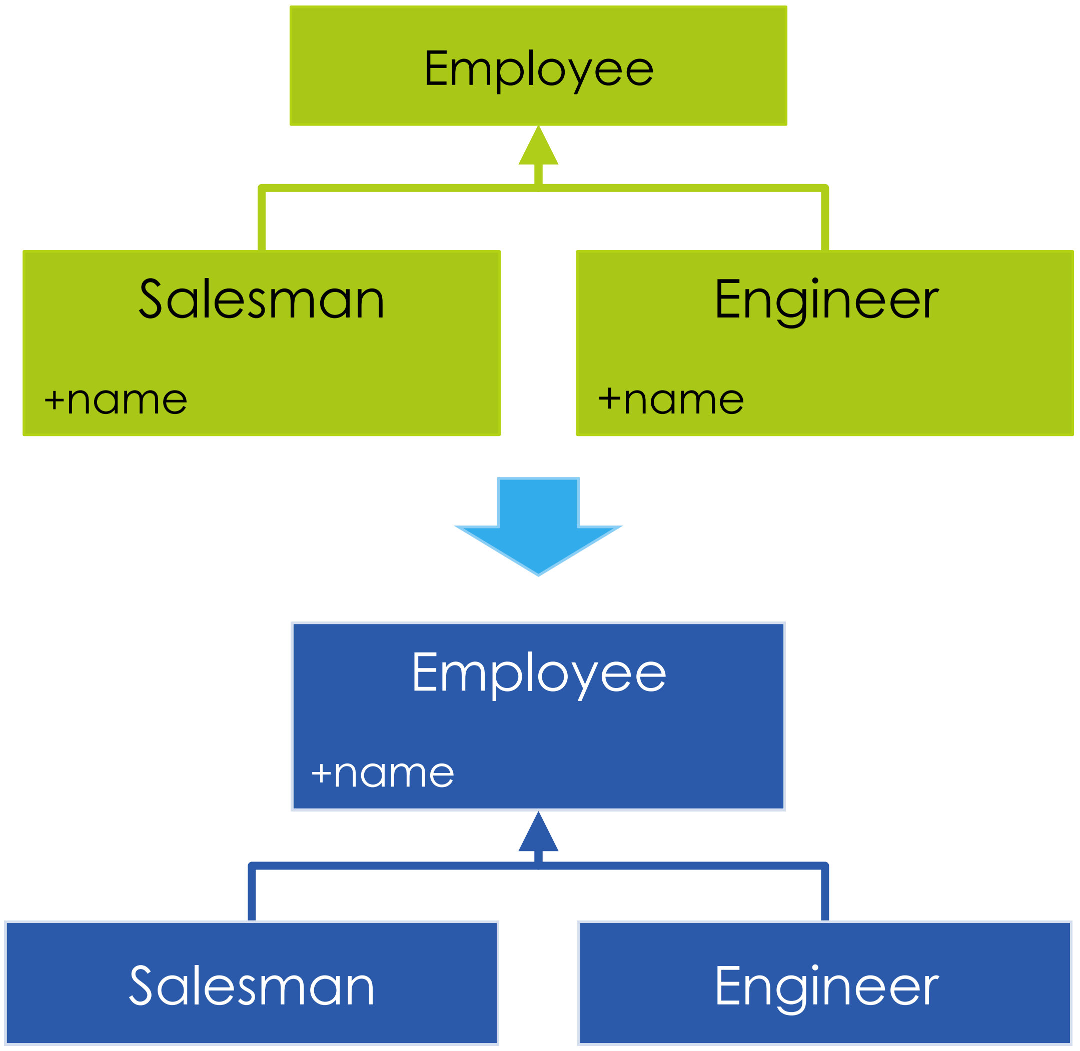
When

- The same method/field is in several child classes

Why

- Remove duplication

Pull Up Field/Method



Push Down Field/Method

Move method/field into the subclass

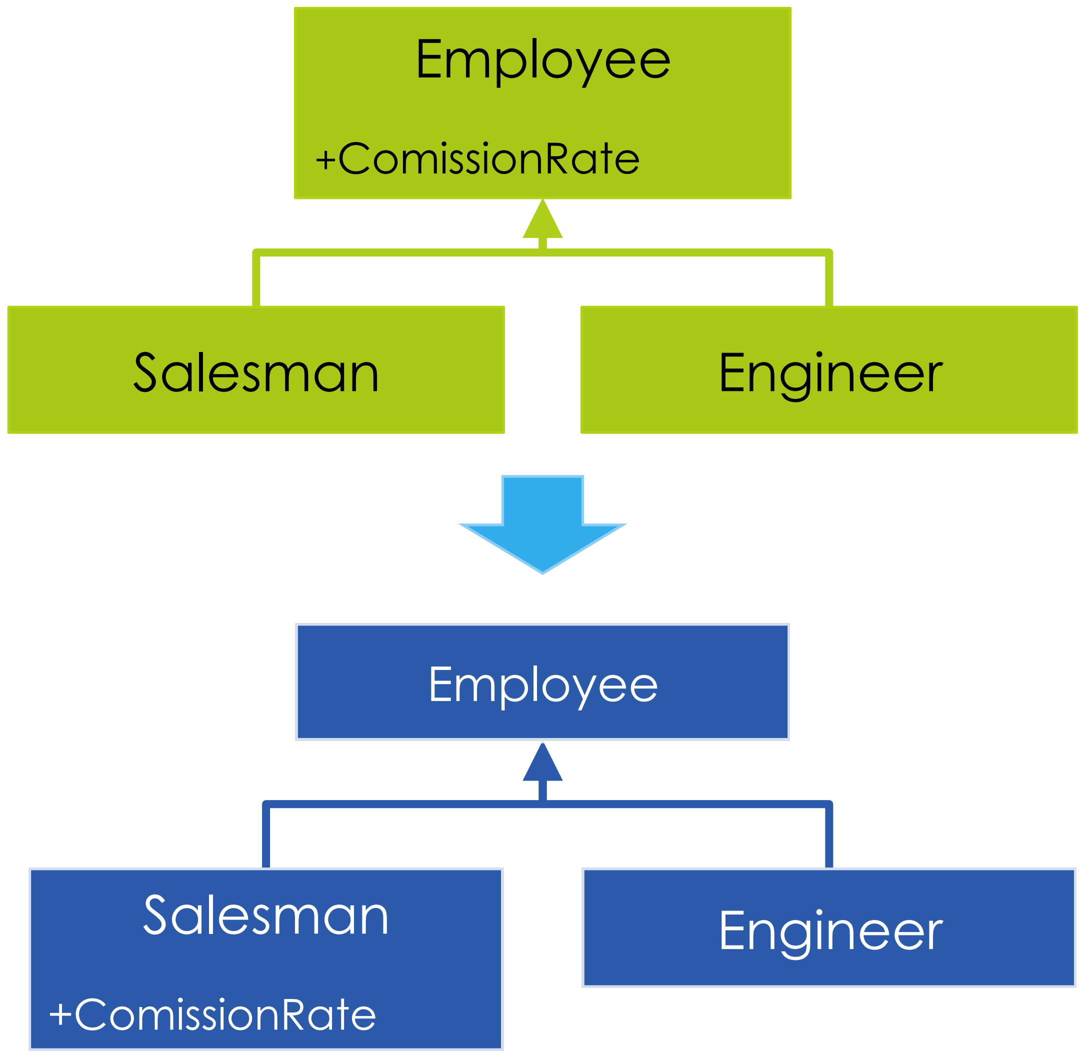
When

- Superclass has method/field used only in one child

Why

- Simplify design

Push Down Field/Method



Extract Subclass

Create subclass for subset of features

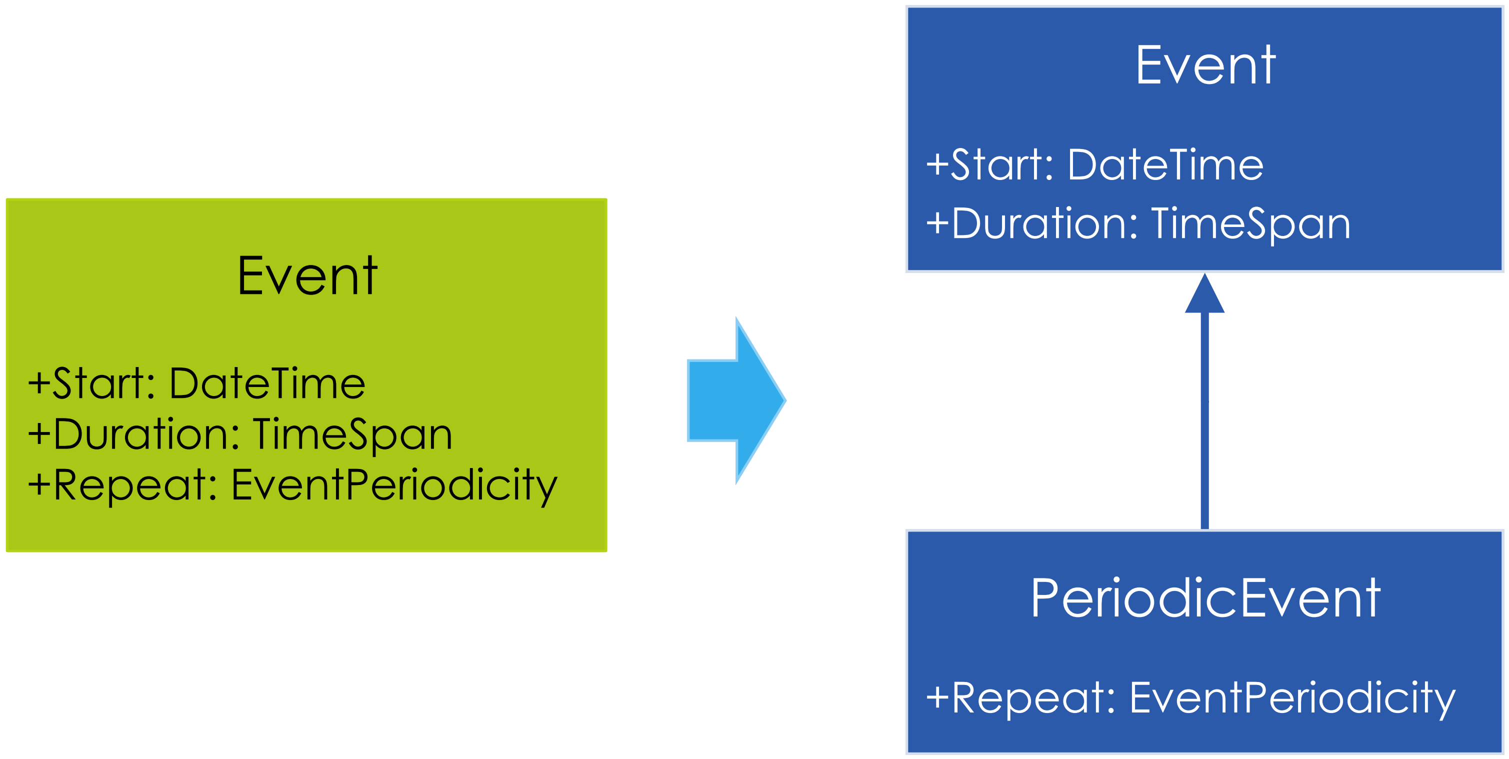
When

- Subset of features used only in some instances

Why

- Improve code readability

Extract Subclass



Extract Superclass

Create superclass and move common features

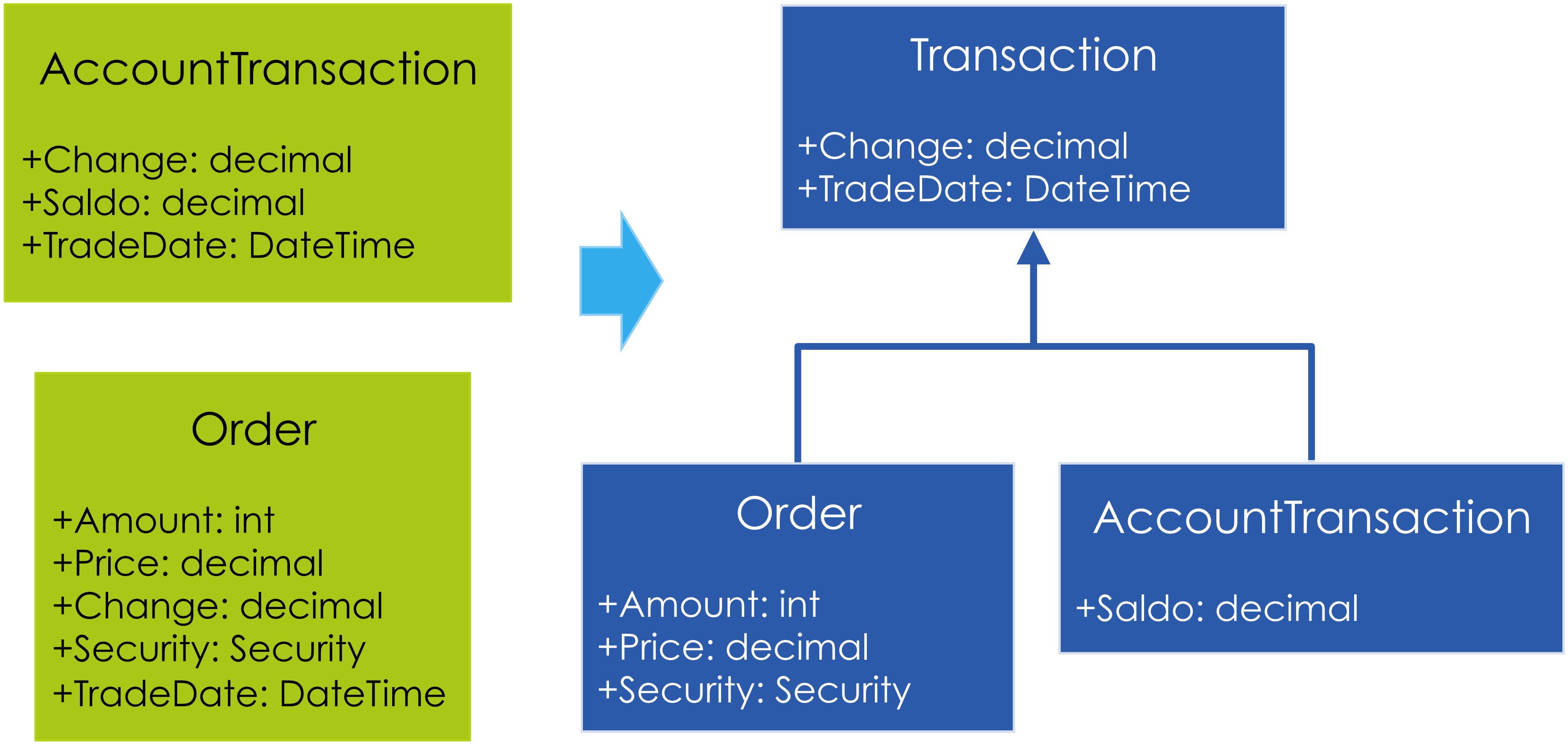
When

- Subset of features used only in some instances

Why

- Remove duplication

Extract Superclass



Replace Inheritance with Delegation

Put superclass to a field and use delegation

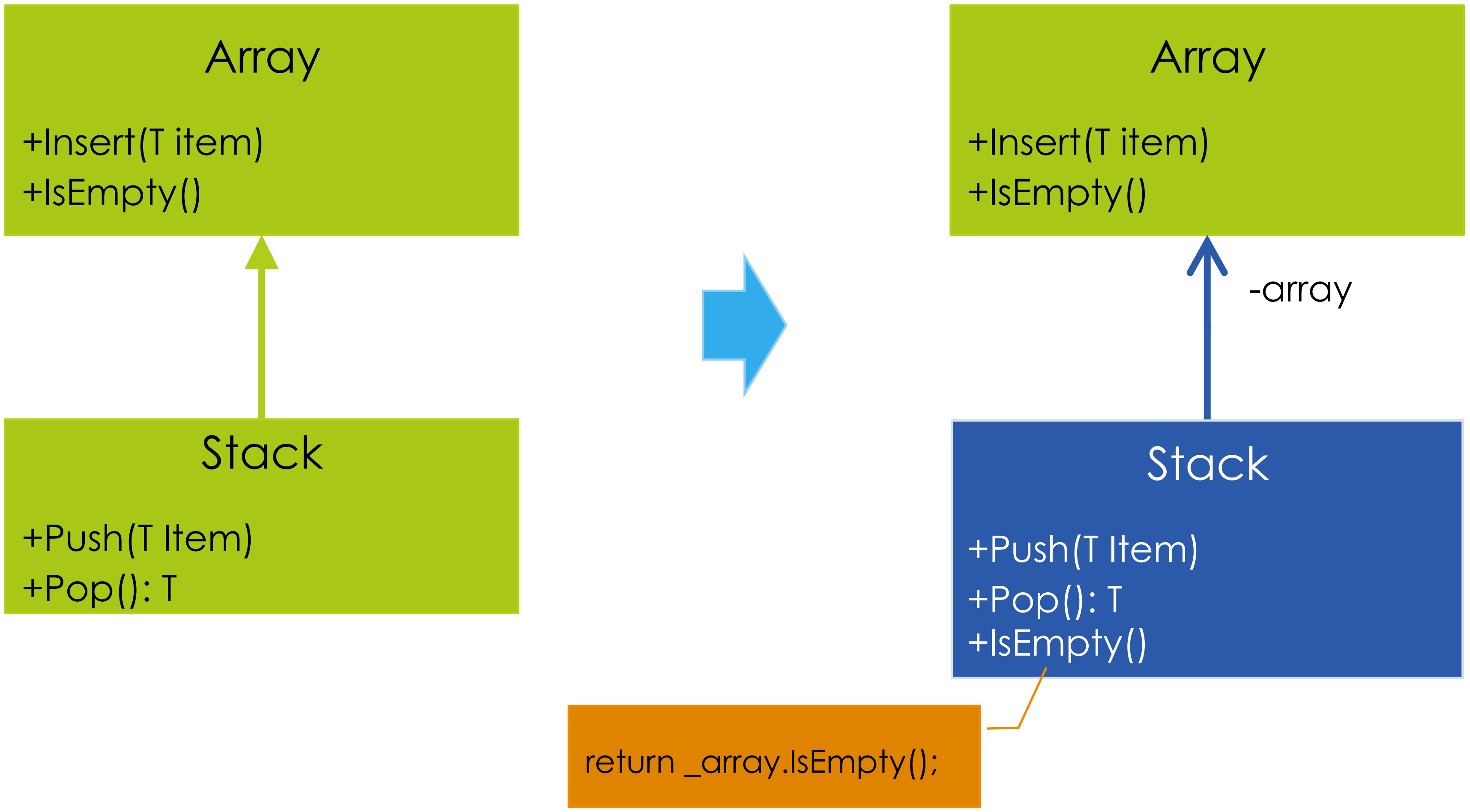
When

- Unable to say “subclass is a superclass”
- Subclass implements superclass interface partially

Why

- Simplify design

Replace Inheritance with Delegation



```
return _array.IsEmpty();
```

Architectural Refactorings

Tease Apart Hierarchies. Extract Hierarchy.
Convert Procedural Design to Objects.
Separate Domain from Presentation.

Tease Apart Inheritance

Create two hierarchies using one another

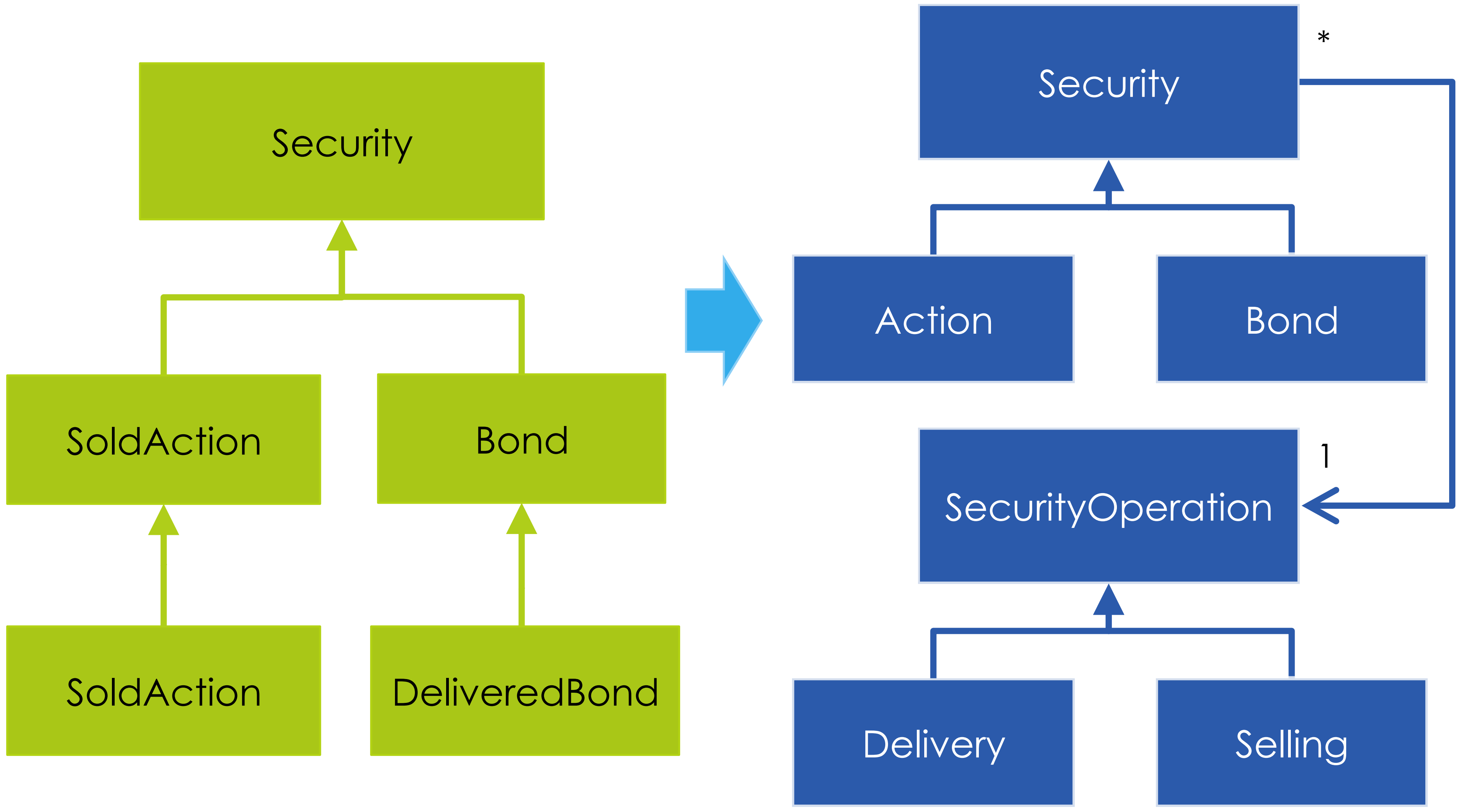
When

- The inheritance hierarchy has two responsibilities

Why

- Simplify design

Tease Apart Inheritance



Extract Hierarchy

Create hierarchy with subclass per special case

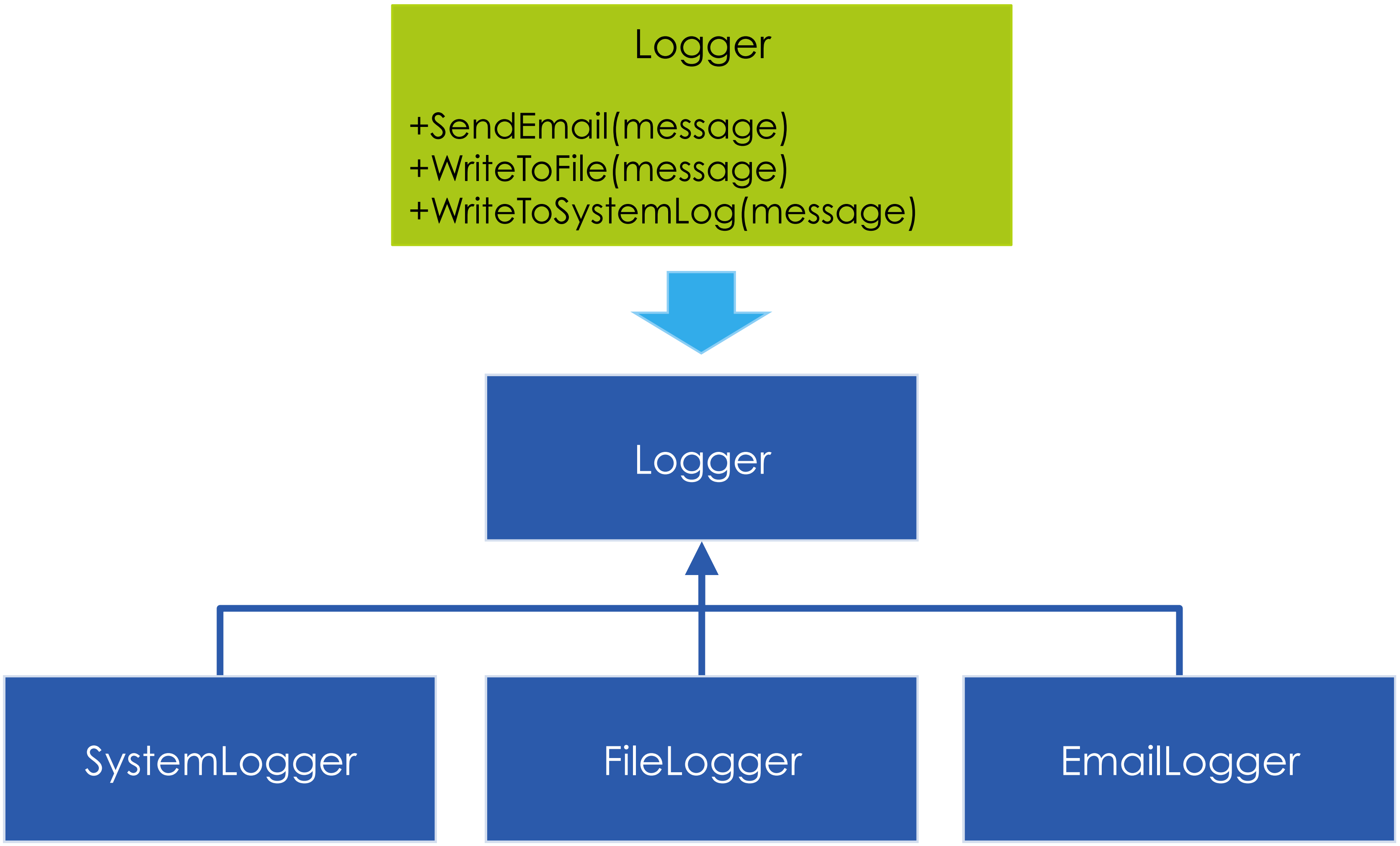
When

- The single class overwhelmed with conditionals

Why

- Improve code readability
- Improve architecture

Extract Hierarchy



Convert Procedural Design to Objects

Turn data into objects and behavior into methods

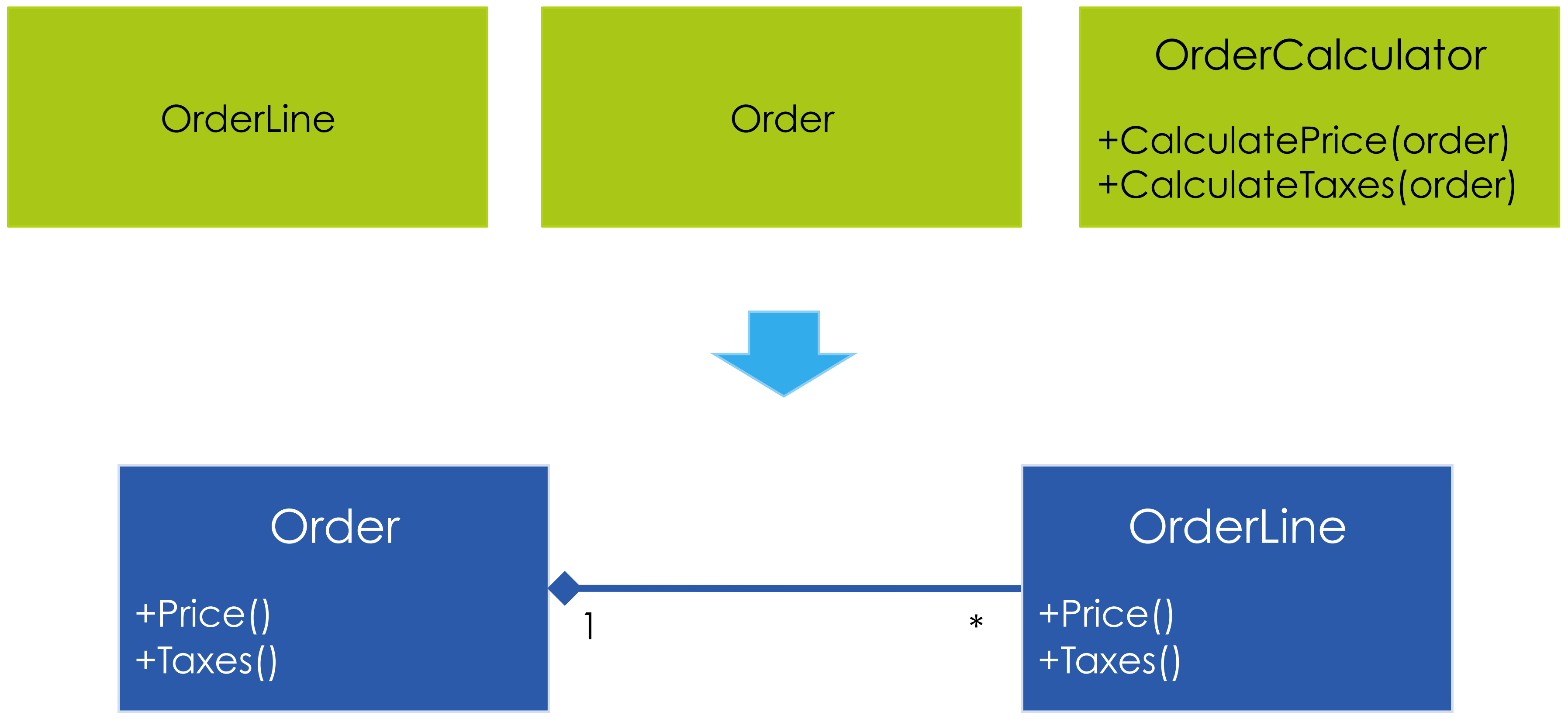
When

- The inheritance hierarchy has two responsibilities

Why

- Improve code readability
- Improve architecture

Convert Procedural Design to Objects



Separate Domain from Presentation

Create domain logic classes separated from ui

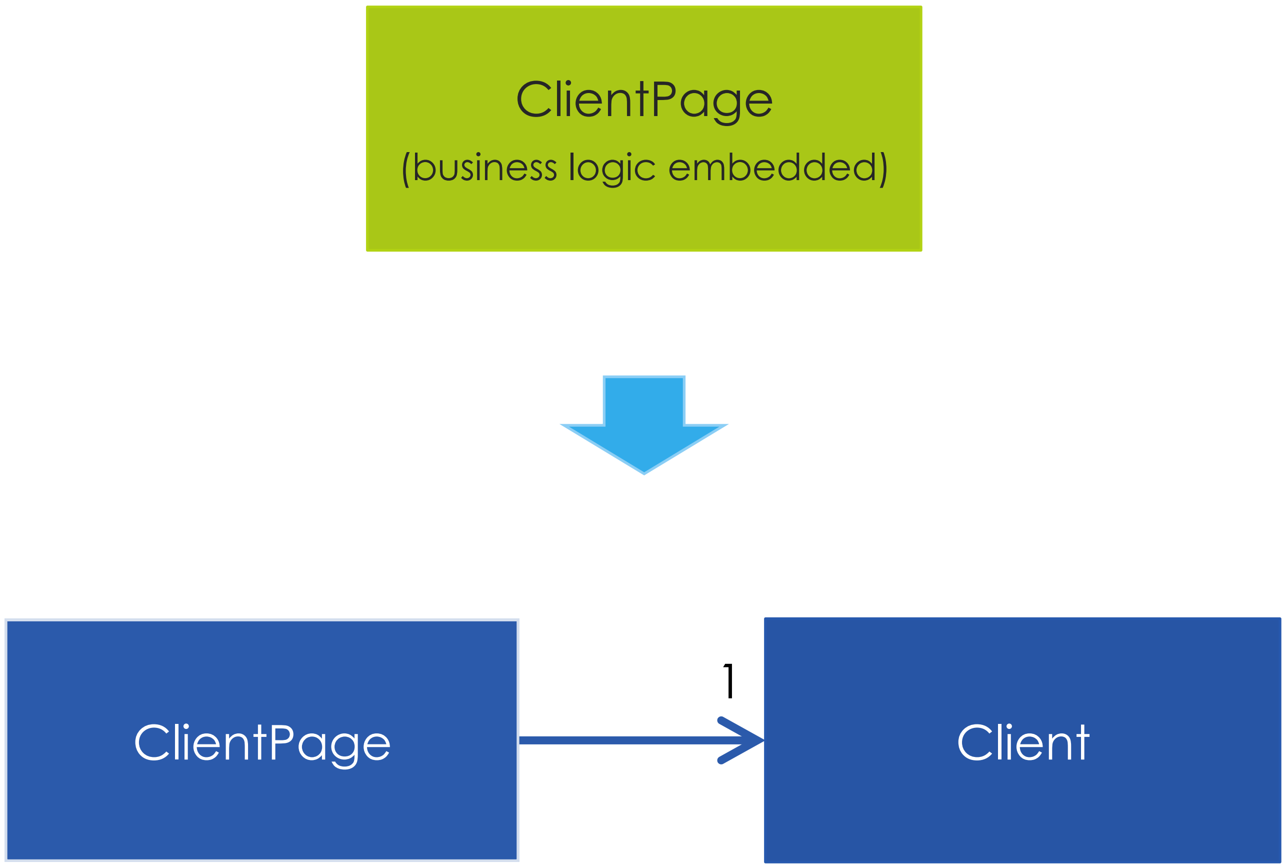
When

- UI logic is mixed with domain logic

Why

- Improve code readability
- Improve architecture

Separate Domain from Presentation



Thank you!

Your questions, please!

References

- Martin Fowler *Refactoring*
- Robert C. Martin *Clean Code*
- Steve McConnell *Code Complete*!