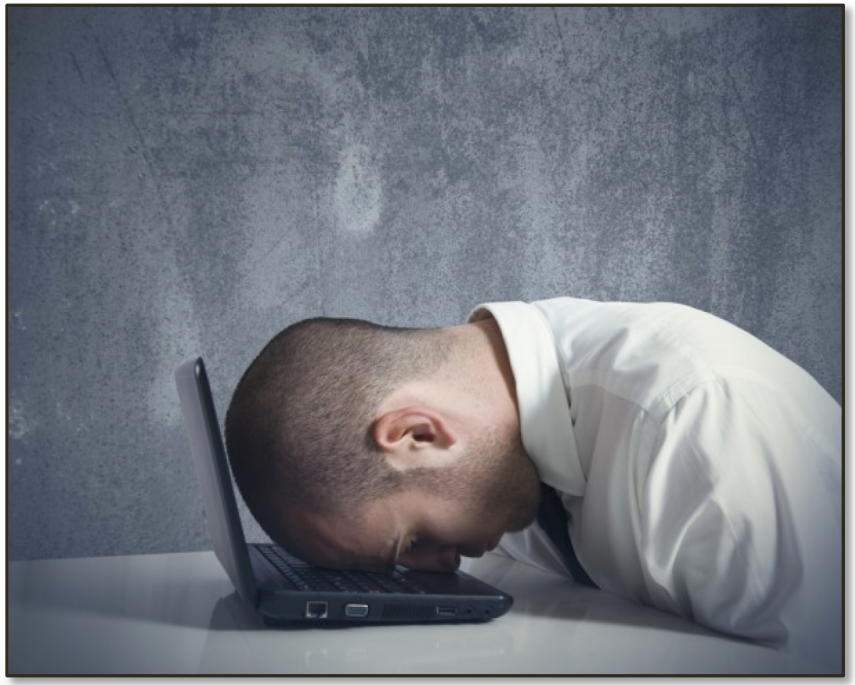
Legacy Code Cipher Exercise

by Mike Rieser and Martin Madathiparambil

Legacy Code can be Challenging!



Sometimes you have a **refactoring target** in mind, and you can refactor into it.
(e.g. State Pattern)

Other times it may be less clear what shape the code should take. When that happens, experiment:

- **apply basic design principles**
- **separate responsibilities**

see what emerges.

Learning Objectives

Work with legacy code in a realistic fashion.

Practice incremental refactoring (green-to-green) – learn techniques to keep code working while refactoring (rather than tearing it apart and trying to reassemble it.)

Apply design principles "bottom-up" to see how a better design emerges without really knowing where it should end up.

SOLID Principle: Single Responsibility (SRP)

A class should have only one reason to change.

Identify responsibilities – consider what causes could force the class to change.

Make the class susceptible to only one cause.

Refactor the other responsibilities into other (singly purposed) classes.

Applying SRP: Separate Creation from Use

- **Object Creation.** One of the most common places that designs depend upon concrete types is where those designs create objects.

```
Map<String, Cipher> ecipherMap = new HashMap<>();
```

- By definition, you cannot create an instance of an abstract class. Thus, where you create an instance, you must depend upon a concrete class.

```
ecipherMap.put(carrierCode, new JBCipher());
```

- **Object Use.** Where you use an object, it is called upon to do its job. How to use the object should be obvious, but not its actual class.

```
ecipherMap.get(carrierCode).encrypt(plainText);
```

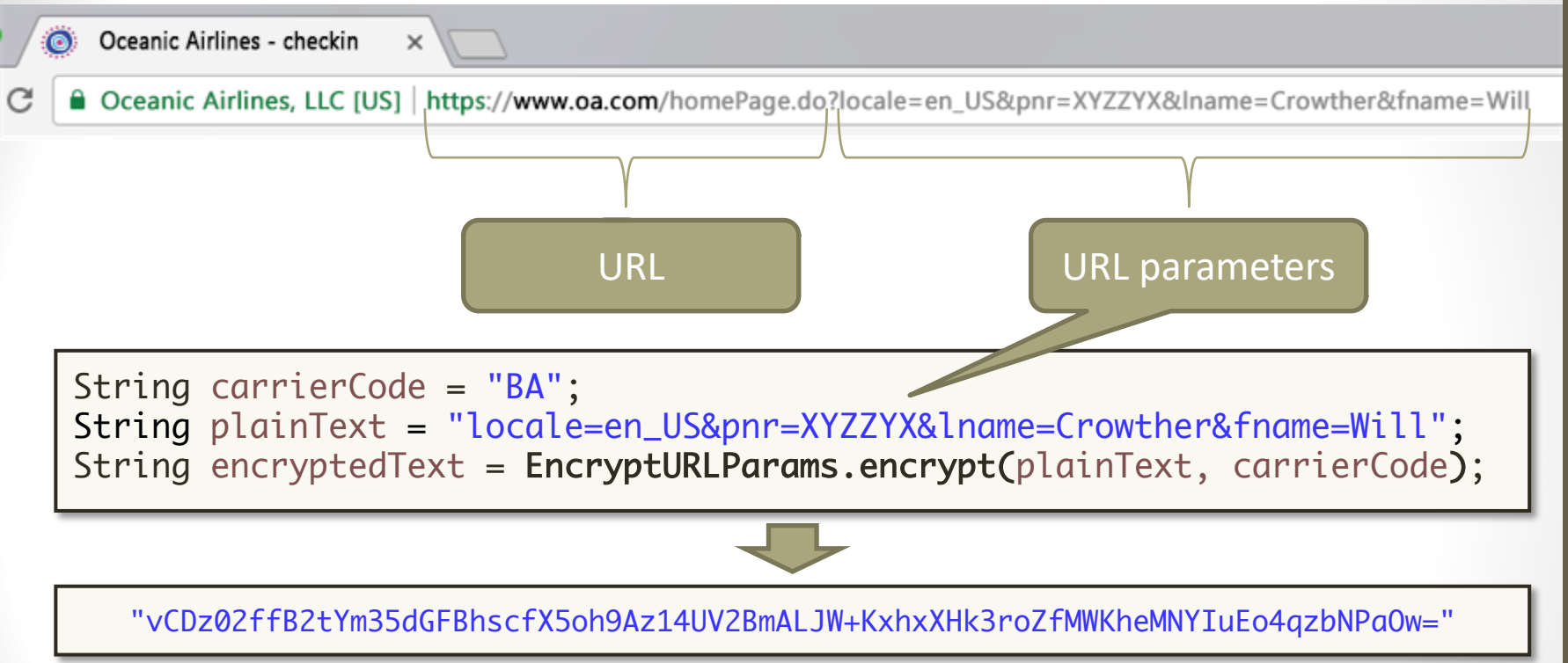
Bartender Rule

Bartender:

It's 3 am! The bar is closed!

You don't have to go home, but you can't stay here!

1. Identify lines of code which are not cohesive with the current class.
2. Extract them and move them to a new class.
3. Repeat as necessary until all classes are cohesive.



Legacy Cipher Problem – EncryptURLParams

Exercise

Refactor the code into a design which makes it easier and safer to add new Joint Business Partners. (2 hours).

- First Refactor **EncryptURLParams**
- Don't change the test cases, leave just the two public static methods.
- Then incorporate **EncryptURLParamsForWMS** and **EncryptDiscountCode** to use **carriercodes** of: "WMS" and "DiscountCode"

Hints:

- Apply **SRP** using the **Bartender Rule** and **Sprout Class** to create new classes.
- **SPOILER**: There is UML Class diagram to use as a refactoring target on the last slide.

Cipher Refactoring (Hints)

- Start with **EncryptURLParams**, identify that the class has multiple responsibilities:
  - **Configuration:** based on ResourceUtil
  - **Creation**: creating Ciphers, CipherMaps, and populating the CipherMaps
  - **Use**: Encrypt & Decrypt
- Lean on the compiler to find usage of ResourceUtil. Isolate the use of ResourceUtil to just one class (SRP)
- Two places to start:
  - Move the configuration to CipherLoader from the static initializer block
  - Move the CipherMaps into a Holder class for the two of them, they're a "data clump"

On Apr 18, 2013, at 12:01 PM, "Madathiparambil, Martin" <Martin.Madathiparambil@aa.com> wrote:

Hi Mike,
Attached the Original classes and test cases.

History of things done

- Started with EncryptURLParams.java . Found that class has multiple responsibilities like configuration, Creating Cipher maps and encrypt/decrypt.
- We created a CipherMapLoader which does the configuration and populate the map in CipherMap.java
- We found that there were two more modules, which does similar functionality(ie Where is my suitcase and Discount Code)
- We looked at the details of how we can generalize the cipher creation part and that ended up in CipherFactory which provides a template method for Cipher Creation
- Since the cipher Algorithm , Key and paramSpec are different in each modules , we Created JBCipher, DiscountCodeCipher and WMS Cipher .

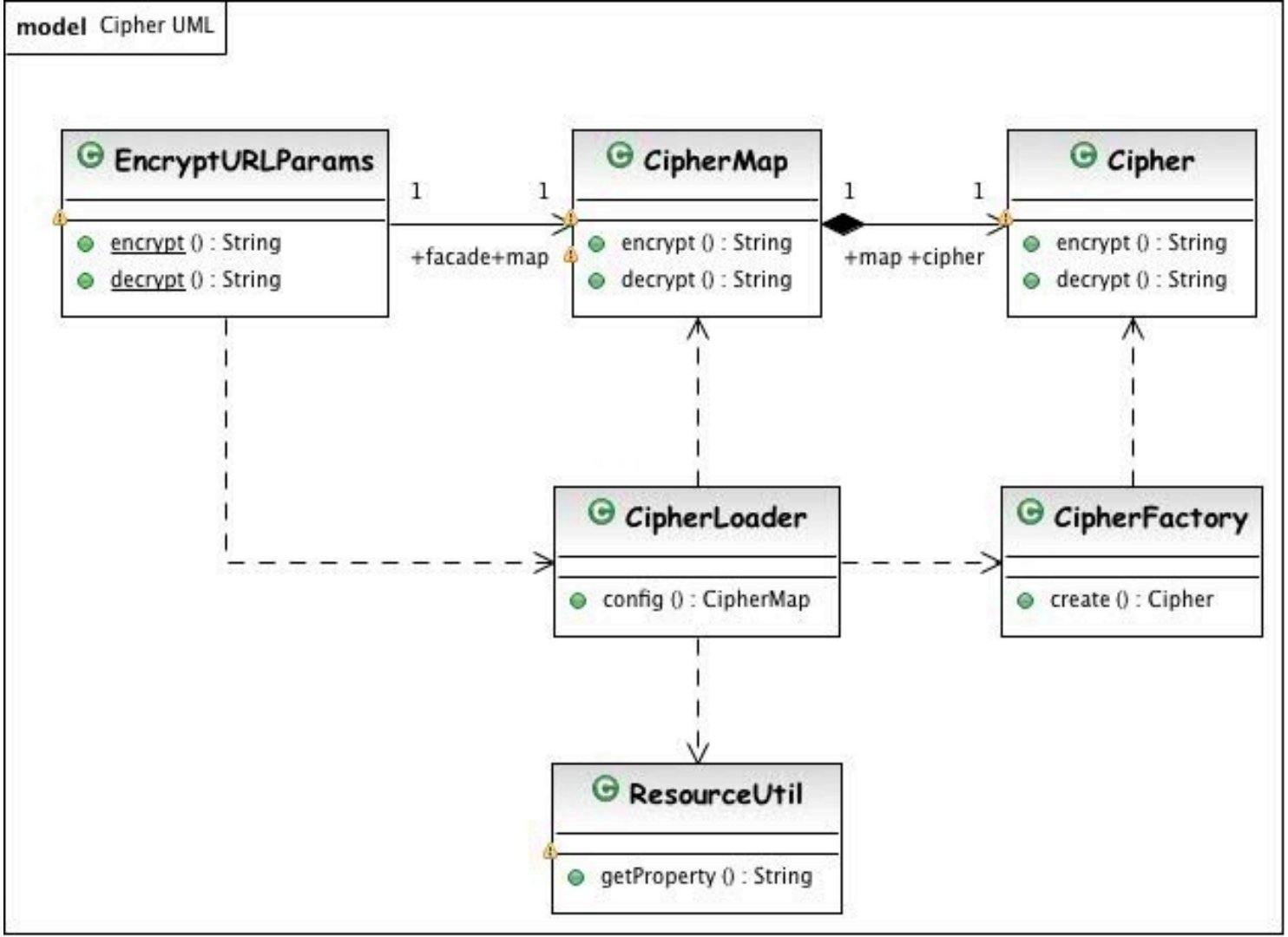
Lessons learned:

- Don't start redesigning the existing class first. Make it simple , apply basic principles like Single Responsibility and Open closed principle, move all common behavior to abstract class and after doing all this
  The classes will automatically falls into a design pattern. What I meant to say that , instead of starting with a design pattern in mind, once you simplify the existing classes it will fall into a better pattern by itself.
  When we started , we had strategy in mind, but later we found that we don't have to use that pattern here.
- Programming to interface is good, but use that when its needed.
- Remove static methods
- Don't change the client call or the interface , instead delegate the call to the refactored code.
- Run always the test cases after doing each change.

Let me know if I can help you in any way.

Thanks,
Martin.

email from Martin


model Cipher UML
EncryptURLParams
encrypt () : String
decrypt () : String
1
1
+facade+map
CipherMap
encrypt () : String
decrypt () : String
1
1
+map +cipher
Cipher
encrypt () : String
decrypt () : String
CipherLoader
config () : CipherMap
CipherFactory
create () : Cipher
ResourceUtil
getProperty () : String

Debrief

- Did that feel like working with real code?
- Did we feel like we need to add test classes for the new classes we've created?
- Does the code now exhibit a GRASP Controller?