8. Exemplary Solutions: Best Practice Patterns and Refactoring

Exercise 8.1

As a strategy to find occurrences of delegation patterns, you can search for fairly common message names in the O2 classes, message sends such as `#name` or `#browse` are good candidates. As a search tool, Method Finder usually proves useful.

Some examples for each of the three types of delegation follow:

Simple Delegation: Much delegation occurs between the domain model of O2 and the actual domain objects, *e.g.*, from O2ClassNode to the actual class. Also the O2Column class delegates much responsibility to other objects.

- O2ClassNode >> name
- O2Column >> children
- O2Column >> browser
- O2Browser >> announce:
- O2SmartAliasNode >> browse

Self Delegation: Typical usage of self delegation includes passing *self* to #for: or #with: (which might just be part of a longer selector). Thus using the method finder to search for message selectors containing "for:" or "with" and then for senders thereof yields some few results:

- O2FixedButtonPanel >> commands
- O2ColumnPanel >> buildOn:
- O2DefinitionPanel >> buildOn:
- O2Browser >> buildOn:

Double Dispatch: Important superclasses of O2 contain some constructs similar to double-dispatch:

- O2Command >> addItemToMenu: (connecting `O2Command` and `MenuMorph`)
- O2Pane >> model: (connecting `O2Pane` and any model object)
- O2ClassNode >> dropOnSmartGroup: (connecting `O2ClassNode` and `O2SmartGroupNode`)

Exercise 8.2

In the following we report on strategies how to find occurrences of Converting Methods, Lazy Initialization, and Choosing Object:

Converting Methods: Examples are very easy to find by searching, using Method Finder, for selectors starting with 'as'.

- O2MessageNode >> asMethodNode
- O2ClosableDefinitionPanel >> asDraggableMorph
- O2CodeNode >> asFan

Lazy Initialization: 'Accessing' categories of important O2 classes usually contain several examples of Lazy Initialization. Searching for senders of `#ifNil:` helps finding the correct methods:

- O2SmartGroupNode >> nodes
- O2SmartRootNode >> nodes
- O2List >> fan

Choosing Object: This pattern can be easily found by considering large class hierarchies and by looking at the root class of each hierarchy. Methods that appear in the browser with an icon facing south (that is, they are overridden in subclasses) are good candidates to analyze further whether they realize the choosing object pattern.

Many examples are in the O2Node hierarchy:

- O2Node >> definition
- O2Node >> displayString
- O2Node >> name
- O2Node >> text
- O2Node >> shouldBeStyledBy:

Exercise 8.3

Finding bad code smells works similar as detecting patterns. In the following we report on specific strategies.

Law Of Demeter: Violations to the Law of Demeter are quite frequent, often these violations are not very problematic though. But in general, navigational code decreases program understanding and exposes the state of an object to other objects that should not be coupled to these objects, that is, navigational code often breaks encapsulation which makes the system harder to maintain and adapt. Here some examples:

- O2CmdAddPackageOnTop >> execute (#resetCache)
- O2CmdAddSmartCategory >> execute (#addNode:, #refresh)

- O2CmdAddToSmartGroup >> execute (#select:, #refresh)
- O2CmdCreatePackage >> execute (#resetCache)
- O2CmdMethodToExtendClass >> execute (#asString)

Navigational code can be avoided by placing behavior close to data, avoiding accessing data from objects further away from the current object.

Duplicated Code: Duplication often happens in class hierarchies in which classes do a similar job. In the O2Command hierarchy or the O2Browser hierarchy we can find some duplications.

- O2CmdAddPackageOnTop >> execute and O2CmdCreatePackage >> execute
- O2SendersBrowser >> sendersNav: and O2ImplementorsBrowser >> implementorsNav:
- O2ListBrowser >> browseRoot:label: and O2ListBrowser >> browseRoot:titel:

Duplicated code makes systems more complex to understand as it increases their size. Instead of writing a piece of code once, it is written plenty of times. Duplicated code is usually not too difficult to remove, identifying it is harder... To actually remove it we can stick to well-known design patterns such as Template Method.

Long Parameter List: Long parameter lists can be directly seen in the method column of the system browser:

- O2CommandScan >> populateMenu:withNode:forRequestor:filter
- O2ChoiceRequest >> setPrompt:labels:values:lines
- O2CodeBrowser class >> buildMetagraphOn:class:comment:metaclass:

Long parameter lists are usually a sign of missing object oriented modeling. This is an artifact of procedural or functional programming that should not occur in object oriented code. Instead of passing several different objects as single arguments to a method, we could introduce a new type of object that bundles these different objects.

Exercise 8.4

Strategies how to find and examples for the three bad smells Type Tests, God Class, and Feature Envy:

Type Tests: Type tests can be easily found by searching for senders of `#isKindOf:`

- O2DraggableButtonMorph >> wantsDroppedMorph:event:
- O2Announcer >> announce:
- O2CmdBrowseProtocol >> isActive
- O2MultipleDefinitionPanel >> hasMethodPanel
- O2CmdTestClass >> isActive

Type tests are not flexible and not object oriented. The objects themselves should decide what to do depending on their state, remote objects should not reason about the type of other objects, but delegate to them. The easiest way to avoid type tests is to implement testing methods in these classes whether to execute the specific task that has to be executed for a certain type.

God Class: God Classes are usually classes with a wide interface, that is, many methods. Often, but not necessarily, they are at a high position in the class hierarchy.

Some examples of God Classes in O2:

- O2Column
- O2CodeBrowser
- O2Node
- O2ClosableDefinitionPanel

God Classes contain way too much functionality and responsibility. A class should perform one single task, but God Classes usually perform several tasks. Thus they are hard to understand, extend, and maintain. God Classes should be split in several smaller classes, each performing one particular task.

Feature Envy: A good place to look at are the command classes and their `#execute` methods as they often deal with several different objects to perform their respective tasks.

- O2CmdCreatePackage >> execute
- O2CmdImportPackage >> execute
- O2CmdPackageDependencies >> execute
- O2MorphBuilder >> closableDefinitionPanel:with:
- O2MultipleDefinitionPanel >> selectionChanged:

Featue Envies access too much state of other objects, thus they are tightly coupled to these other objects. Adapting the internal implementation of those objects makes it necessary to also adapt the Feature Envy. Thus these distinct objects cannot evolve independently, leading to an overall system which is more difficult to maintain and evolve. If you need too much information of an object, you should delegate the job to the object itself.

Exercise 8.5

Possible fix for the code smell in `O2MorphBuilder >> #closableDefinitionPanel`:

This code smell is an example of a feature envy. Instead of doing the job in O2MorphBuilder, O2DefinitionPanel has to take the responsibility to perform these tasks. A possible solution looks like this:

```
O2MorphBuilder >> closableDefinitionPanel: aDefinitionPanel with: aBlock
   | morph |
   morph := (RectangleMorph new)
       layoutPolicy: TableLayout new;
       listDirection: #topToBottom;
       borderWidth: 0;
       hResizing: #spaceFill;
       vResizing: #spaceFill;
       wrapCentering: #center;
       cellPositioning: #topLeft;
       rubberBandCells: true;
       layoutInset: 2;
       cellInset: 2;
       color: Color white;
       yourself.

   aDefinitionPanel handleMorph: morph with: self.

   current ifNotNil: [current addMorphBack: morph].
   ^ self current: morph do: aBlock

O2DefinitionPanel >> handleMorph: aMorph with: aBuilder
   | contentMorph |
   ((Preferences visualizations and: [self respondsTo: #codeTabPanel])
         and: [self codeTabPanel commands notEmpty])
       ifTrue: [aMorph
         addMorphBack: (self codeTabPanel buildOn: aBuilder)].
   self hasSeveralPanels
       ifTrue: [aMorph addMorphBack: self createTitleBar].
   contentMorph := self createContentMorph.
   self setContentMorph: contentMorph.
   aMorph addMorphBack: contentMorph
```