# Replace Implicit Language with Interpreter

1. Utworzyć pakiet Spec
2. Utworzyć klasę "ColorSpec"

* wystarczy dodać atrybut "Color color"
* w oparciu o niego wygenerować konstruktor i getter("getColorOfProductToFind")

1. Przerabiam finder "byColor" aby używał ColorSpec

* odpalam testy - mają przechodzić

1. Warunek logiczny przenosze za pomocą 'ExtractMethod' do 'isSatisfiedBy'

* Atl+Ctrl+Shift+T (Refactor this...) , 7 (extract method)
* ważne - zaznaczyć tylko treść 'if'a
* zostanie stworzona prywatna metoda

1. Przenieść metodę z "Product Finder" do "Color Spec"

* najechac na metodę
* Atl+Ctrl+Shift+T (Refactor this...) , 5 (move)
* wybrać "ColorSpec"

1. Stworzyć klasę abstrakcyjną "Spec" za pomocą "Extract Superclass"

* wejść do "Color Spec"
* Atl+Ctrl+Shift+T (Refactor this...) , (Extrac Superclass...)

1. Powtórzyć to samo teraz z "BelowPrice"
   1. tworze obiekt BelowPriceSpec - i go generuje
   2. tworze gettera do ceny
   3. zamieniam warunek if by używał Spec'a
   4. reffactor - extract metthod - isSatisfiedBy
   5. reffactor - move method do Spec'a
   6. dodaj extends Spec
2. Mamy '&&' pomiędzy dwoma Spec'ami - należy wprowadzić teraz kolejny Spec jakim jest "AndSpec",

* jak elegancko wydzielać te klasy ??
* AndSpec(Spec first, Spec second) { bool isSatisfiedByProduct(Product) { return getFirst().isSatisfied(product) && getSecond().isSatisfied(product); } }

1. W teście "BelowPriceAvoidingColor" musimy odwrócić ColorSpec(), wprowadzamy "NotSpec"

* NotSpec(Spec spec) { bool isSatisfiedByProduct(Product) { return !getSpec().isSatisfied(product);} }

1. ZADANIE - przerobić pozostałe funkcje tak aby kożystałyze specyfikacji...

* należy rozszerzyć 'AndSpec' by łączył więcej niz jeden warunek And
* należy wprowdzić 'SizeSpec', 'PriceSpec', 'AbovePriceSpec',
* przerobić wszystkie przykłady tak by kod wyszukujący był taki sam dla wszystkich metod

1. Wszystkie metody są takie same i różnią się tylko Specyfikacją, można extraktować metodę

* Atl+Ctrl+Shift+T (Refactor this...) , 7 (extract method)
* selectBy(Spec spec) - InteliJ powinien podpowiedzieć że można zrobić to dla 5 funkcji

1. W zasadzie mógł by być już koniec, ale po co nam są te metody?? - zrefaktorować

* każdą metodę możemy teraz uzunąć używając "Inline method" - powinno nam to uprościć testy, ale także każde użycie w kodzie produkcyjnym !!!
* należy przejżeć testy i sprawdzić czy nie warto neiktórych parametrów Spec'ów też inlinować...

1. Zadanie:

* napisac OrSpec w taki sposób by móc znaleźć produkty: {niebieskie, poniżej 10$} lub {czerwone, małe}
* może najpierw niech napisza sobie na to stosowny test
* może być pokusa napisania funkcji "selectBy(List<Spec>)" - NIE TEDY DROGA !!
* należy wporwadzić "OrSpec"
* szczegóły i rozwinięcie w przykładzie "Replace One/Many DDistinction with Composite"

# Replace Implicit Tree with Composite

* 1. Dziele OrderWritter getContent na trzy funkcje
     + uzywam extractMethod
  2. Wybieram Price jako pierwszy lisc nowego compowsitu
  3. Tworze nowy test TagTest (ctrl+shift+T)
  4. Tworze test 'testSimpleTagWithOneAttributeAndValue'
     + zadaniem testu jest zbudowanie wezla reprezentujacego 'price'
     + wezel ma miec Nazwe/Wartosc/atrybuty
     + trzeba nadpisac toString()
  5. Jak upewnie sie ze wynik mnie zadowala zamieniam w OrderWritter sposób jego budowania
     + nie ruszam testów - maja przechodzic
  6. Tworze test 'testCompositeTagOneChild'
     + zadaniem testu jest zbudowanie wezla 'product' i dodanie do neigo wezla 'price'
     + w wyniku chce otrzymac XML'a '<product><price></price></product>'
     + dodaje funkcje Add do TagNode'a (dodaje do ArrayList<TagNode>())
     + modyfikuje stosownie ToString()
  7. Tworze drugi zagniezdzony test 'testAddingChildrenAndGrandchildren'
     + tworze orders { order { product } }
     + oczekuje rezultatu '<orders><order><product></product></order></orders>'
  8. Tworze trzeci test 'testAddingManyChildren'
     + tworze orders { product, product}
     + oczekuje rezultatu '<order><product></product><product></product></order>'
  9. Czas wrzucic TagNode'a do kodu produkcyjnego
     + przerabiam 'writeProductTo' aby kozystal z TagNode'a
     + metoda 'writePriceTo' teraz musi dodac cene do wezla TagNode, a nie do XML'a
     + nalezy zmienic sygnature metody oraz miejsce dodawania
     + analogicznie z order, orders, itd.

# Permissions

1. Add two new states : UNIX\_REQUESTED, UNIX\_CLAIMED

1. Replace Type Code with Class (points 1,2,3,4,5)

* Introduce getters and setters usage to reach/set "state" field.
* Eclipse : Source -> Refactor -> Encapsulate Field
  1. Encapsulate Field State
  2. Introduce Permission State Class
  3. Create PermissionState instances (public, static, final)
  4. Introduce PermissionState member inside SystemPermision class
* Introduce a method setState(PermissionState state) next to old setState method.

1. Replace Type Code with Class (points 6,7) \*\*\*
   1. Enhance PermissionState class with toString method and contructor that takes "name".
   2. Change getState() method to return permission.toString() value
   3. Get rid of setState(String state) method

1. Replace Type Code with Class (point 8) \*\*\*

* Replace code pertaining to constant String (like REQUIRED) with code refering to PermissionState constants.
  1. Inline String Constants (like SystemPermission.REQUIRED) used in PermissionState5
  2. Change getState() method to return PermissionState instead of String
  3. Replace String Constants used in SystemPermission with (public static final) constants present in PermissionState.

1. Replace State Altering Conditionals with State (points 1,2,3) \*\*\*

1. PermisionState : Extract Subclass 6x : Constructor {super(STATE\_NAME)}
2. Move state altering methods to PermissionState class
   1. Create method claimedBy in PermissionState and move content of the same-method from SystemPermission
   2. Run tests
   3. Create method deniedBy in PermissionState and move content of the same-method from SystemPermission
   4. Run tests
   5. Create method grantedBy in PermissionState and move content of the same-method from SystemPermission
   6. Run tests

1. Replace State Altering Conditionals with State (points 4,5) \*\*\*
   1. Copy all methods that change the state from REQUESTED into any other from base-state-class into PermissionRequested class (this is claimedBy in this case)
   2. Copy all methods that change the state from UNIX\_REQUESTED into any other from base-state-class into UnixPermissionRequested class (this is claimedBy in this case)
   3. Copy all methods that change the state from CLAIMED into any other from base-state-class into PermissionClaimed class (this is deniedBy and grantedBy in this case)
   4. Copy all methods that change the state from GRANTED (no such methods at all)
   5. Copy all methods that change the state from DENIED (no such methods at all)
   6. Copy all methods that change the state from UNIX\_CLAIMED into any other from base-state-class into PermissionClaimed class (this is deniedBy and grantedBy in this case)

# Command

\*\*\* Step 1 : Extract Method(s) \*\*\*

Extract to separate methods the logic between if/else brackets

- addBook

- removeBook

- getAllBooks

- getBookByIsbn

\*\*\* Step 2 : Extract Class(es) \*\*\*

2.1 Self encapsulate fields

Introduce getter for XmlBooksCreator/BooksManager

2.2 Extract classes

- AddBookAction (take it from addBook method)

- DeleteBookAction (take it from removeBook method)

- FindBookByIsbnAction (take it from getBookByIsbn method)

- FindAllBooksAction (take it from getAllBooks method)

Inline methods within CatalogApp that contain only single line of code.

Then some of the methods (addBook, removeBook, getAllBooks, getBookByIsbn) will not exist any longer.

\*\*\* Step 3 : Identify common-method and Extract Super-Class or Interface \*\*\*

3.1 Identify Method that will be common for all command-objects

- public HandlerResponse execute(Map<String, String> parameters)

3.2 Rename the existing methods in different classes to the above one

3.3 Extract Super Class

- extract super-class out of AddBookAction (move CatalogApp variable there)

AbstractAction

- change others as well

- DeleteBookAction extends AbstractAction

- FindBookByIsbnAction extends AbstractAction

- FindAllBooksAction

\*\*\* Step 4 : Change implicit map (ACTION\_NAME -> ACTION) into explicit one \*\*\*

4.1 Introduce a Map that stores that stores all actions.

4.2 CatalogApp now finds given action from the map and run its execute.

# Replace Inheritance with Bridge

The refactoring will be based on introducing SortHandler usage (obtained from SortHandlerFactory) instead of usage of field “S subject” in sorter classes. The refactoring will start from BubbleSorter and once it is fully finished for “bubble sorting” will be done in CombSorter and SelectionSorter.  
Please study the UML diagrams in the presentation.

1. Create interface SortHandler and put it under “handler” subpackage  
   The interface will have the following methods as the abstract ones in BubbleSorter :  
   - getLenght()  
   - compare(i, j)  
   - swap (i, j)
2. BubbleSorter : Change visibility of methods : getLength(), compare() and swap() methods from protected to public and make the same in classes that extend BubbleSorter
3. BubbleSorter : Introduce new field : protected SortHandler<S> handler
4. BubbleSorter : Introduce abstract method “SoftHandler<S> createHandle(S subject” and provide its empty implementations (“return null”) in the classes that inherit from BubbleSorter
5. BubbleSorter : sort-method : Initialize handler field next to initializing subject field : “handler = createHandler(subject)”
6. BubbleSorter : replace 3 abstract methods with implementation that forwards the call to the handler. For example “public void swap(int i,int j) {handler.swap(i, j}”
7. Run all the tests – they must pass in order to continue
8. BubbleSorter : make the implementations of getLenght(), swap(), compare() as final
9. IntArrayBubbleSorter : implement createHandler method. The method can return anonymous implementation of SortHandler interface by moving the 3 other methods (getLength, compare, swap) from the outer class (IntArrayBubbleSorter) into implementation of this inner/anonymous class. As a result IntArrayBubbleSorter will have only single method createHandler.
10. DoubleArrayBubbleSorter : same as above
11. ComparableArrayBubbleSorter : same as above
12. Run all the tests – they must pass in order to continue
13. IntArrayBubbleSorter : extract class IntArraySortHandlerFactory that will contain method createHandler. Now IntArrayBubbleSorter will have a member field “factory” and its createHandler method will do : “createHandler(S subject) {return factory.createHandler(subject);}”
14. DoubleArrayBubbleSorter : same as above with corresponding names respectively
15. ComparableArrayBubbleSorter : same as above with corresponding names respectively
16. BubbleSorter : remove field “S subject” as it is not used any longer. Inline methods getLength, swap, compare
17. Extract common interface “SortHandlerFactory” out of IntArraySortHandlerFactory, DoubleArraySortHandlerFactory, ComparableArraySortHandlerFactory. The interface will have a method “SortHandler<S> createHandler(S subject)”.
18. Move factory field up (to superclass) from IntArrayBubbleSorter, DoubleArrayBubbleSorter, ComparableArrayBubbleSorter. The variable will be set up by calling new BubbleSorter contructor - “BubbleSorter(SortHandlerFactory<S> factory)” from constractors of the 3 above classes.
19. Int/Double/Comparable-ArraySortHandlerFactory : pull up createHandlerFactory into base class. In the base class it will not be abstract, but it will use the factory field instead : “SortHandler<S> createHandler(S subject) {return factory.createHandler(subject}”
20. BubbleSorter : the class is not abstract any longer. Remove abstract modified from definition of this class
21. Int/Double/ComparableArrayBubbleSorter – the classes are not needed (they do not contain any methods now). You can get rid of them by inlining their contructors.