



Advanced Refactorings

Michael Inden

Agenda



- **Checklist and example**
 - **Catalog of «bigger» refactorings**
 - **Defensive programming**
-



Checklist and example



Checklist



1. **Run Tests** – Run existing unit tests. In particular, this should be done continuously after each of the following steps.
2. **Apply Coding Conventions** – Follow the coding conventions:
 - Format the source code.
 - If possible, define passing parameters and local variables `final`.
 - Ensure comprehensible constant, variable and method names by the basic refactoring *RENAME*
3. **Break down into individual parts** – If necessary and useful, first break down a more complex section of the program into manageable and useful smaller parts by using (several times) the basic refactoring *EXTRACT METHOD*

Checklist



4. Write unit tests* – If requirements are known from the business, the customer or from a requirements document, you should use them to create test cases. Otherwise, write corresponding tests for previously extracted methods:

- For normal inputs
- For invalid inputs and for edge or special cases.

5. Clean Up – Clean up the source code by:

- Removing unused variables and unused methods.
- Removing old, unnecessary or (meanwhile) wrong comments.

*ideally you start writing some pin down tests before you modify unknown code to ensure not to introduce undesired behavioural changes



6. Simplify unit tests – Reduce complexity

- Remove duplicated source code, create helper methods if necessary.
- Simplify conditions.
- Add explanatory comments if necessary often prefer to consider using descriptive identifiers for attributes and methods.

7. Perform concrete refactorings unit tests – improve the source code by using a refactoring from the refactoring catalog that is appropriate for the identified weakness.

The individual steps are not required to be performed slavishly and exactly in this order. It is quite possible to vary the sequence and to omit some steps or possibly even to perform them several times.



Applying checklist

Initial code ... yes I found it like that!

```
catch (final Fault aF)
{
    final String stSrc = aF.getSource();
    final String stErr = aF.getFaultCode();
    if (stErr != null && stErr.equalsIgnoreCase("FileNotFoundException"))
    {
        stErrorMsg = aF.getFaultString() + ": " + stSrc;
    }
    else
    {
        stErrorMsg = aF.getFaultString() + ": " + stSrc;
    }
}
```

LET'S DO SOME ANALYSIS ...



Applying checklist

Step 2: Apply Coding Conventions: Prefixes and Naming

- Nowadays, prefixes should be used rarely or better avoided altogether – but when reworking an existing part of a program, it is advisable not to turn everything upside down and to follow the existing style to some extent.

```
catch (final Fault fault)
{
    final String strSource = fault.getSource();
    final String strFaultCode = fault.getFaultCode();
    if (strFaultCode != null && strFaultCode.equalsIgnoreCase("FileNotFoundException"))
    {
        this.strErrorMsg = fault.getFaultString() + ": " + strSource;
    }
    else
    {
        this.strErrorMsg = fault.getFaultString() + ": " + strSource;
    }
}
```



Applying checklist

Step 6: Simplify

- In this simple example, you will find an exact duplication of the statements in the if and else block.

```
catch (final Fault fault)
{
    final String strSource = fault.getSource();
    final String strFaultCode = fault.getFaultCode();

    this.strErrorMsg = fault.getFaultString() + ":" + strSource;
}
```



Applying checklist

Step 5: Clean Up

- As a result of these simplifications it becomes obvious that the variable `strFaultCode` is now unused and thus superfluous. Thus it can be dropped. Also the variable `strSource` determines only data from the passed fault object.

```
catch (final Fault fault)
{
    this.strErrorMsg = fault.getFaultString() + ":" + fault.getSource();
}
```



Conclusion

- The result is impressive: Ten lines of source code have become one line. Just as bad smells spread when you are careless, there is fortunately also an opposite effect: **the more you ensure clarity and structure, the easier it is to make further improvements.**
 - However, we have not been able to remove one problem: There is still an assignment to the strErrorMsg attribute in the catch block. Such a side effect should be avoided, since state changes occur at unexpected places in the program. **During refactorings we normally do not want to change the externally visible program behavior, but only improve the inner structure.**
-



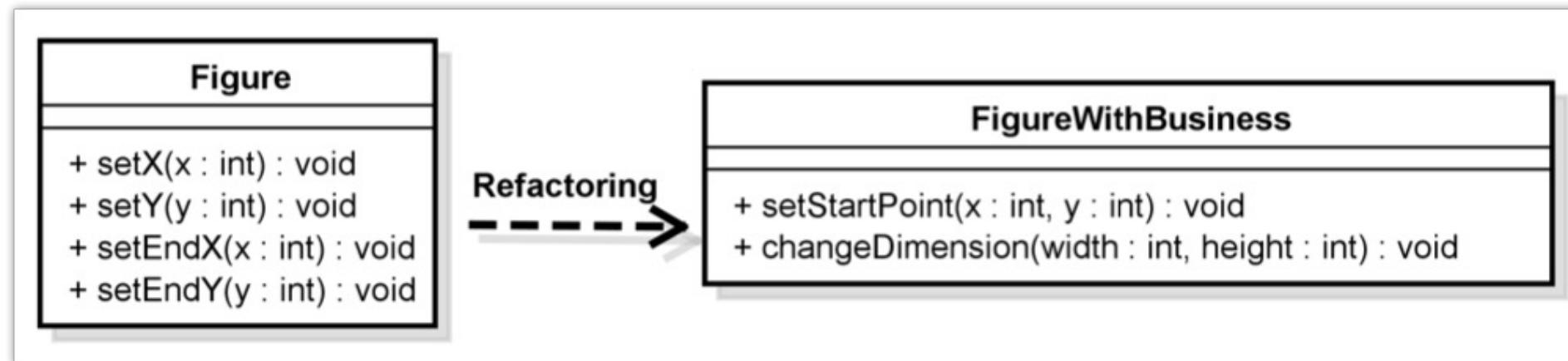
Catalog of «bigger» refactorings



Replace Mutator with Business Method



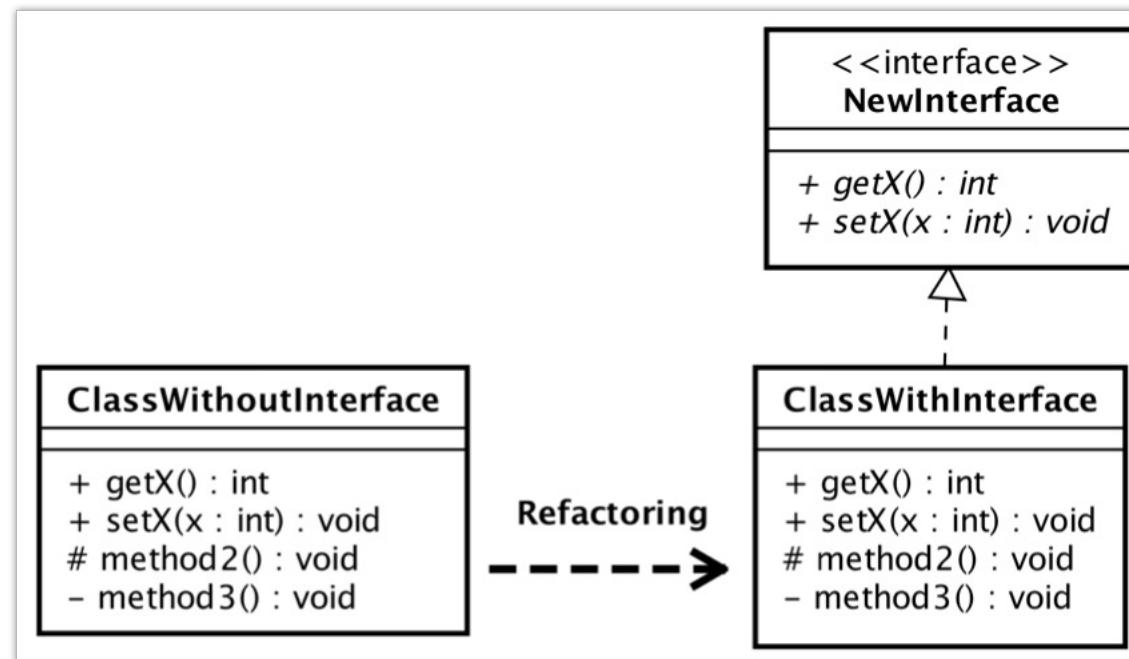
- This refactoring helps to protect the state of an object from fine-grained changes and addresses the object-oriented idea of defining behavior through business methods.



Introduce an interface



- Turn an existing class without an explicit interface* into a class that offers an interface.* With this refactoring we bundle the desired public methods in an interface.
- Pay attention: this does not make sense in every case, for example for only internal classes, but it does for classes with a public interface.

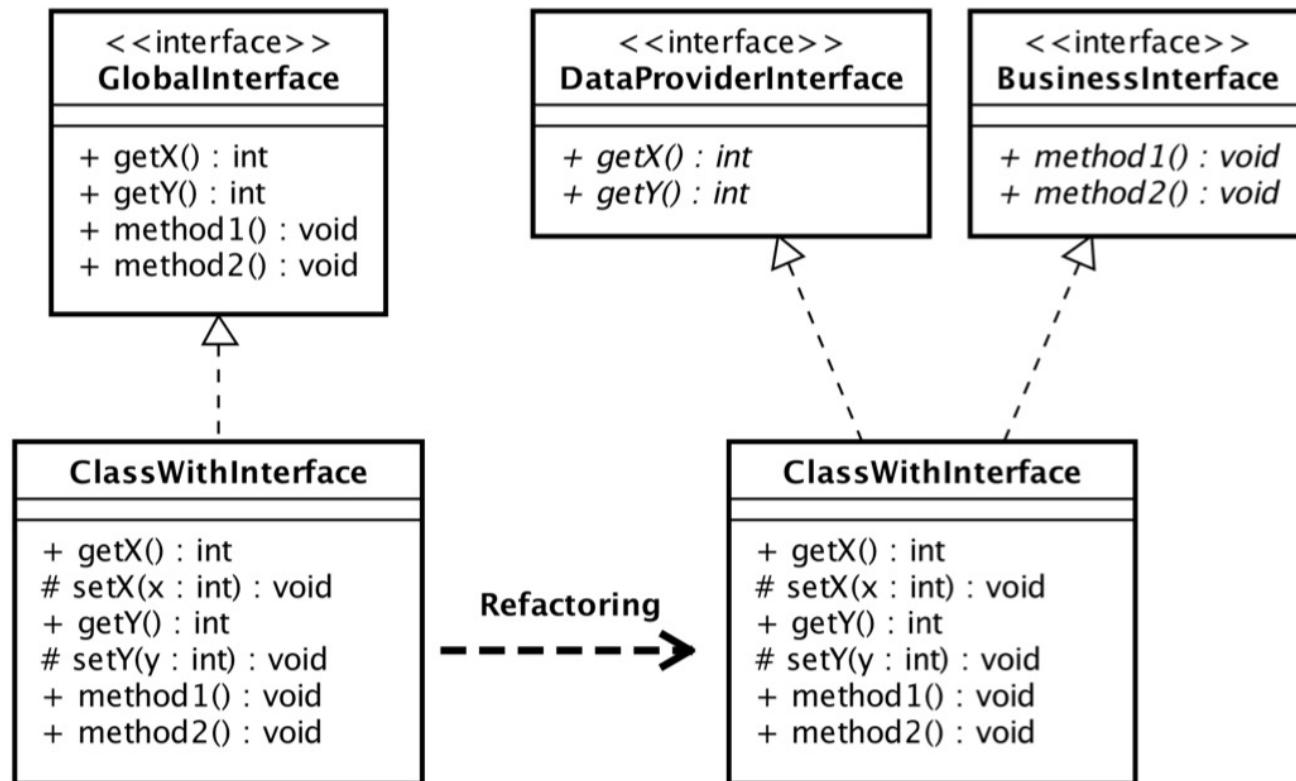


*every class has an interface, namely the public and protected methods, but explicit interface uses keyword interface

Split interface



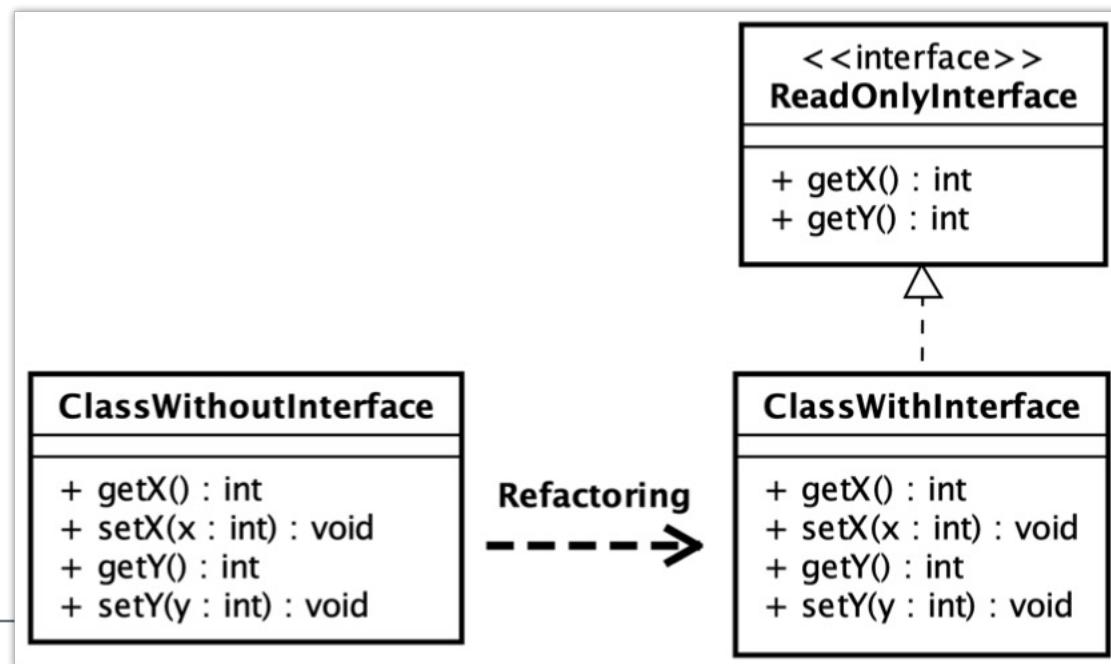
- Sometimes during development you realize that an interface should better be split into two or more interfaces.
- As an example, the interface `GlobalInterface` will be split in the following, since it offers business functionality in addition to pure read access.



Introduce a read-only interface



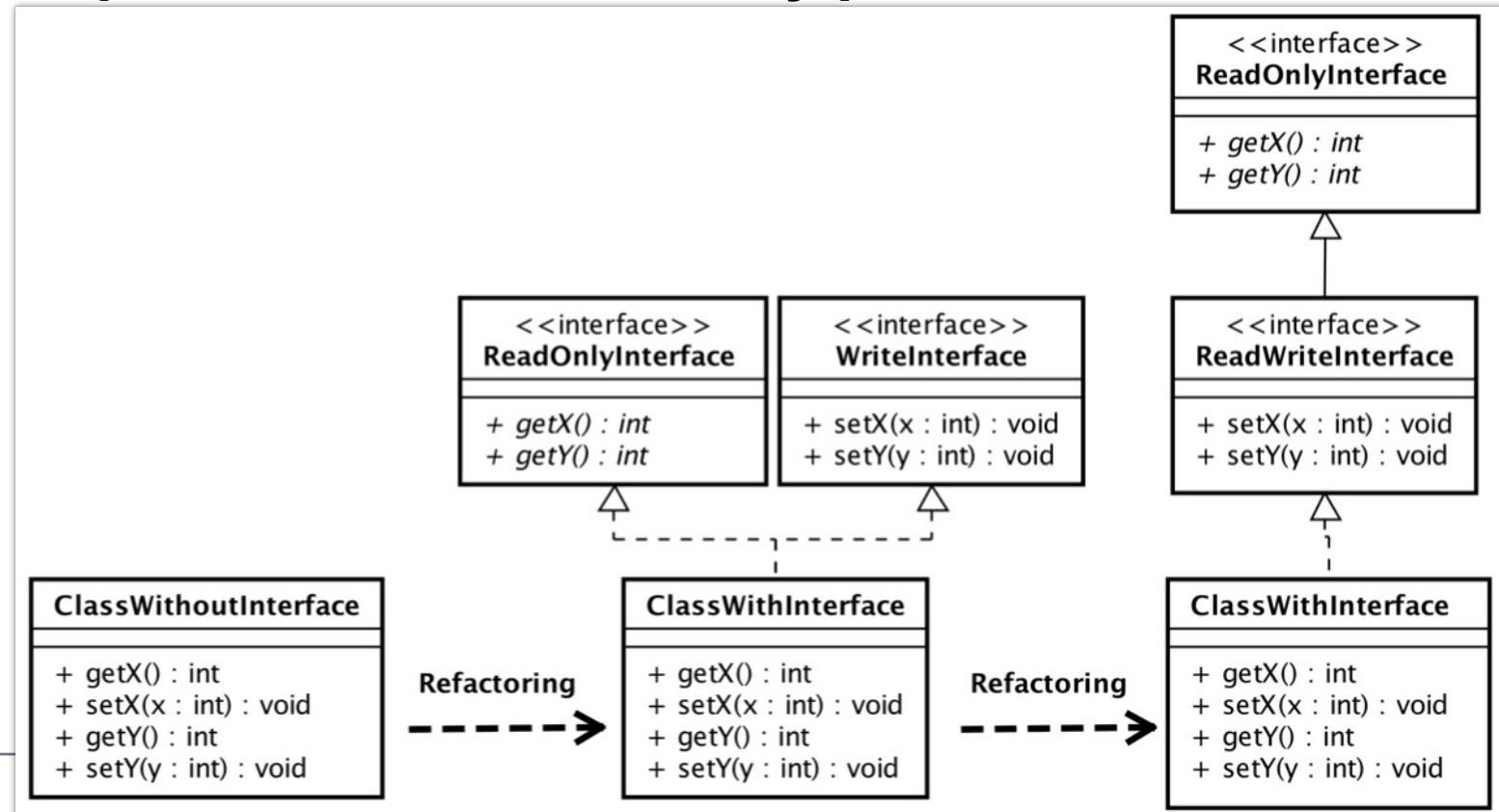
- In more complex systems, providing read-only interfaces can be useful to protect against unwanted changes by other components.
- Without access protection, the model data could be modified not only internally, but also by external clients. To prevent this, one can introduce a read-only interface



Introduce a read-write interface



- **Read-only interfaces can be used to prevent write access by package-external classes. Some components should have write access, but under the premise without direct access to concrete objects. In this case, you can define a special interface that only provides write access:**





Separate information acquisition / retrival and processing



Separate information acquisition / retrieval and processing



- In many cases, it is advantageous to separate the acquisition of information from its processing and storage.
 - These subtasks can generally be outsourced quite well to separate methods, which increases orthogonality (combinability) and reusability and avoids invalid intermediate states.
-

Separate information acquisition / retrieval and processing



- Five `set()` calls are distributed all over the method. This seems harmless.

```
public void updateModelFromElement(final DataElementV0 elementV0)
{
    if (elementV0 != null) {
        if (elementV0.getQueueStatus() != null) {
            model.setState(elementV0.getQueueStatus());
        }
        else {
            model.setState(QueueStates.UNKNOWN);
        }
        model.setQueuedJobs(elementV0.getEntryCount());
    }
    else {
        model.setState(QueueStates.UNREACHABLE);
        model.setQueuedJobs(0);
    }
}
```

- However, one should keep in mind that in principle an error can occur with every method call and as a consequence an exception can be thrown. This can lead to inconsistencies due to partially initialized objects.

Separate information acquisition / retrieval and processing



- Five set() calls are distributed all over the method. This seems harmless.

```
public void updateModelFromElement(final DataElementV0 elementV0)
{
    if (elementV0 != null) {
        if (elementV0.getQueueStatus() != null) {
            model.setState(elementV0.getQueueStatus());
        }
        else {
            model.setState(QueueStates.UNKNOWN);
        }
        model.setQueuedJobs(elementV0.getEntryCount());
    }
    else {
        model.setState(QueueStates.UNREACHABLE);
        model.setQueuedJobs(0);
    }
}
```

- However, one should keep in mind that in principle an error can occur with every method call and as a consequence an exception can be thrown. This can lead to inconsistencies due to partially initialized objects.

Separate information acquisition / retrieval and processing

Step 1: Detect changing elements and introduce auxiliary variables



- in the if- and else respectively two variables are changed by set() calls.

```
public void updateModelFromElement(final DataElementV0 elementV0)
{
    if (elementV0 != null) {
        if (elementV0.getQueueStatus() != null) {
            model.setState(elementV0.getQueueStatus());
        }
        else {
            model.setState(QueueStates.UNKNOWN);
        }
        model.setQueuedJobs(elementV0.getEntryCount());
    }
    else {
        model.setState(QueueStates.UNREACHABLE);
        model.setQueuedJobs(0);
    }
}
```

- Introduce corresponding auxiliary variables state and entryCount. The basic refactoring EXTRACT LOCAL VARIABLE helps us with this.

Separate information acquisition / retrieval and processing

Step 1: Detect changing elements and introduce auxiliary variables



- Introduce corresponding auxiliary variables state and entryCount. The basic refactoring EXTRACT LOCAL VARIABLE helps us with this

```
public void updateModelFromElement(final DataElementV0 elementV0) {  
    if (elementV0 != null) {  
        QueueStates state = elementV0.getQueueStatus();  
        if (queueStatus != null) {  
            model.setState(state);  
        } else {  
            QueueStates state2 = QueueStates.UNKNOWN;  
            model.setState(state2);  
        }  
        int entryCount = elementV0.getEntryCount();  
        model.setQueuedJobs(entryCount);  
    } else {  
        QueueStates state = QueueStates.UNREACHABLE;  
        model.setState(state);  
        int entryCount = 0;  
        model.setQueuedJobs(entryCount);  
    }  
}
```



Separate information acquisition / retrieval and processing

Step 2: Combine the set() methods



- Move some lines up and down to group read / write access

```
public void updateModelFromElement(final DataElementVO elementVO) {  
    if (elementVO != null) {  
        QueueStates state = elementVO.getQueueStatus();  
        if (state == null) {  
            state = QueueStates.UNKNOWN;  
        }  
        int entryCount = elementVO.getEntryCount();  
  
        model.setState(state);  
        model.setQueuedJobs(entryCount);  
    }  
    else {  
        QueueStates state = QueueStates.UNREACHABLE;  
        int entryCount = 0;  
  
        model.setState(state);  
        model.setQueuedJobs(entryCount);  
    }  
}
```

Separate information acquisition / retrieval and processing

Step 3: Separate information acquisition / retrieval and processing



- Move some lines up and down to group read / write access

```
public void updateModelFromElement(final DataElementVO elementVO) {  
    if (elementVO != null) {  
        QueueStates state = elementVO.getQueueStatus();  
        if (state == null) {  
            state = QueueStates.UNKNOWN;  
        }  
        int entryCount = elementVO.getEntryCount();  
  
        model.setState(state);  
        model.setQueuedJobs(entryCount);  
    }  
    else {  
        QueueStates state = QueueStates.UNREACHABLE;  
        int entryCount = 0;  
  
        model.setState(state);  
        model.setQueuedJobs(entryCount);  
    }  
}
```

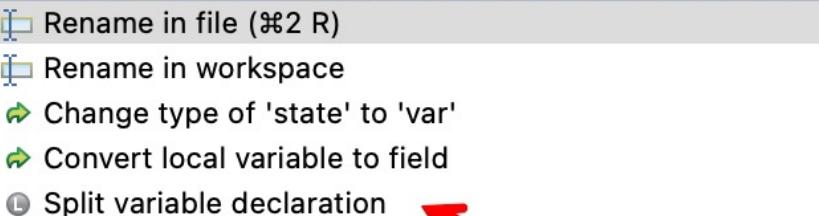
GOAL: Move them to the end

Separate information acquisition / retrieval and processing

Step 3: Separate information acquisition / retrieval and processing



```
public void updateModelFromElement(final DataElementVO elementVO) {  
    if (elementVO != null) {  
        QueueStates state = elementVO.getQueueStatus();  
        if (state == Q  
            state = Q  
        }  
        int entryCount  
  
        model.setState(state);  
        model.setQueuedJobs(entryCount);  
    }  
}
```



Separate information acquisition / retrieval and processing

Step 3: Separate information acquisition / retrieval and processing



```
public void updateModelFromElement(final DataElementVO elementVO) {  
    if (elementVO != null) {  
        QueueStates state;  
        state = elementVO.getQueueStatus();  
        if (state == null) {  
            state = QueueStates.UNKNOWN;  
        }  
        int entryCount;  
        entryCount = elementVO.getEntryCount();  
  
        model.setState(state);  
        model.setQueuedJobs(entryCount);  
    } else {  
        QueueStates state;  
        state = QueueStates.UNREACHABLE;  
        int entryCount;  
        entryCount = 0;  
  
        model.setState(state);  
        model.setQueuedJobs(entryCount);  
    }  
}
```

Separate information acquisition / retrieval and processing

Step 3: Separate information acquisition / retrieval and processing



```
public void updateModelFromElement(final DataElementVO elementVO) {  
    if (elementVO != null) {  
        QueueStates state;  
        state = elementVO.getQueueStatus();  
        if (state == null) {  
            state = QueueStates.UNKNOWN;  
        }  
        int entryCount;  
        entryCount = elementVO.getEntryCount();  
  
        model.setState(state);  
        model.setQueuedJobs(entryCount);  
    } else {  
        QueueStates state;  
        state = QueueStates.UNREACHABLE;  
        int entryCount;  
        entryCount = 0;  
  
        model.setState(state);  
        model.setQueuedJobs(entryCount);  
    }  
}
```

Separate information acquisition / retrieval and processing

Step 3: Separate information acquisition / retrieval and processing



```
› public void updateModelFromElement(final DataElementVO elementVO) {  
    QueueStates state;  
    int entryCount;  
  
    if (elementVO != null) {  
        state = elementVO.getQueueStatus();  
        if (state == null) {  
            state = QueueStates.UNKNOWN; information retrieval  
        }  
        entryCount = elementVO.getEntryCount();  
  
        model.setState(state); processing  
        model.setQueuedJobs(entryCount);  
    } else {  
        QueueStates state;  
        state = QueueStates.UNREACHABLE; information retrieval  
        int entryCount;  
        entryCount = 0;  
  
        model.setState(state); processing  
        model.setQueuedJobs(entryCount);  
    }  
}
```

Separate information acquisition / retrieval and processing

Step 3: Separate information acquisition / retrieval and processing



```
public void updateModelFromElement(final DataElementVO elementVO) {  
    QueueStates state;  
    int entryCount;  
  
    if (elementVO != null) {  
        state = elementVO.getQueueStatus();  
        if (state == null) {  
            state = QueueStates.UNKNOWN;  
        }  
        entryCount = elementVO.getEntryCount();  
    } else {  
        state = QueueStates.UNREACHABLE;  
        entryCount = 0;  
    }  
  
    model.setState(state);  
    model.setQueuedJobs(entryCount);  
}
```

information retrieval

processing

Separate information acquisition / retrieval and processing

Step 4: Encapsulate information gathering and processing



```
public void updateModelFromElement(final DataElementVO elementVO) {  
    QueueStates state;  
    int entryCount;  
  
    if (elementVO != null) {  
        state = elementVO.getQueueStatus();  
        if (state == null) {  
            state = QueueStates.UNKNOWN;  
        }  
        entryCount = elementVO.getEntryCount();  
    } else {  
        state = QueueStates.UNREACHABLE;  
        entryCount = 0;  
    }  
  
    storeInModel(state, entryCount);  
}  
  
public void storeInModel(QueueStates state, int entryCount) {  
    model.setState(state);  
    model.setQueuedJobs(entryCount);  
}
```

Separate information acquisition / retrieval

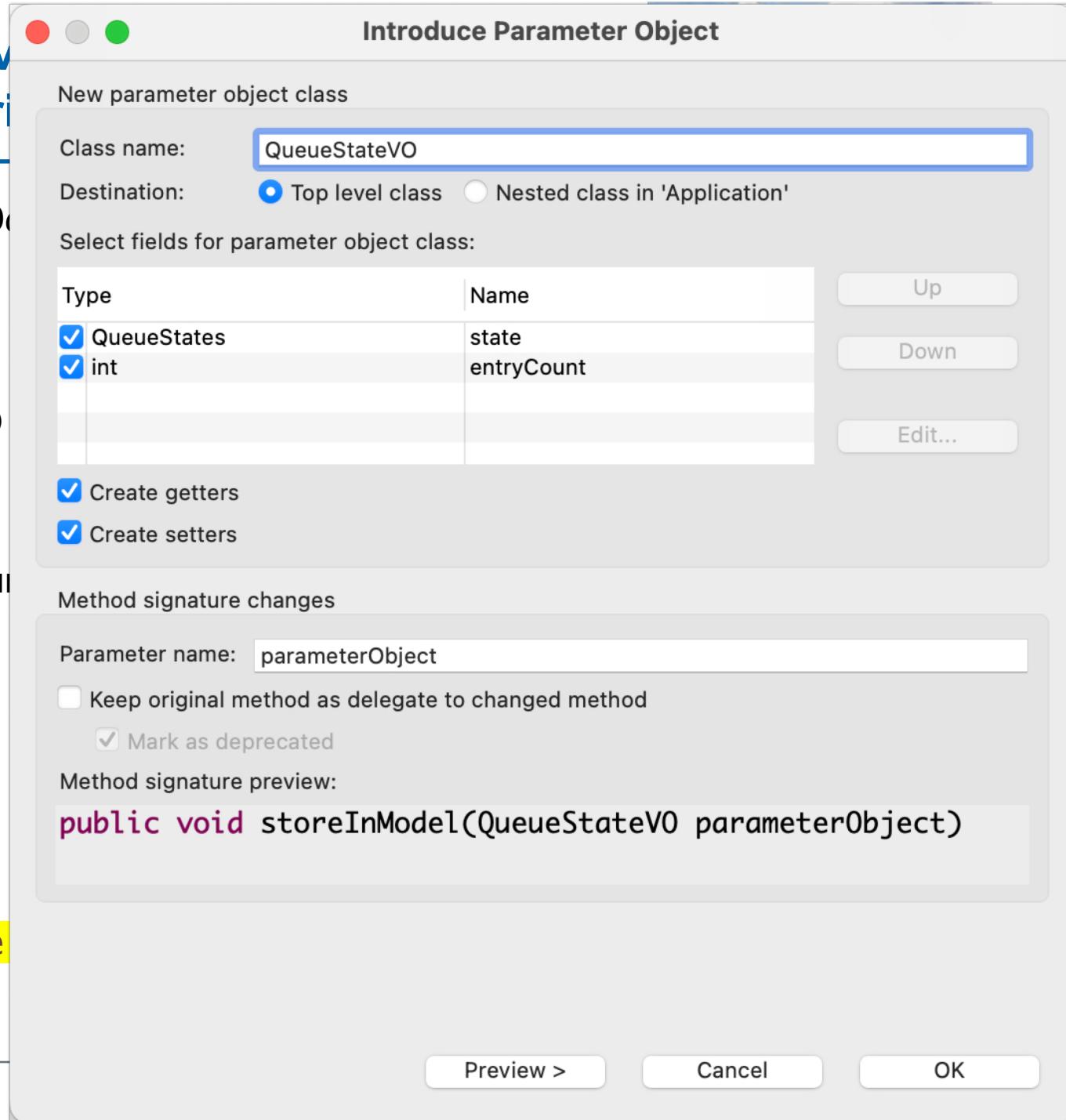
Step 4: Encapsulate information gathering

```
public void updateModelFromElement(final DataObject element) {
    QueueStates state;
    int entryCount;

    if (elementV0 != null) {
        state = elementV0.getQueueStatus();
        if (state == null) {
            state = QueueStates.UNKNOWN;
        }
        entryCount = elementV0.getEntryCount();
    } else {
        state = QueueStates.UNREACHABLE;
        entryCount = 0;
    }

    storeInModel(state, entryCount);
}

public void storeInModel(QueueStates state) {
    model.setState(state);
    model.setQueuedJobs(entryCount);
}
```



Separate information acquisition / retrieval and processing

Step 5: Finetune



```
public class QueueStateV0 {  
    private QueueStates state;  
    private int entryCount;  
  
    public QueueStateV0(QueueStates state, int entryCount) {  
        this.state = state;  
        this.entryCount = entryCount;  
    }  
  
    public QueueStates getState() {  
        return state;  
    }  
  
    public void setState(QueueStates state) {  
        this.state = state;  
    }  
  
    public int getEntryCount() {  
        ...  
    }  
}
```

Separate information acquisition / retrieval and processing

Step 4: Encapsulate information gathering and processing



```
public void updateModelFromElement(final DataElementVO elementVO) {  
    QueueStates state;  
    int entryCount;  
  
    if (elementVO != null) {  
        state = elementVO.getQueueStatus();  
        if (state == null) {  
            state = QueueStates.UNKNOWN;  
        }  
        entryCount = elementVO.getEntryCount();  
    } else {  
        state = QueueStates.UNREACHABLE;  
        entryCount = 0;  
    }  
  
    QueueStateVO parameterObject = new QueueStateVO(state, entryCount);  
    storeInModel(parameterObject);  
}  
  
public void storeInModel(QueueStateVO parameterObject) {  
    model.setState(parameterObject.getState());  
    model.setQueuedJobs(parameterObject.getEntryCount());  
}
```

Separate information acquisition / retrieval and processing

Step 4: Encapsulate information gathering and processing



```
public void updateModelFromElement(final DataElementVO elementVO) {  
    QueueStates state;  
    int entryCount;  
  
    if (elementVO != null) {  
        state = elementVO.getQueueStatus();  
        if (state == null) {  
            state = QueueStates.UNKNOWN;  
        }  
        entryCount = elementVO.getEntryCount();  
    } else {  
        state = QueueStates.UNREACHABLE;  
        entryCount = 0;  
    }  
  
    QueueStateVO parameterObject = new QueueStateVO(state, entryCount);  
    storeInModel(parameterObject);  
}  
  
public void storeInModel(QueueStateVO parameterObject) {  
    model.setState(parameterObject.getState());  
    model.setQueuedJobs(parameterObject.getEntryCount());  
}
```

Separate information acquisition / retrieval and processing

Step 4: Encapsulate information gathering and processing

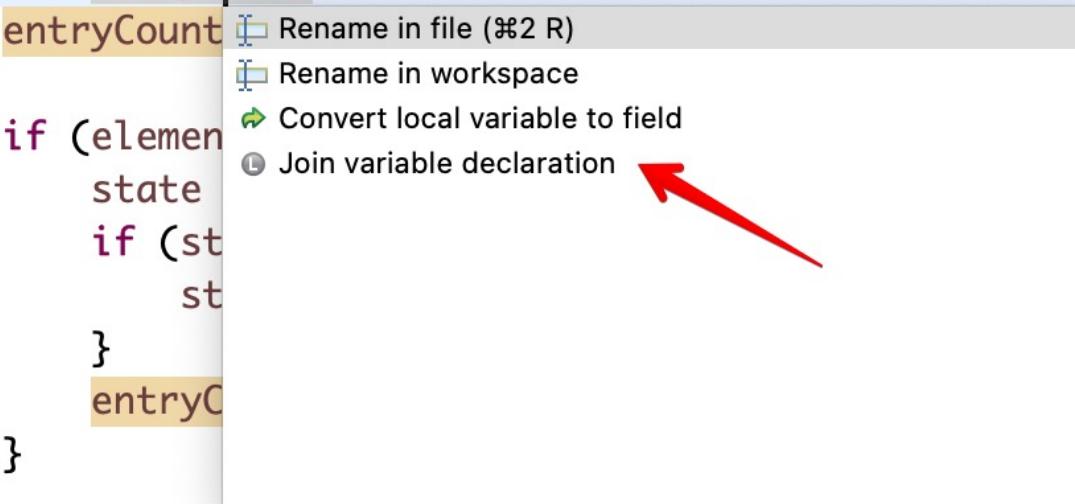
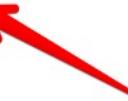


```
public void updateModelFromElement(final DataElementVO elementVO) {  
    QueueStates state;  
    state = QueueStates.UNREACHABLE;  
    int entryCount;  
    entryCount = 0;  
  
    if (elementVO != null) {  
        state = elementVO.getQueueStatus();  
        if (state == null) {  
            state = QueueStates.UNKNOWN;  
        }  
        entryCount = elementVO.getEntryCount();  
    } else {  
    }  
  
    QueueStateVO parameterObject = new QueueStateVO(state, entryCount);  
    storeInModel(parameterObject);  
}
```

Separate information acquisition / retrieval and processing

Step 4: Encapsulate information gathering and processing



```
public void updateModelFromElement(final DataElementVO elementVO) {  
    QueueStates state;  
    state = QueueStates.UNREACHABLE;  
    int entryCount;  
    entryCount   Rename in file (⌘2 R)  
        Rename in workspace  
        Convert local variable to field  
        Join variable declaration  
    if (elementVO != null) {  
        state = QueueStates.REACHABLE;  
        if (state == QueueStates.REACHABLE) {  
            state = QueueStates.REACHABLE;  
        }  
        entryCount = 0;  
    }  
    QueueStateVO parameterObject = new QueueStateVO(state, entryCount);  
    storeInModel(parameterObject);  
}
```

Separate information acquisition / retrieval and processing

Step 4: Encapsulate information gathering and processing



```
public void updateModelFromElement(final DataElementVO elementVO) {  
    QueueStateVO parameterObject = retrieveData(elementVO);  
  
    storeInModel(parameterObject);  
}  
  
public QueueStateVO retrieveData(final DataElementVO elementVO) {  
    QueueStates state = QueueStates.UNREACHABLE;  
    int entryCount = 0;  
  
    if (elementVO != null) {  
        state = elementVO.getQueueStatus();  
        if (state == null) {  
            state = QueueStates.UNKNOWN;  
        }  
        entryCount = elementVO.getEntryCount();  
    }  
    QueueStateVO parameterObject = new QueueStateVO(state, entryCount);  
    return parameterObject;  
}
```

Separate information acquisition / retrieval and processing

Step 4: Finetune I – if you like get() / set() to be adjusted



The screenshot shows an IDE interface with two main components:

- Code Editor:** On the left, there is a code editor window displaying Java code. The code is part of a class and includes methods for retrieving queue state and entry count from a data model. It uses annotations like `@DataModel`, `@QueueStates`, and `@Int`.
- Extract Class Dialog:** A modal dialog titled "Extract Class" is open over the code editor. It contains the following fields:
 - Class name:** QueueStateVO2
 - Destination:** Top level class (radio button selected)
 - Select fields for extracted class:** A table mapping types to names:

Type	Name
<input type="checkbox"/> DataModel	model
<input checked="" type="checkbox"/> QueueStates	state
<input checked="" type="checkbox"/> int	entryCount
 - Create getters and setters:** A checked checkbox.
 - Field name:** data

Separate information acquisition / retrieval and processing

Step 4: Finetune II – move var up and adjust assignments by hand



```
public void updateModelFromElement(final DataElementVO elementVO) {  
    QueueStateVO parameterObject = retrieveData(elementVO);  
  
    storeInModel(parameterObject);  
}  
  
public QueueStateVO retrieveData(final DataElementVO elementVO) {  
    QueueStateVO parameterObject = new QueueStateVO(QueueStates.UNREACHABLE, 0);  
  
    if (elementVO != null) {  
        parameterObject.setState(elementVO.getQueueStatus());  
        if (parameterObject.getState() == null) {  
            parameterObject.setState(QueueStates.UNKNOWN);  
        }  
        parameterObject.setEntryCount(elementVO.getEntryCount());  
    }  
    return parameterObject;  
}  
  
public void storeInModel(QueueStateVO parameterObject) {  
    model.setState(parameterObject.getState());  
    model.setQueuedJobs(parameterObject.getEntryCount());  
}
```

Separate information acquisition / retrieval and processing

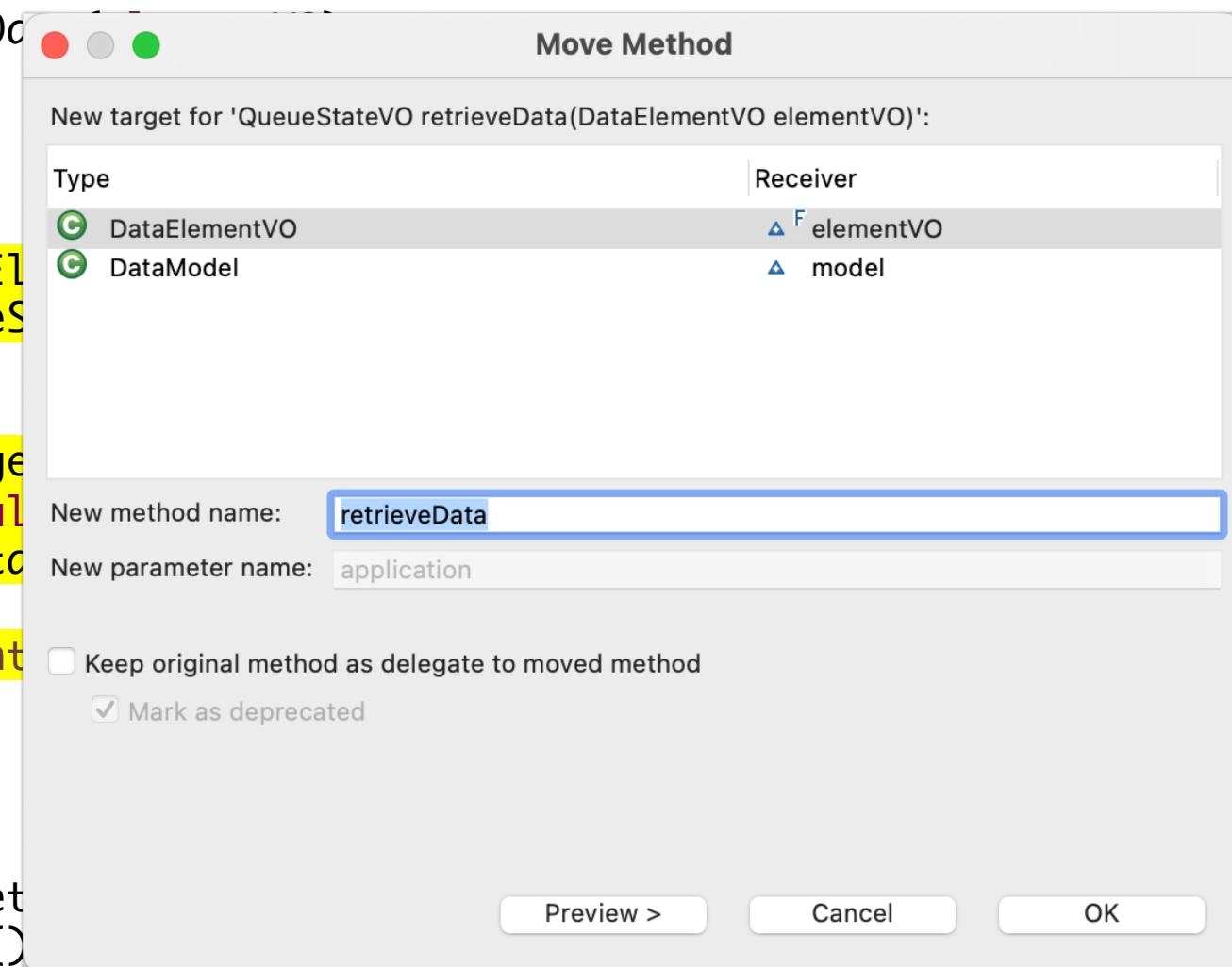
Step 5: Move to QueueStateVO



```
public void updateModelFromElement(final DataElementVO elementVO) {  
    QueueStateVO parameterObject = retrieveData(elementVO);  
    storeInModel(parameterObject);  
}
```

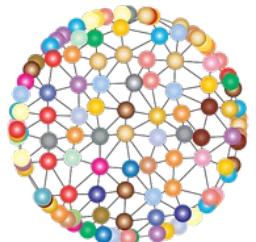
```
public QueueStateVO retrieveData(final DataElementVO elementVO) {  
    QueueStateVO parameterObject = new QueueStateVO();  
  
    if (elementVO != null) {  
        parameterObject.setState(elementVO.getState());  
        if (parameterObject.getState() == null)  
            parameterObject.setState(QueueStateVO.STATE_WAITING);  
    }  
    parameterObject.setEntryCount(elementVO.getEntryCount());  
    return parameterObject;  
}
```

```
public void storeInModel(QueueStateVO parameterObject) {  
    model.setState(parameterObject.getState());  
    model.setQueuedJobs(parameterObject.getEntryCount());  
}
```





But how to move it to
QueueStateVO?



Separate information acquisition / retrieval and processing

Step 5: Move to QueueStateVO



Change Method Signature

Access modifier: public Return type: QueueStateVO Method name: retrieveData

Parameters Exceptions

Type	Name	Default value	Add
DataElementVO	elementVO	-	Edit...
QueueStateVO	tmp	<code>new QueueStateVO(null, 0)</code>	Remove
			Up
			Down

Keep original method as delegate to changed method
 Mark as deprecated

Method signature preview:

```
public QueueStateVO retrieveData(DataElementVO elementVO, QueueStateVO tmp)
```

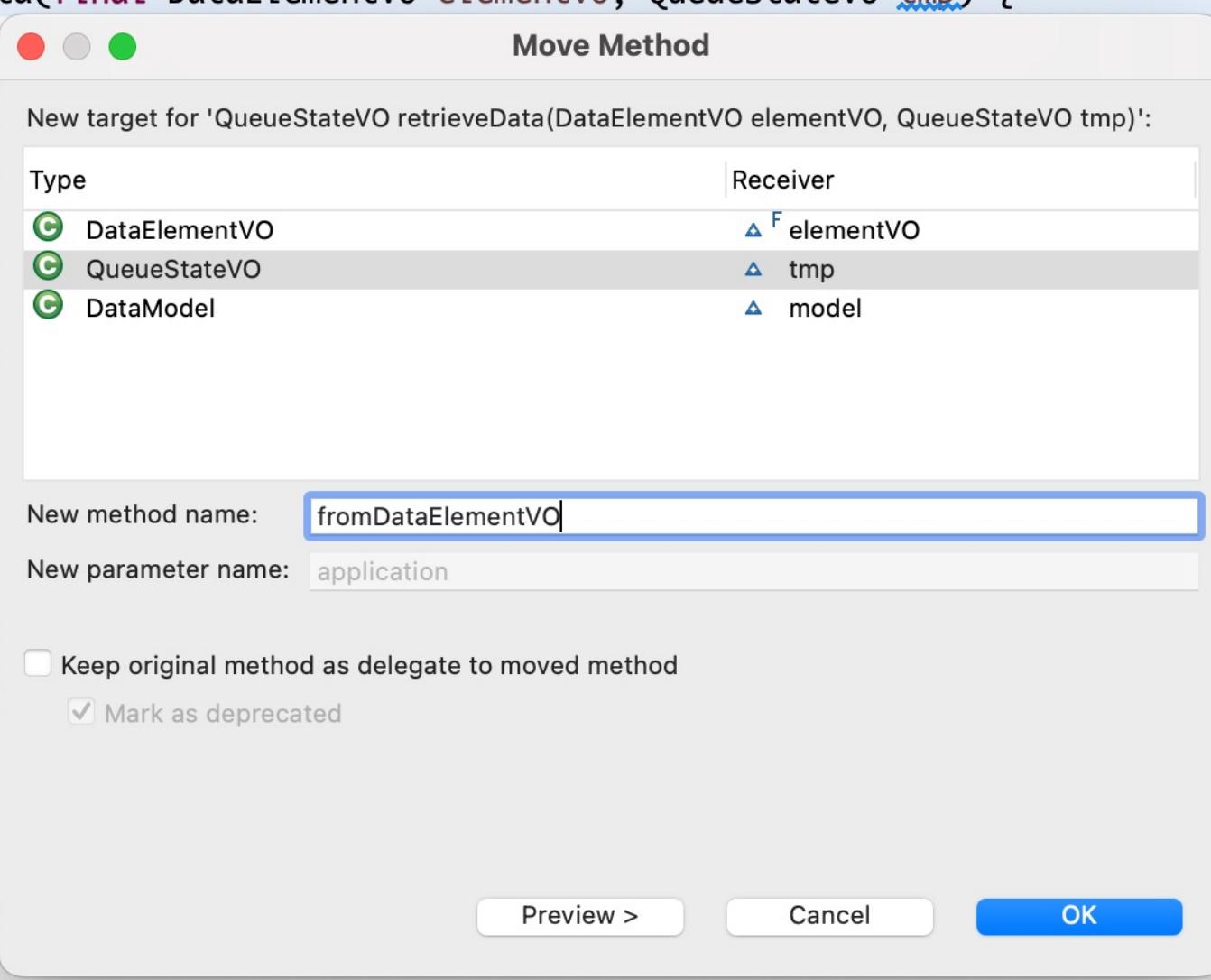
Preview > Cancel OK

Separate information acquisition / retrieval and processing

Step 5: Move to QueueStateVO



```
public QueueStateVO retrieveData(final DataElementVO elementVO, QueueStateVO tmp) {  
    QueueStateVO parameterObject = new QueueStateVO();  
  
    if (elementVO != null) {  
        parameterObject.setState(elementVO.getState());  
        if (parameterObject.getState() == null)  
            parameterObject.setState("N/A");  
        parameterObject.setEntered(true);  
    }  
  
    return parameterObject;  
}  
  
public void storeInModel(QueueStateVO model) {  
    model.setState(parameterObject.getState());  
    model.setQueuedJobs(parameterObject.getJobs());  
}
```



Separate information acquisition / retrieval and processing

Step 5: Move to QueueStateVO



```
public static QueueStateVO fromDataElementVO(final DataElementVO elementVO) {  
    QueueStateVO parameterObject = new QueueStateVO(QueueStates.UNREACHABLE, 0);  
  
    if (elementVO != null) {  
        parameterObject.setState(elementVO.getQueueStatus());  
        if (parameterObject.getState() == null) {  
            parameterObject.setState(QueueStates.UNKNOWN);  
        }  
        parameterObject.setEntryCount(elementVO.getEntryCount());  
    }  
    return parameterObject;  
}
```

Separate information acquisition / retrieval and processing

Step 5: Move to QueueStateVO



```
public void updateModelFromElement(final DataElementVO elementVO) {  
    QueueStateVO parameterObject = QueueStateVO.fromDataElementVO(elementVO);  
  
    storeInModel(parameterObject);  
}
```

```
public void storeInModel(QueueStateVO parameterObject) {  
    model.setState(parameterObject.getState());  
    model.setQueuedJobs(parameterObject.getEntryCount());  
}
```





Resolve if-else / instanceof by polymorphism



Resolve if-else / instanceof by polymorphism



```
for (final Object obj : figures)
{
    if (obj instance Rect)
    {
        ((Rect) obj).drawRect();
    }
    else if (obj instance Line)
    {
        ((Line) obj).drawLine();
    }
    else if (obj instance Circle)
    {
        ((Circle) obj).drawCircle();
    }
}
```

Resolve if-else / instanceof by polymorphism



With a little programming experience it is obvious that this is a strange construct and here is an indication for the use of polymorphism. We transform the source code in the following three steps:

1. interface unification – introduce common method draw()
2. introduce common base type BaseFigure
3. adaptation of the caller

```
for (final Object obj : figures)
{
    if (obj instance Rect)
    {
        ((Rect) obj).drawRect();
    }
    else if (obj instance Line)
    {
        ((Line) obj).drawLine();
    }
    else if (obj instance Circle)
    {
        ((Circle) obj).drawCircle();
    }
}
```

Resolve if-else / instanceof by polymorphism



1. interface unification – introduce common method draw()
2. introduce common base type BaseFigure
3. adaptation of the caller

```
for (final Object obj : figures)
{
    if (obj instance Rect)
    {
        ((Rect) obj).draw();
    }
    else if (obj instance Line)
    {
        ((Line) obj).draw();
    }
    else if (obj instance Circle)
    {
        ((Circle) obj).draw();
    }
}
```

Resolve if-else / instanceof by polymorphism



1. interface unification – introduce common method draw()
2. **introduce common base type BaseFigure**
3. adaptation of the caller

```
public interface BaseFigure
{
    public abstract void draw();
}
```

Resolve if-else / instanceof by polymorphism



1. interface unification – introduce common method draw()
2. introduce common base type BaseFigure
3. **adaptation of the caller**

```
for (final BaseFigure baseFigure : figures)
{
    baseFigure.draw();
}
```



Defensive programming



Defensive programming



1. In (more) complex systems with many components and external calls, it is recommended to program carefully or defensively, especially at the system or component boundaries.
 2. By this is meant that one validates inputs and checks correct initializations and states. (Pre- / Postconditions, Invariants and Design By Contract)
 3. Input parameter checking helps to reject unexpected values. This preserves object integrity and method execution only occurs under safe environmental conditions.
 4. Furthermore, we can make our software more informative in case of errors. As a reaction to invalid parameter values one should raise exceptions, since these can supply a caller with a quantity of information about a possible error cause in contrast to return values.
-

Defensive programming



1. Ensure that methods get called with valid data and fail fast if this is not the case
 2. Please don't overdo it: Checks are not needed and reasonable in every method, this applies foremost to private methods
 3. Why?
 1. To avoid that half of your code consists of sanity checks
 2. Because for private methods you are in charge to ensure validity and integrity of parameters passed

ANALOGY: If you leave your dirty shoes outside your front door, you don't have to keep cleaning the floor of your entire apartment after a walk in the woods.
 4. Just check at the system borders and for public methods which build up the API for your subsystem
- ANALOGY: If you already check IDs and goods at the border, then you don't have to do it again and again throughout the country.



State checks



Defensive programming – Starting point



- **Lot's of sanity checks for the API**
- **A lot of duplicated code**

```
public static final String getExternalNameServiceName()
{
    if (parameterAccessService == null)
        throw new IllegalStateException("ParameterAccessService is not correctly" +
                                         " initialized. Use initialize() before any other method call.");

    return parameterAccessService.getValue("SYSPARAM_EXTERNAL_NAME_SERVICE");
}

public static final String getExternalORBHost()
{
    if (parameterAccessService == null)
        throw new IllegalStateException("ParameterAccessService is not correctly" +
                                         " initialized. Use initialize() before any other method call.");

    return parameterAccessService.getValue("SYSPARAM_EXTERNAL_ORB_HOST");
}

public static final String getInternalORBHost()
{
    if (parameterAccessService == null)
        throw new IllegalStateException("ParameterAccessService is not correctly" +
                                         " initialized. Use initialize() before any other method call.");

    return parameterAccessService.getValue("SYSPARAM_INTERNAL_ORB_HOST");
}
```

Defensive programming



Step 1: Implement / extract validation method(s)

```
private static void checkParameterAccessServiceInitialized()
{
    if (parameterAccessService == null)
        throw new IllegalStateException("ParameterAccessService is not correctly" +
            " initialized. Use initialize() before any other method call.");
}
```



Step 2: Usage of validation method(s)

```
public static final String getExternalNameServiceName()
{
    checkParameterAccessServiceInitialized();
    return parameterAccessService.getValue("SYSPARAM_EXTERNAL_NAME_SERVICE");
}

public static final String getExternalORBHost()
{
    checkParameterAccessServiceInitialized();
    return parameterAccessService.getValue("SYSPARAM_EXTERNAL_ORB_HOST");
}

public static final String getInternalORBHost()
{
    checkParameterAccessServiceInitialized();
    return parameterAccessService.getValue("SYSPARAM_INTERNAL_ORB_HOST");
}
```



Parameter checks



Defensive programming – Starting point



- No parameter checks at all

```
public CommandExecutor(final int minExecutions, final int maxExecutions,  
                      final int registrationStrategy)  
{  
    this.minExecutions = minExecutions;  
    this.maxExecutions = maxExecutions;  
    this.registrationStrategy = registrationStrategy;  
}
```

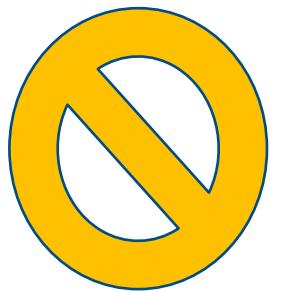
Defensive programming



- Step 1: Adding checks

```
public CommandExecutor(final int minExecutions, final int maxExecutions, final
                      int registrationStrategy)
{
    if (minExecutions < 0 || maxExecutions > MAX_EXECUTIONS)
    {
        throw new IllegalArgumentException("parameter 'minExecutions' or " +
                                           "'maxExecutions' is out of valid range");
    }
    if (minExecutions > maxExecutions)
    {
        throw new IllegalArgumentException("parameter 'minExecutions' must be" +
                                           " <= 'maxExecutions'");
    }
    // Achtung: Hier korrekter, aber potenziell fehlerträchtiger und fragiler
    // Vergleich, weil das Ganze stark vom Mapping und Werten abhängig ist!
    if (registrationStrategy < REPLACE_OLD_OR_ADD_AS_LAST ||
        registrationStrategy > ADD_AS_LAST)
    {
        throw new IllegalArgumentException("parameter 'registrationStrategy'" +
                                           " is invalid");
    }

    this.minExecutions = minExecutions;
    this.maxExecutions = maxExecutions;
    this.registrationStrategy = registrationStrategy;
}
```



Defensive programming



- Step 2: Use of enumerations when useful and possible

```
public CommandExecutor(final int minExecutions, final int maxExecutions,
                      final RegistrationStrategy registrationStrategy)
{
    // ... zwei Prüfungen ausgelassen ...
    Objects.requireNonNull(registrationStrategy,
                         "parameter 'registrationStrategy' must not be null");

    this.minExecutions = minExecutions;
    this.maxExecutions = maxExecutions;
    this.registrationStrategy = registrationStrategy;
}
```



Defensive programming



- **Step 3: Specification of passed values and of validity areas**

```
public CommandExecutor(final int minExecutions, final int maxExecutions,
                      final RegistrationStrategy registrationStrategy)
{
    if (minExecutions < 0 || maxExecutions > MAX_EXECUTIONS)
    {
        throw new IllegalArgumentException("parameter 'minExecutions'=" +
                                           minExecutions + " or 'maxExecutions'=" + maxExecutions +
                                           " is out of valid range: [" + 0 + " - " + MAX_EXECUTIONS + "]");
    }
    if (minExecutions > maxExecutions)
    {
        throw new IllegalArgumentException("parameter 'minExecutions'=" +
                                           minExecutions + " must be <= 'maxExecutions'=" + maxExecutions);
    }

    Objects.requireNonNull(registrationStrategy,
                         "parameter 'registrationStrategy' must not be null");

    this.minExecutions = minExecutions;
    this.maxExecutions = maxExecutions;
    this.registrationStrategy = registrationStrategy;
}
```

Defensive programming



- **Step 4: Extraction of validation / range checker methods**

```
public CommandExecutor(final int minExecutions, final int maxExecutions,
                      final RegistrationStrategy registrationStrategy)
{
    assertExecutionsInValidRange(minExecutions, maxExecutions);
    assertMinExecutionsLessOrEqualToMax(minExecutions, maxExecutions);
    Objects.requireNonNull(registrationStrategy,
                         "parameter 'registrationStrategy' must not be null");

    this.minExecutions = minExecutions;
    this.maxExecutions = maxExecutions;
    this.registrationStrategy = registrationStrategy;
}
```

Defensive programming



- **Step 4: Extraction of validation / range checker methods II**

```
static void assertExecutionsInValidRange(final int minExecutions,
                                         final int maxExecutions)
{
    if (minExecutions < 0 || maxExecutions > MAX_EXECUTIONS)
    {
        throw new IllegalArgumentException("parameter 'minExecutions'=" +
                                           minExecutions + " or 'maxExecutions'=" + maxExecutions +
                                           " is out of valid range: [" + 0 + " - " + MAX_EXECUTIONS + "]");
    }
}

static void assertMinExecutionsLessOrEqualToMax(final int minExecutions,
                                              final int maxExecutions)
{
    if (minExecutions > maxExecutions)
    {
        throw new IllegalArgumentException("parameter 'minExecutions'=" +
                                           minExecutions + " must be <= 'maxExecutions'=" + maxExecutions);
    }
}
```



Exercises

<https://github.com/Michaeli71/ADC BOOTCAMP REFACTORINGS>





Take Aways



Check List Summary



- 1. Run Tests**
 - 2. Apply Coding Conventions**
 - 3. Break down into individual parts**
 - 4. Write unit tests**
 - 5. Clean Up**
 - 6. Simplify unit tests**
 - 7. Perform concrete refactorings unit tests**
-

Take Away: Read vs. Write



We READ 10x more time than we WRITE

→ Make it more readable, even if it's harder to write

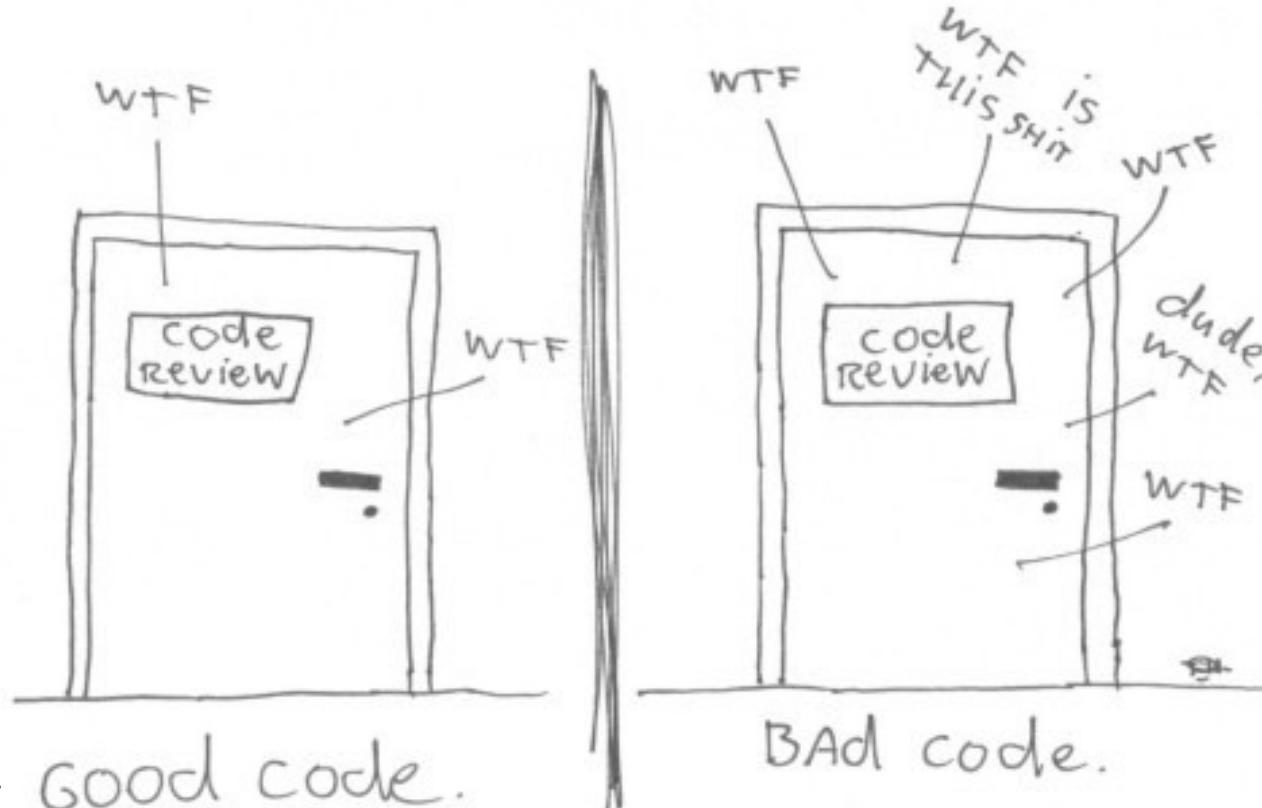
**Take Away: Boycout Rule
Leave the campground cleaner than you found it**



Take Away: WTF – The TRUE measurement



The ONLY VALID measurement
OF code QUALITY: WTFs/minute





Further Info

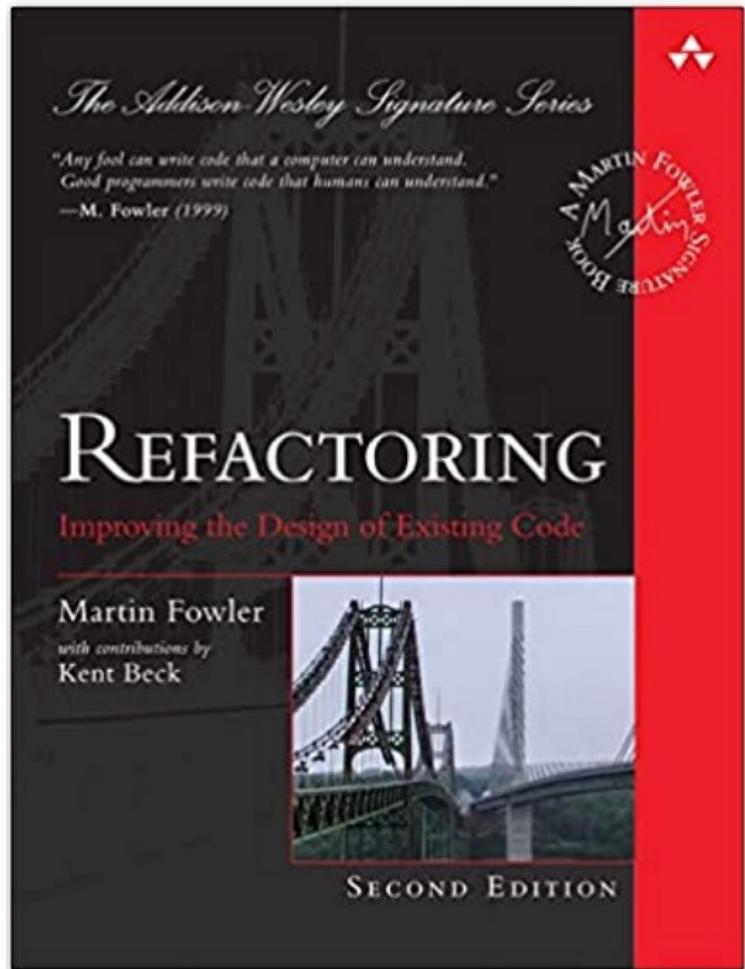


Further Infos I



- <https://www.ibm.com/developerworks/library/os-eclipse-refactoring/index.html>
 - <https://www.jetbrains.com/help/idea/tutorial-introduction-to-refactoring.html#5db90>
 - <https://refactoring.guru/refactoring>
 - <http://www.tutego.de/java/refactoring/catalog/index.html>
 - <https://www.baeldung.com/intellij-refactoring>
 - <https://github.com/lamchau/refactoring-exercise>
-

Further Info II





Thank You