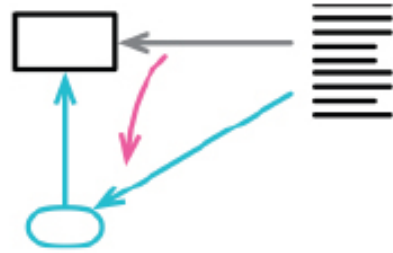


Design Defects & Restructuring

Week 11: 18/19 Nov 2022

Rahim Hasnani

Basic Refactoring: Encapsulate Variable



```
let defaultOwner = {firstName: "Martin", lastName: "Fowler"};
```



```
let defaultOwnerData = {firstName: "Martin", lastName: "Fowler"};  
export function defaultOwner() {return defaultOwnerData;}  
export function setDefaultOwner(arg) {defaultOwnerData = arg;}
```

- ▶ Create encapsulating functions to access and update the variable.
- ▶ For each reference to the variable, replace with a call to the appropriate encapsulating function.
- ▶ Restrict the visibility of the variable.
- ▶ How to prevent updates to data?

Basic Refactoring: Rename Variable

name
nm

```
let a = height * width;
```

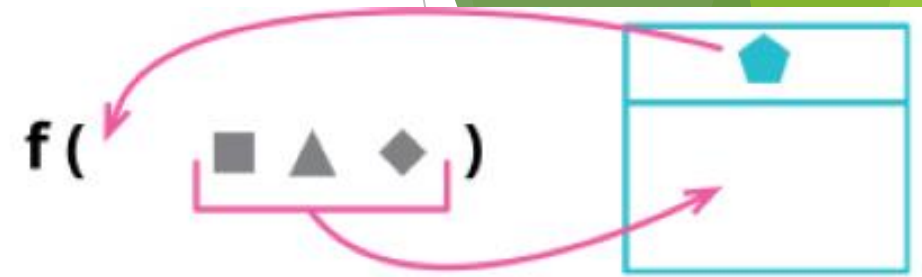


```
let area = height * width;
```

- ▶ If the variable is used widely, consider *Encapsulate Variable*
- ▶ Find all references to the Variable, and change every one

Basic Refactoring: Introduce Parameter Object

- ▶ If there isn't a suitable structure already, create one.
- ▶ Use *Change Function Declaration* (124) to add a parameter for the new structure.
- ▶ Test.
- ▶ Adjust each caller to pass in the correct instance of the new structure. Test after each one.
- ▶ For each element of the new structure, replace the use of the original parameter with the element of the structure. Remove the parameter. Test.



```
function amountInvoiced(startDate, endDate) {...}  
function amountReceived(startDate, endDate) {...}  
function amountOverdue(startDate, endDate) {...}
```



```
function amountInvoiced(aDateRange) {...}  
function amountReceived(aDateRange) {...}  
function amountOverdue(aDateRange) {...}
```

Basic Refactoring: Combine Functions Into Class

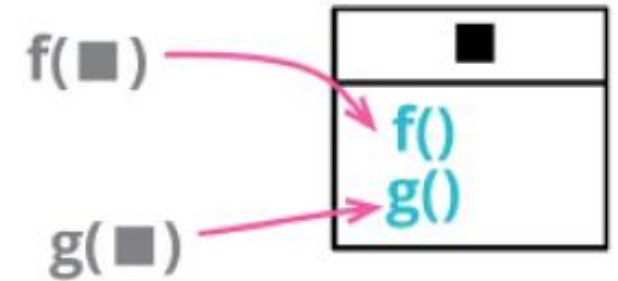
- ▶ Apply *Encapsulate Record* (162) to the common data record that the functions share.

If the data that is common between the functions isn't already grouped into a record structure, use *Introduce Parameter Object* to create a record to group it together.

- ▶ Take each function that uses the common record and use *Move Function* to move it into the new class.

Any arguments to the function call that are members can be removed from the argument list.

- ▶ Each bit of logic that manipulates the data can be extracted with *Extract Function* and then moved into the new class.



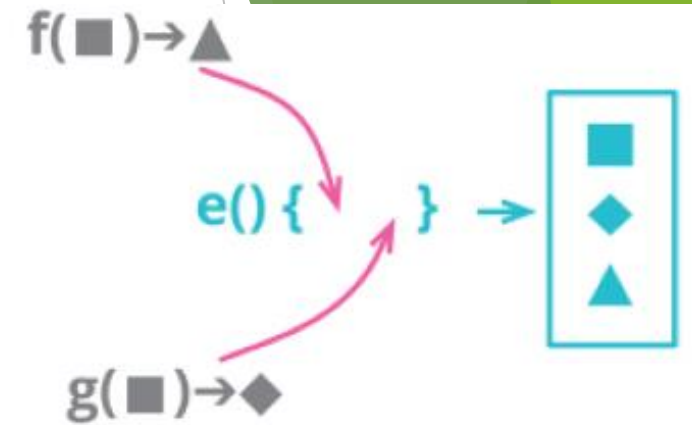
```
function base(aReading) {...}
function taxableCharge(aReading) {...}
function calculateBaseCharge(aReading) {...}
```



```
class Reading {
  base() {...}
  taxableCharge() {...}
  calculateBaseCharge() {...}
}
```

Basic Refactoring: Combine Functions Into Transform

- ▶ Alternative to Combine Functions Into Class
- ▶ Rather than creating a new class, derived fields are added to the structure
- ▶ Problem occurs when data is changed after enriching...



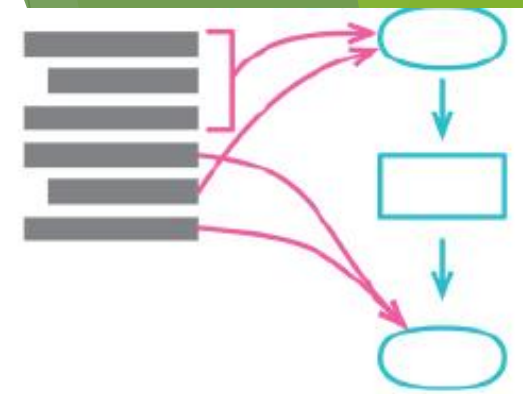
```
function base(aReading) {...}  
function taxableCharge(aReading) {...}
```



```
function enrichReading(argReading) {  
  const aReading = _.cloneDeep(argReading);  
  aReading.baseCharge = base(aReading);  
  aReading.taxableCharge = taxableCharge(aReading);  
  return aReading;  
}
```

Basic Refactoring: Split Phase

- ▶ Extract the second phase code into its own function.
- ▶ Test.
- ▶ Introduce an intermediate data structure as an additional argument to the extracted function.
- ▶ Test.
- ▶ Examine each parameter of the extracted second phase. If it is used by first phase, move it to the intermediate data structure. Test after each move.
- ▶ Apply *Extract Function* on the first-phase code, returning the intermediate data structure.



```
const orderData = orderString.split(/\s+/);
const productPrice = priceList[orderData[0].split("-")[1]];
const orderPrice = parseInt(orderData[1]) * productPrice;
```



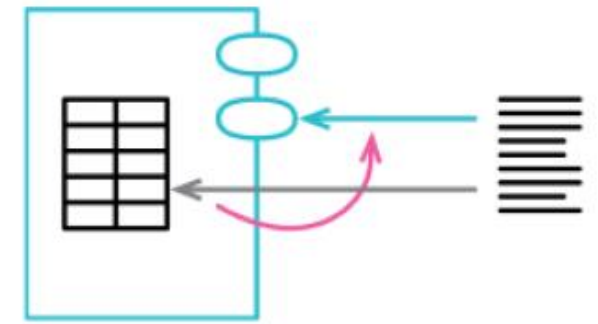
```
const orderRecord = parseOrder(order);
const orderPrice = price(orderRecord, priceList);

function parseOrder(aString) {
  const values = aString.split(/\s+/);
  return ({
    productID: values[0].split("-")[1],
    quantity: parseInt(values[1]),
  });
}

function price(order, priceList) {
  return order.quantity * priceList[order.productID];
}
```


Encapsulate: Encapsulate Record

- ▶ Use *Encapsulate Variable* on the variable holding the record.
Give the functions that encapsulate the record names that are easily searchable.
- ▶ Replace the content of the variable with a simple class that wraps the record. Define an accessor inside this class that returns the raw record. Modify the functions that encapsulate the variable to use this accessor.
- ▶ Test.
- ▶ Provide new functions that return the object rather than the raw record.
- ▶ For each user of the record, replace its use of a function that returns the record with a function that returns the object. Use an accessor on the object to get at the field data, creating that accessor if needed. Test after each change.
- ▶ Remove the class's raw data accessor and the easily searchable functions that returned the raw record.
- ▶ Test.
- ▶ If the fields of the record are themselves structures, consider using Encapsulate Record and *Encapsulate Collection* recursively.



```
organization = {name: "Acme Gooseberries", country: "GB"};
```

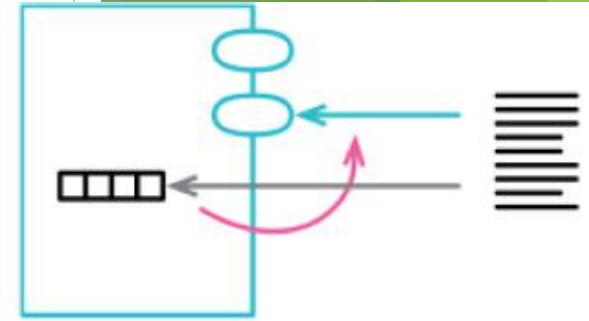


```
class Organization {  
  constructor(data) {  
    this._name = data.name;  
    this._country = data.country;  
  }  
  get name() {return this._name;}  
  set name(arg) {this._name = arg;}  
  get country() {return this._country;}  
  set country(arg) {this._country = arg;}  
}
```


Encapsulate: Encapsulate Collection

- ▶ Apply *Encapsulate Variable* if the reference to the collection isn't already encapsulated.
- ▶ Add functions to add and remove elements from the collection.

If there is a setter for the collection, use *Remove Setting Method* if possible. If not, make it take a copy of the provided collection.
- ▶ Run static checks.
- ▶ Find all references to the collection. If anyone calls modifiers on the collection, change them to use the new add/remove functions. Test after each change.
- ▶ Modify the getter for the collection to return a protected view on it, using a read-only proxy or a copy.
- ▶ Test.



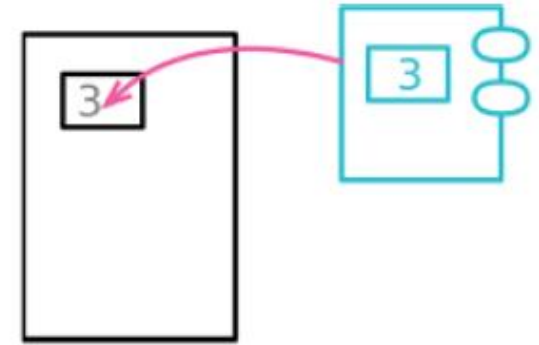
```
class Person {  
  get courses() {return this._courses;}  
  set courses(aList) {this._courses = aList;}  
}
```



```
class Person {  
  get courses() {return this._courses.slice();}  
  addCourse(aCourse) { ... }  
  removeCourse(aCourse) { ... }  
}
```

Encapsulate: Replace Primitive with Object

- ▶ Apply *Encapsulate Variable* if it isn't already.
- ▶ Create a simple value class for the data value. It should take the existing value in its constructor and provide a getter for that value.
- ▶ Run static checks.
- ▶ Change the setter to create a new instance of the value class and store that in the field, changing the type of the field if present.
- ▶ Change the getter to return the result of invoking the getter of the new class.
- ▶ Test.
- ▶ Consider using *Rename Function* on the original accessors to better reflect what they do.
- ▶ Consider clarifying the role of the new object as a value or reference object by applying *Change Reference to Value* or *Change Value to Reference*.



```
orders.filter(o => "high" == o.priority  
    || "rush" == o.priority);
```



```
orders.filter(o => o.priority.higherThan(new Priority("normal")))
```

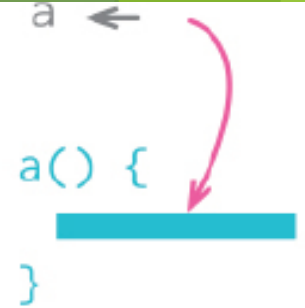
Encapsulate: Replace Temp with Query

- ▶ Check that the variable is determined entirely before it's used, and the code that calculates it does not yield a different value whenever it is used.
- ▶ If the variable isn't read-only, and can be made read-only, do so.
- ▶ Test.
- ▶ Extract the assignment of the variable into a function.

If the variable and the function cannot share a name, use a temporary name for the function.

Ensure the extracted function is free of side effects. If not, use *Separate Query from Modifier*.

- ▶ Test.
- ▶ Use *Inline Variable* to remove the temp.



a ←
a() {
}
The diagram shows a variable 'a' with an arrow pointing to a function definition 'a() { }'. A pink arrow points from the function definition to the variable 'a'.

```
const basePrice = this._quantity * this._itemPrice;  
if (basePrice > 1000)  
  return basePrice * 0.95;  
else  
  return basePrice * 0.98;
```

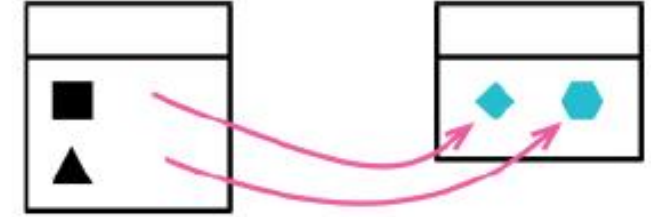


```
get basePrice() {this._quantity * this._itemPrice;}  
  
...  
  
if (this.basePrice > 1000)  
  return this.basePrice * 0.95;  
else  
  return this.basePrice * 0.98;
```

Encapsulate: Extract Class

- ▶ Decide how to split the responsibilities of the class.
- ▶ Create a new child class to express the split-off responsibilities.

If the responsibilities of the original parent class no longer match its name, rename the parent.
- ▶ Create an instance of the child class when constructing the parent and add a link from parent to child.
- ▶ Use *Move Field* on each field you wish to move. Test after each move.
- ▶ Use *Move Function* to move methods to the new child. Start with lower-level methods (those being called rather than calling). Test after each move.
- ▶ Review the interfaces of both classes, remove unneeded methods, change names to better fit the new circumstances.
- ▶ Decide whether to expose the new child. If so, consider applying *Change Reference to Value* to the child class.



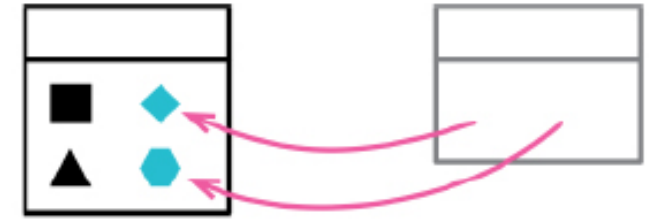
```
class Person {  
    get officeAreaCode() {return this._officeAreaCode;}  
    get officeNumber()   {return this._officeNumber;}  
}
```



```
class Person {  
    get officeAreaCode() {return this._telephoneNumber.areaCode;}  
    get officeNumber()   {return this._telephoneNumber.number;}  
}  
class TelephoneNumber {  
    get areaCode() {return this._areaCode;}  
    get number()   {return this._number;}  
}
```

Encapsulate: Inline Class

- ▶ In the target class, create functions for all the public functions of the source class.
- ▶ These functions should just delegate to the source class.
- ▶ Change all references to source class methods so they use the target class's delegators instead. Test after each change.
- ▶ Move all the functions and data from the source class into the target, testing after each move, until the source class is empty.
- ▶ Delete the source class and hold a short, simple funeral service.



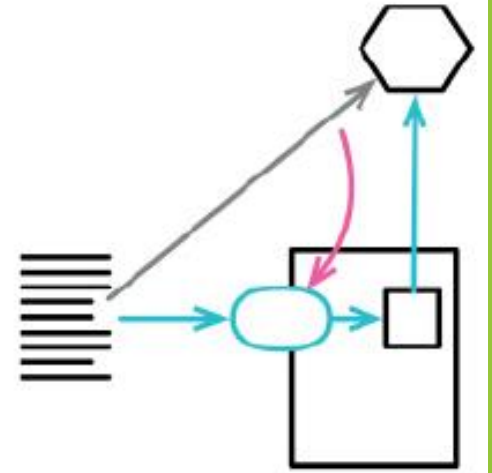
```
class Person {  
    get officeAreaCode() {return this._telephoneNumber.areaCode;}  
    get officeNumber()   {return this._telephoneNumber.number;}  
}  
class TelephoneNumber {  
    get areaCode() {return this._areaCode;}  
    get number()   {return this._number;}  
}
```



```
class Person {  
    get officeAreaCode() {return this._officeAreaCode;}  
    get officeNumber()   {return this._officeNumber;}  
}
```

Encapsulate: Hide Delegate

- ▶ For each method on the delegate, create a simple delegating method on the server.
- ▶ Adjust the client to call the server. Test after each change.
- ▶ If no client needs to access the delegate anymore, remove the server's accessor for the delegate.
- ▶ Test.



```
manager = aPerson.department.manager;
```



```
manager = aPerson.manager;
```

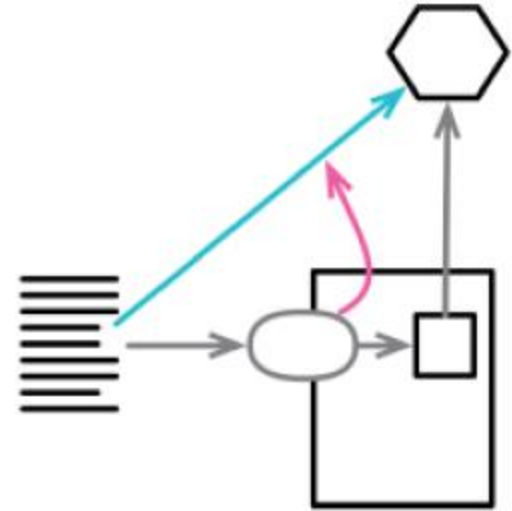
```
class Person {
    get manager() {return this.department.manager;}
```


Encapsulate: Remove Middle Man

- ▶ Create a getter for the delegate.
- ▶ For each client use of a delegating method, replace the call to the delegating method by chaining through the accessor. Test after each replacement.

If all calls to a delegating method are replaced, you can delete the delegating method.

With automated refactorings, you can use *Encapsulate Variable* on the delegate field and then *Inline Function* on all the methods that use it.



```
manager = aPerson.manager;
```

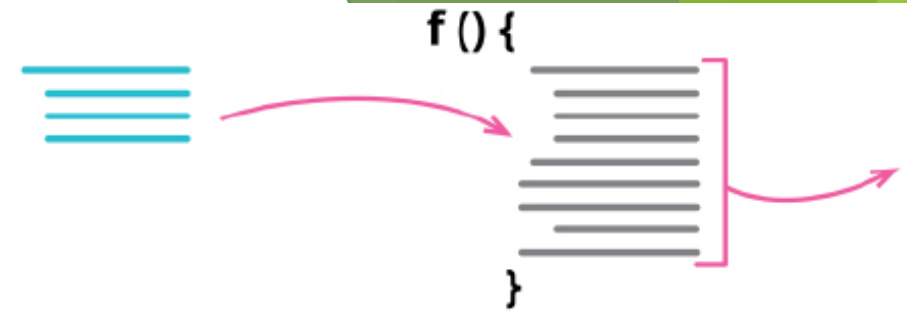
```
class Person {
    get manager() {return this.department.manager;}
}
```



```
manager = aPerson.department.manager;
```


Encapsulate: Substitute Algorithm

- ▶ Arrange the code to be replaced so that it fills a complete function.
- ▶ Prepare tests using this function only, to capture its behavior.
- ▶ Prepare your alternative algorithm.
- ▶ Run static checks.
- ▶ Run tests to compare the output of the old algorithm to the new one. If they are the same, you're done. Otherwise, use the old algorithm for comparison in testing and debugging.



```
function foundPerson(people) {  
  for(let i = 0; i < people.length; i++) {  
    if (people[i] === "Don") {  
      return "Don";  
    }  
    if (people[i] === "John") {  
      return "John";  
    }  
    if (people[i] === "Kent") {  
      return "Kent";  
    }  
  }  
  return "";  
}
```



```
function foundPerson(people) {  
  const candidates = ["Don", "John", "Kent"];  
  return people.find(p => candidates.includes(p)) || '';  
}
```

Exercise

- ▶ Carry out refactoring