Motivation:

The more lines found in a method, the harder it’s to figure out what the method does. This is the main reason for this refactoring. Besides eliminating rough edges in code, extracting methods is also a step in many other refactoring approaches.

**Before:**

class pizzaShop

{

public:

virtual void printBill() = 0;

};

class BeefPizza: public pizzaShop

{

public:

void printBill()

{

cout << "Pirce is 15 AUD$ \n";

}

};

class VegetablePizza: public pizzaShop

{

public:

void printBill()

{

cout << "Pirce is 13 AUD$ \n";

}

};

class CickenPizza: public pizzaShop

{

public:

void printBill()

{

cout << "Pirce is 12 AUD$ \n";

}

};

int main()

{

vector<pizzaShop\*> roles;

int choice;

while (true)

{

cout << "BeefPizza(1) VegetablePizza(2) CickenPizza(3) Go(0): ";

cin >> choice;

if (choice == 0)

break;

else if (choice == 1)

roles.push\_back(new BeefPizza);

else if (choice == 2)

roles.push\_back(new VegetablePizza);

else

roles.push\_back(new CickenPizza);

}

for (int i = 0; i < roles.size(); i++)

roles[i]->printBill();

for (int i = 0; i < roles.size(); i++)

delete roles[i];

}

**Mechanics:**

In this example passing a reference to the class that will be returning the computation to a new object that has the multiple methods via the constructor or passing the individual parameters to the constructor of the method object.

**After:**

public class OrderLineItem

{

public decimal Price { get; private set; }

}

public class Order

{

public IEnumerable<OrderLineItem> OrderLineItems { get; private set; }

public IEnumerable<decimal> Discounts { get; private set; }

public decimal Tax { get; private set; }

public decimal Calculate()

{

return new OrderCalculator(this).Calculate();

}

}

public class OrderCalculator

{

private decimal SubTotal { get; set; }

private IEnumerable<OrderLineItem> OrderLineItems { get; set; }

private IEnumerable<decimal> Discounts { get; set; }

private decimal Tax { get; set; }

public OrderCalculator(Order order)

{

OrderLineItems = order.OrderLineItems;

Discounts = order.Discounts;

Tax = order.Tax;

}

public decimal Calculate()

{

CalculateSubTotal();

SubtractDiscounts();

CalculateTax();

return SubTotal;

}

private void CalculateSubTotal()

{

// Total up line items

foreach (OrderLineItem lineItem in OrderLineItems)

SubTotal += lineItem.Price;

}

private void SubtractDiscounts()

{

// Subtract Discounts

foreach (decimal discount in Discounts)

SubTotal -= discount;

}

private void CalculateTax()

{

// Calculate Tax

SubTotal += SubTotal \* Tax;

}

}