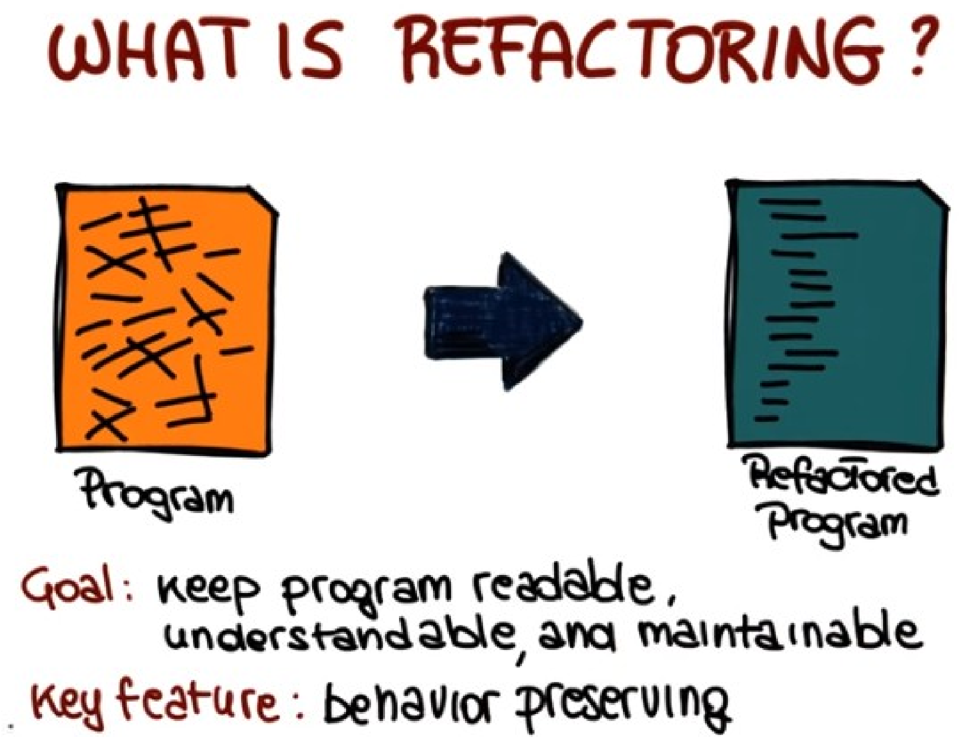
1. In the previous lesson, we discussed agile software development. Its principles and practices. And two specific agile processes. XP and Scrub. In this lesson, we will introduce a practice that is of fundamental importance in the context of agile software development. Software refactoring. Software refactoring is the process of taking a program and transforming it to make it better. That is, to make it easier to understand, make it more maintainable, and in general to improve its design. Specifically, we will discuss what the refactoring is and why it is important? When to perform refactoring, and when not to perform refactoring? And how to perform refactoring in a fully automated way, using tools. We will also introduce the concept of bad smells, where a bad smell, in software, is the indication that there might be something wrong with the code that might call for the application of refactoring.

我的理解: refactor 就想當於是一個美化工作, 類似於一篇文章寫好了後, polish 它的措辭等

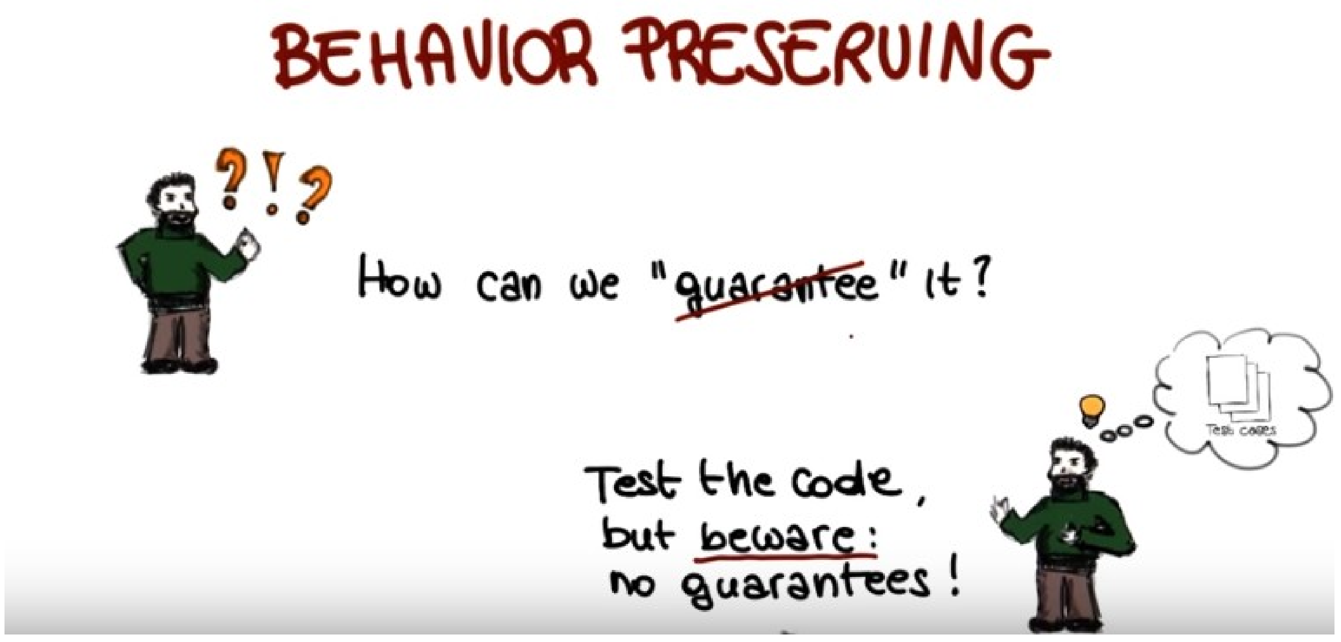
2. Let me start this lesson by discussing what is refactoring. Refactoring is the process of applying transformation or refactoring to a program. So as to obtain a refactor program. With an improved design but with same functionality as the original program. So key aspect of refactoring is the fact that refactoring should be somatic to perserving. So what is the main goal of refactoring? The goal is to keep the program readable, understandable, and maintainable as we evolve it. And to do this by eliminating small problems soon, so that you can avoid big trouble later. And I want to stress once more a key feature of refactoring, which is the fact that it is behavior per serving. But how can we ensure that the refactoring is behavior preserving? In other words, how can we ensure that the program does the same thing before and after applying a refactoring. So what we would like to do is to have some guarantee that, that happens. And unfortunately in general, there are no guarantees. But something we can do is to test the code. For example, we can write tests that exercise the parts of the program affected by the refactoring, and if we're in a [INAUDIBLE] context, we might already have plenty of test cases that exercise that part of the code. So we might just have to rerun the test cases after the refactoring. And in fact, that's a very advantageous situation, and that's a very good use of existing test cases. And I want to make sure that you remember, and that you beware that tests provide no guarantees. Testing can only show the presence of defects, but cannot demonstrate their absence. So we can use testing to get confidence in our refactorings, but we can't really guarantee that the refactorings are behavior preserving. I'd also like to point out that for some simple refactoring, we can use a static analysis to actually provide these guarantees. And in fact we will see examples of such refactorings that are incorporated into IDs and that leverage these kinds of analysis to perform refactoring in a safe way.



WHAT IS REFACTORING?

Goal: keep program readable, understandable, and maintainable

Key feature: behavior preserving

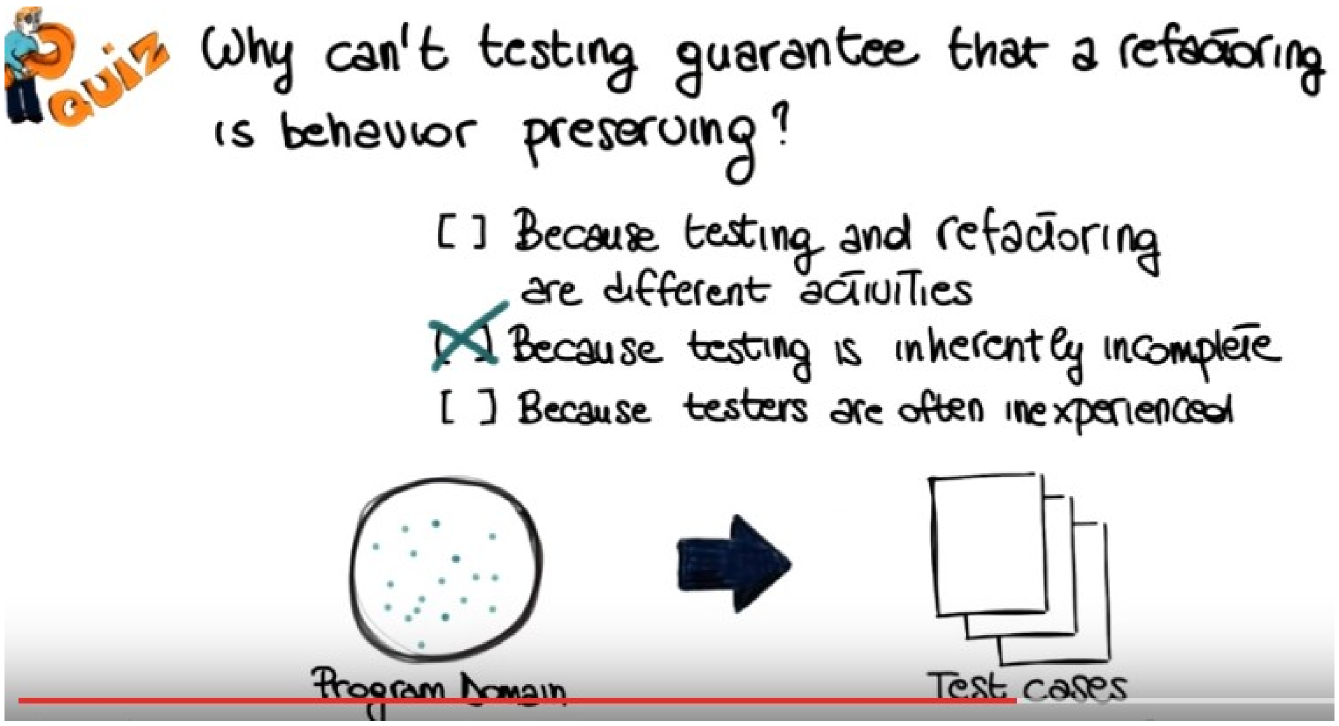


BEHAVIOR PRESERVING

How can we "guarantee" it?

Test the code, but beware: no guarantees!

3. So let's have a small quiz and see whether you can remember why can testing guarantee that that a refactoring is behavior is preserving. So why testing can only show the absence of defects and not their presence? Is that because testing and refactoring are different activities? Because testing is inherently incomplete? Or, just because testers are often inexperienced? Make your choice.



Why can't testing guarantee that a refactoring is behavior preserving?

[ ] Because testing and refactoring are different activities

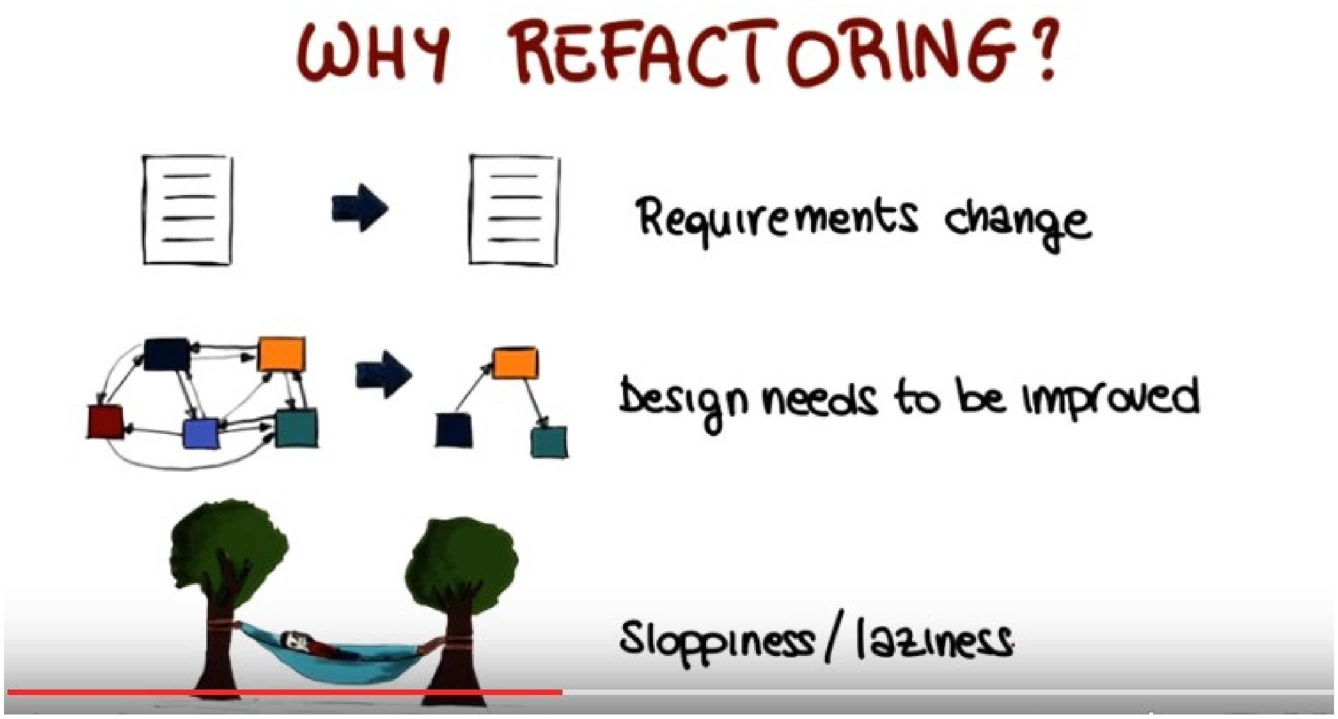
[✓] Because testing is inherently incomplete

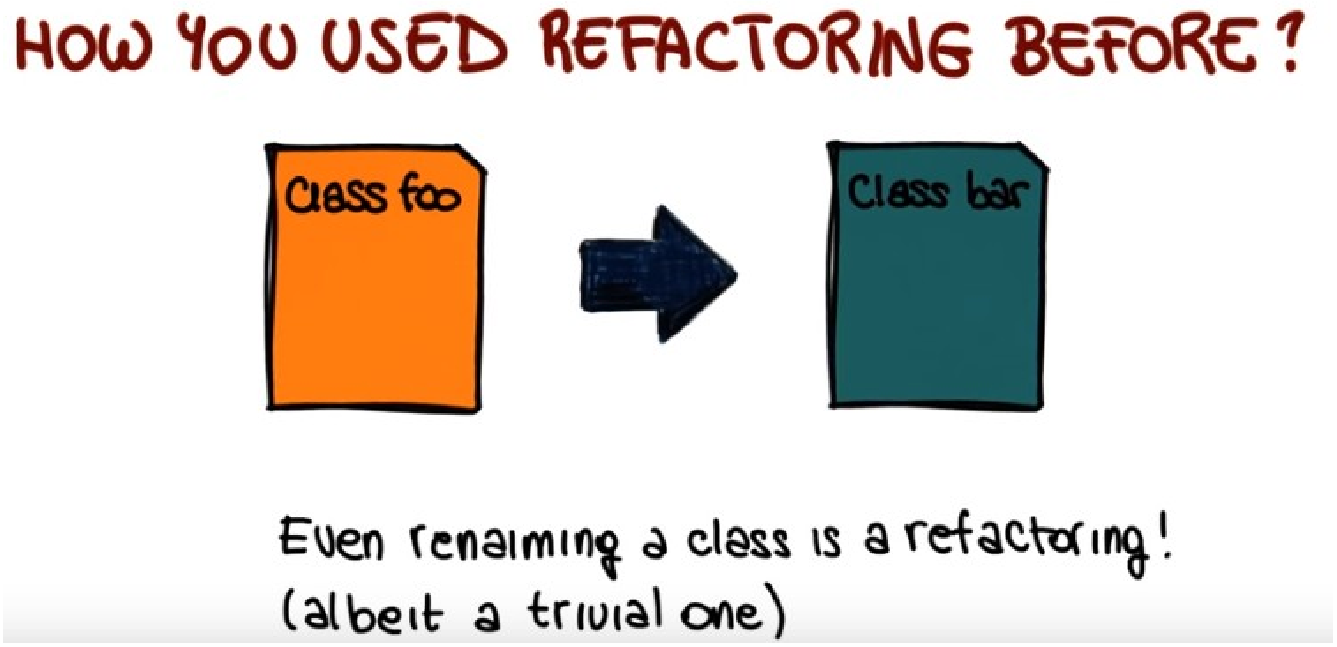
[ ] Because testers are often inexperienced

4. And the reason for this is because testing is inherently incomplete. So let me re-size this a little bit, so that I can make room for an illustration. And what I'm going to show you here is just a reminder, that when we test, we have a huge virtually infinite input domain, and we have to derive from this input domain a few test cases. By picking specific inputs in the domain, and of course, corresponding outputs. And so what happens normally is that these test cases represent the teeny tiny fraction of the domain, and therefore testing is always incomplete.

5. We saw at the beginning of the lesson, what are the goals of refactoring? Or what are the reasons ,why we need to refactor in the first place? The first reason is that requirements change, and when the requirements change, we often need to change our design accordingly. In other cases if any of the requirements unchange, we might need to improve our design. And this happens for many reasons. For example, we need to add a new feature, we want to make the code more maintainable, and also in general programmers don't come up with the best design the first time. So they might need to adapt it after the fact. And the final reason I want to mention is sloppiness, and to some extent laziness, of programmers. And a typical example of this is something that we all have done, which is copy and paste programming. So instead of rewriting a new piece of code, because we know that there is some code in some other parts for the program that does a similar thing, we'll just copy the code over. And before we know, we end up with tons of copies of the same functionality. And when that happens, a good way of consolidating(鞏固,使聯合, 合併) that code and extracting that functionality is to use refactoring, for example, by creating a method or a class that provides the functionality. And we'll see specific examples of that. A question I would like to ask at this point of the class is whether you have used refactoring before? So I want you to take a second and think about it. And no matter what you're history is, if you ever coded I bet you any money that the answer is yes, you have done refactoring. What do I mean? I'm going to give you an example. I'm sure you renamed the class or a method or

change the name of some variables in the code before. That's refactoring. Even something as simple as renaming a class is refactoring, because, for example, it might help you making your code more understandable. And of course I'll admit that in this case, this is a trivial refactoring, and there are much more interesting ones.

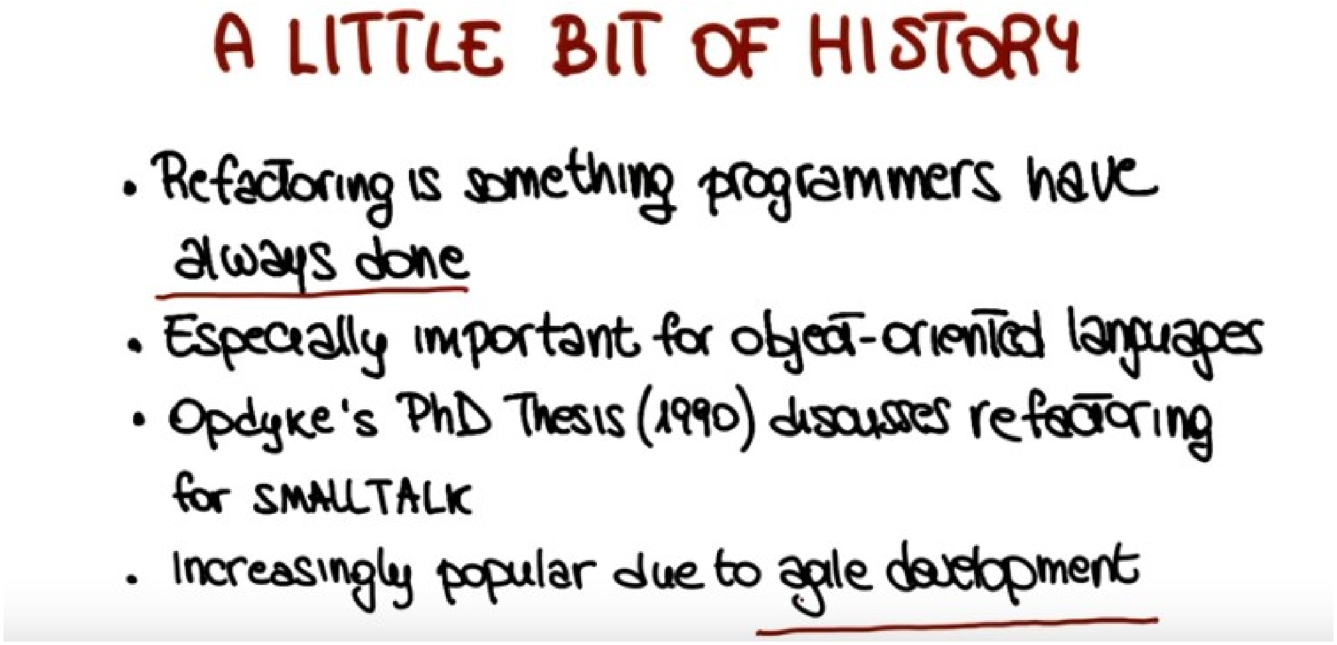



HOW YOU USED REFACTORING BEFORE?
Class foo
Class bar
Even renaming a class is a refactoring!
(albeit a trivial one)

(上圖第一行的 How 應該是 Have)

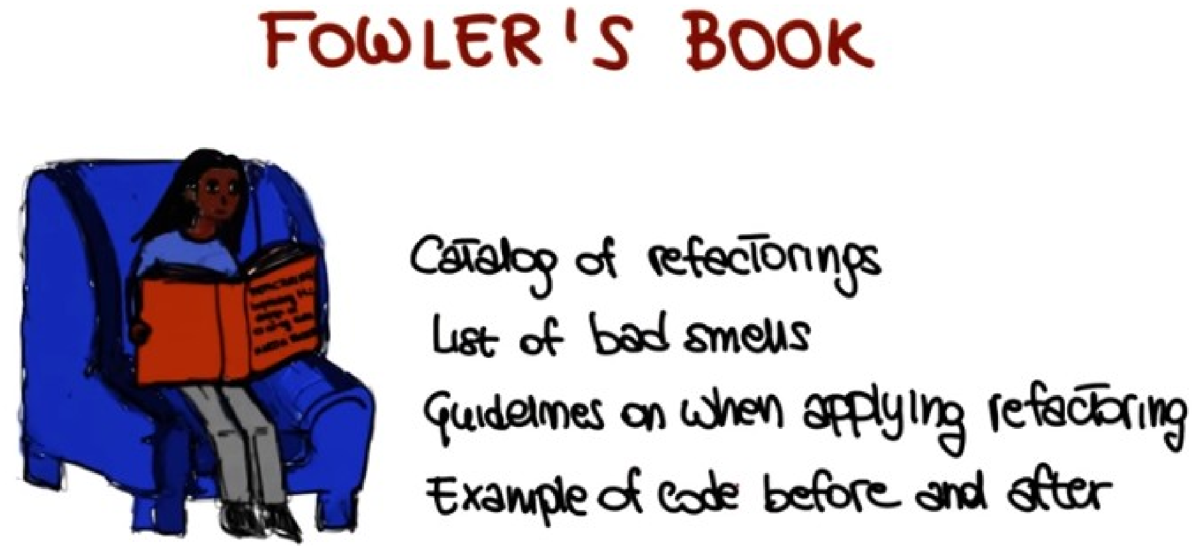
6. So if you follow my class so far, you know that I like to give a little bit of history when I talk about a specific topic. So I'm going to do the same also in this case for refactoring. I'm going to start by mentioning, the fact that refactoring is something that programmers have always done. I gave you a

trivial example just a minute ago of what refactoring is. So even more complicated refactorings are something that are commonplace for developers. Somehow refactoring is especially important in the context of object-oriented languages and probably it's because the object-oriented features are well suited to make designs flexible and reusable. Because of the fact that help encapsulation, information hiding, and so they make it easier to modify something without changing the functionality that it provides to the outside world. However, you should keep in mind that refactoring is really not specific to object oriented languages, you can also refactor other languages, it's just more common to see it in that context. So one of the first examples of a specific discussion of what the refactorings are is Opdyke's PhD thesis in 1990. Which discusses refactorings for small talk. And some of you might be familiar with small talk, which is a specific objectory language. And in more recent times, refactoring's becoming increasing popular due to lightweight development methodoogies, due to agile development, which is something that we just discussed in this class. For example, when we talked about extreme programming, we mentioned refactoring a few times. And the reason why its so popular is because re-factoring is one of the practices that help. Making changes less expensive. And therefore adapt to changing requirements and changing environments more quickly. And continuing with historical perspective, one of the milestones in the history of re-factoring [INAUDIBLE] is a book by Martin Fowler. This is a book entitled Improving the Design of Existing [INAUDIBLE]. And it contains a catalog of refactorings, a list of bad smells, in code, and we're going to see what that mean exactly. Nothing to do with other kinds of bad smells. It talks about guidelines on when to apply refactoring. And finally, which is very useful, it provides example of code, before and after. Applying the refactoring and we're going to use more of the same style when discussing refactoring in the rest of this lesson. More specifically what we're discussing next, are some examples of refactoring and also some examples of code bad smells.



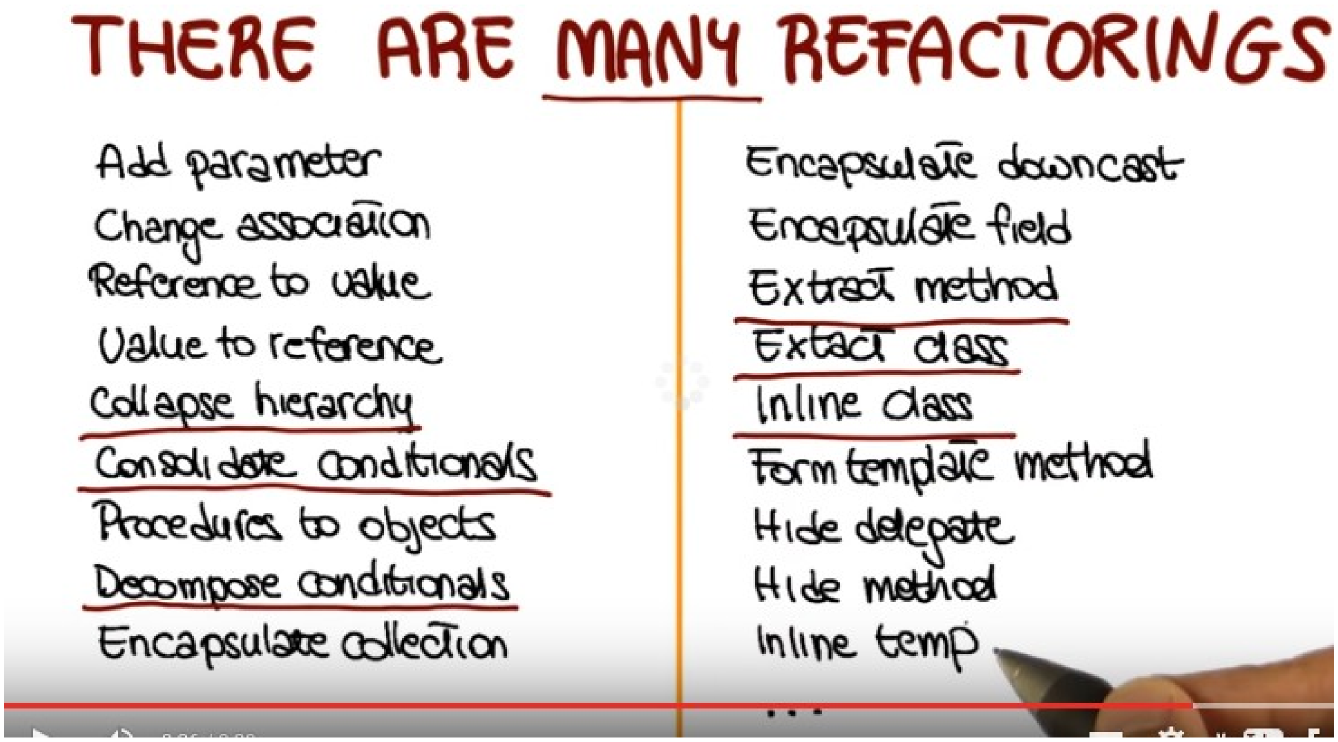
A LITTLE BIT OF HISTORY

- Refactoring is something programmers have always done
- Especially important for object-oriented languages
- Opdyke's PhD Thesis (1990) discusses refactoring for SMALLTALK
- Increasingly popular due to agile development


FOWLER'S BOOK
Catalog of refactorings
List of bad smells
Guidelines on when applying refactoring
Example of code before and after

COMING UP NEXT
CLASS A
CLASS B
CLASS C
Some examples of refactoring
CLASS A
Some examples of "bad smells" in code

7. There are many refactorings in Fowler's book, and what I'm showing here is just a partial list. And we're not going to have time to go through the complete list of refactorings, so what I'm going to do instead, I'm just going to pick a few of those, but I'm going to explain in more depth, and for which I'm going to provide some examples. In particular, we're going to talk about the collapse hierarchy refactoring, the consolidate conditional expressions, the decompose conditionals, extract method, extract class, and inline class. And we're going to see each of those individually in the rest of the lesson.

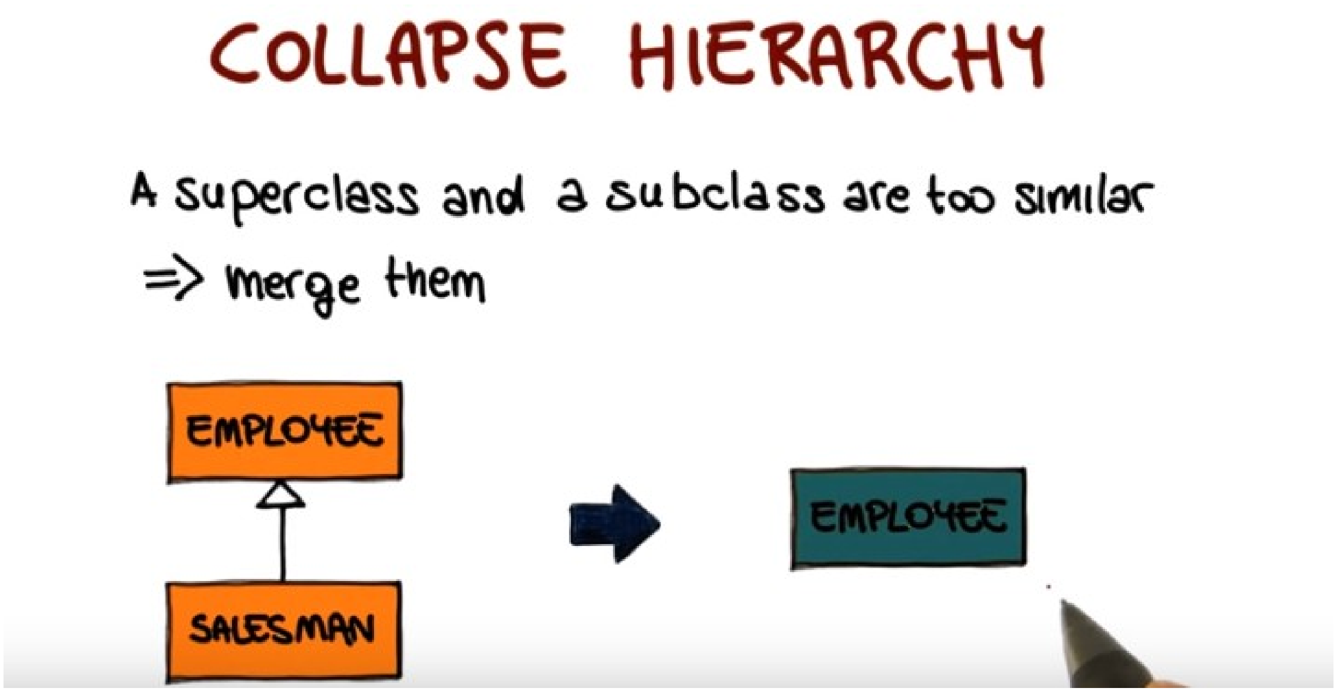


THERE ARE MANY REFACTORINGS

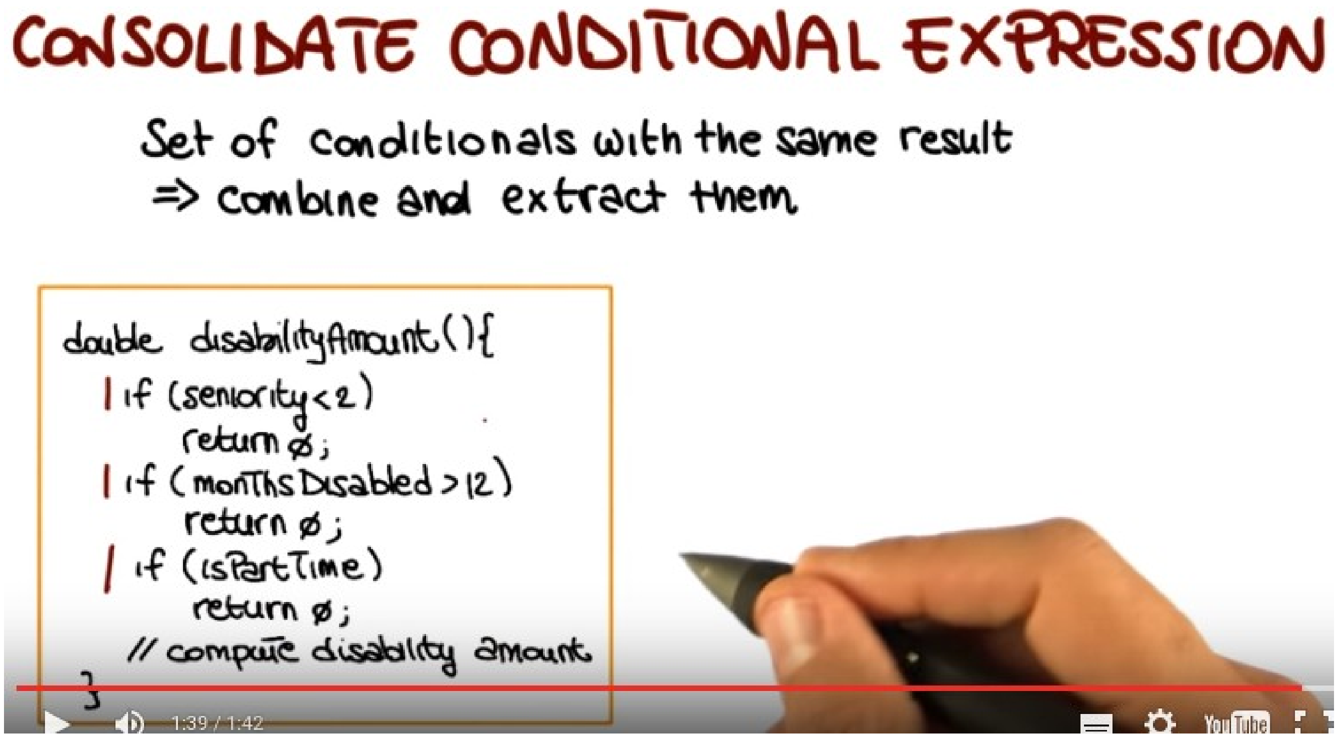
Add parameter
Change association
Reference to value
Value to reference
Collapse hierarchy
Consolidate conditionals
Procedures to objects
Decompose conditionals
Encapsulate collection

Encapsulate downcast
Encapsulate field
Extract method
Extact class
Inline class
Form template method
Hide delegate
Hide method
Inline temp
...

8. The first refactoring we're going to see is the collapse hierarchy refactoring. When a software system undergoes a number of changes, over time the collapse hierarchy may become, let's say, sub-optimal. There are several refactorings that address this issue for example, refactorings that allow you to move methods and fields up and down the class hierarchy. So what happens when you apply a number of these refactorings, is that a subclass might become too similar to its superclass and might not be adding much value to the system. In this case, it is a good idea to merge the classes together. That's exactly what the Collapse Hierarchy refactoring does. Imagine, for instance, that we have two classes: employee and salesman. And that salesman is just so similar to employee that it does not make sense to keep them separated. In this case, you could merge the two classes, so that at the end of the refactoring, only employee is left. And the resulting structure of the system is improved.



9. We're now going to talk about the consolidate conditional expression refactoring. A common situation in code is that you have a set of conditionals with the same result. What that means that sometimes the code contains a series of conditional checks in which each check is different, yet the resulting action is the same. In these cases, the code could be improved by combining the conditionals using, for example, and, and or, as connectors. So as to have a single conditional check, with a single result. At that point you can also extract those conditional into a method. And replace the conditional with a call, to debt matter consolidating the conditional code in this way can make the checks clearer by showing that you're really making a single check rather than multiple checks, and extracted that condition and having that matter instead of a condition can clarify your code by explaining why you're doing a given check, rather than how you're doing it. You can see an example of that situation in this code, which is the disabilityAmount method. As the name of the method says, the purpose of this code is to compute the disability amount for a given, for example, employee. And there is a set of initial checks in the methods whose goal is to decide whether there's this disabilityAmount should be instead zero. And as you can see, there's multiple conditions. For example, there's a check about the seniority level, and about the number of months that the employee's been disabled. So far, whether the employee is part time and the outcome of all these check is always the same. If they're true, if the check is satisfied then there is no disability amount. So the disabilityAmount is zero. So what I will do if I apply the consolidate conditional expression to this matter, is that I will take these three conditionals. I will put them together by saying basically that if seniority is less than 2 or monthsDisabled is greater than 12 or isPartTime is true then the return should be zero. And once I have this combined conditional, as I see here, I will just extract that into a method.

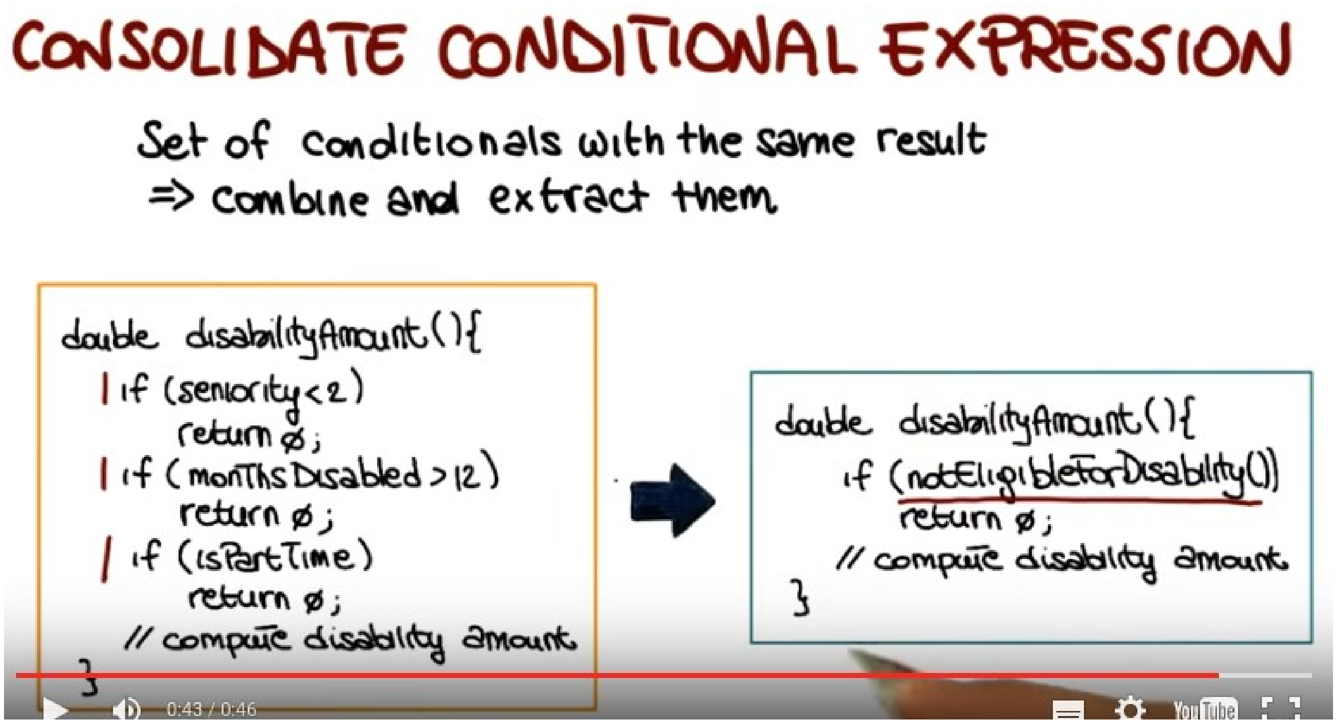


CONSOLIDATE CONDITIONAL EXPRESSION

Set of conditionals with the same result
=> combine and extract them

```
double disabilityAmount(){
  if (seniority<2)
      return 0;
  if (monthsDisabled >12)
      return 0;
  if (isPartTime)
      return 0;
  // compute disability amount
}
```

10. So the resulting code will be like this. As you can see here, now I don't have the conditionals any longer, but I just have a call to this notEligibleForDisability method. And this makes the code so much clearer, because if I just need to understand how disabilityAmount works, I can clearly see there is an initial check that is actually checking whether an employee's eligible for disabilities or not. And if the check is false, so if the employee's eligible, then I'll just perform the rest of the computation. Otherwise I'll simply return zero. So if I don't need to understand the details of this check, I can simply look at this method and understand it all. And if I need to look at the details I can just go, and look at the implementation of this method, and I will get exactly the same information that I have here. But I'm sort of separating the concerns, and making the code overall more understandable, and therefore more maintainable, which is the main goal of refactoring.



CONSOLIDATE CONDITIONAL EXPRESSION

Set of conditionals with the same result
=> combine and extract them

```
double disabilityAmount(){
  | if (seniority<2)
        return ø;
  | if (monthsDisabled>12)
        return ø;
  | if (isPartTime)
        return ø;
   // compute disablity amount
}
```

```
double disabilityAmount(){
    if (notEligibleForDisablity())
        return ø;
    // compute disablity amount
}
```

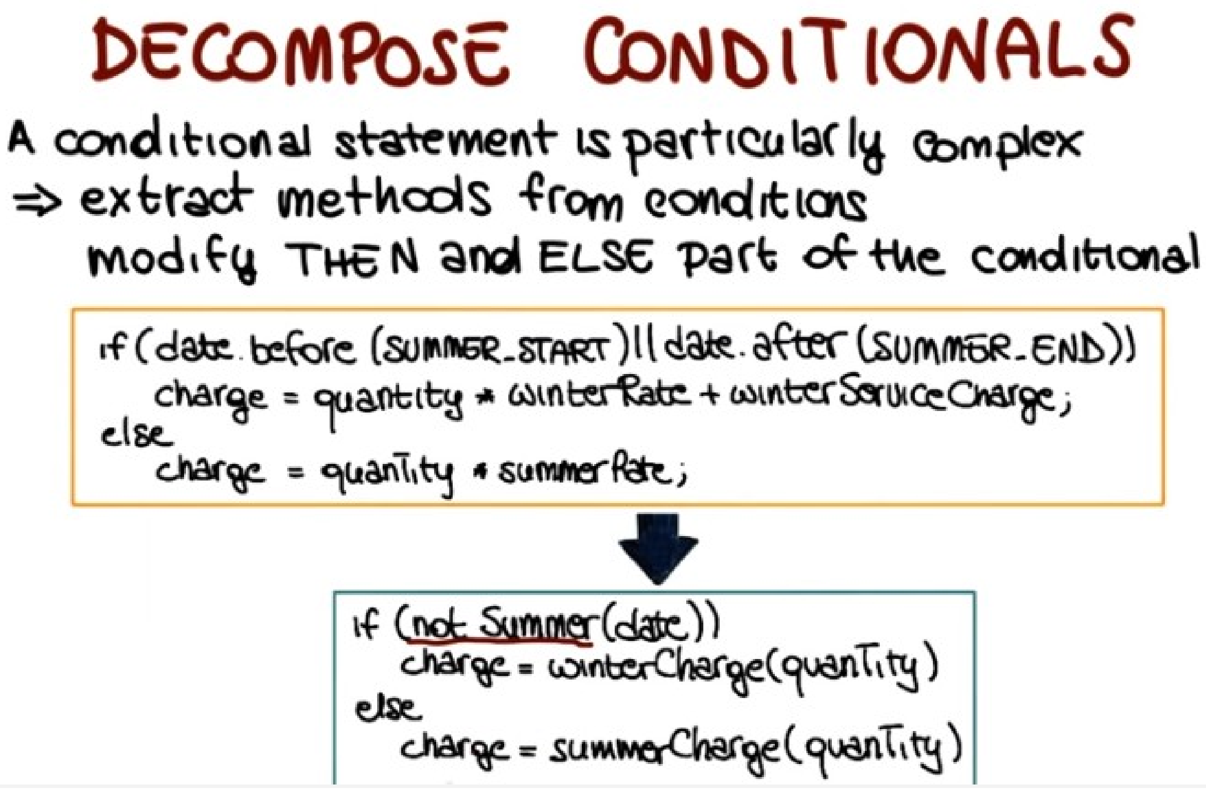
```
public class Employee {
	private boolean isPartTime;
	private int seniority;
	private int monthsDisabled;

	public Employee(boolean isPartTime, int seniority, int monthsDisabled) {
		this.isPartTime = isPartTime;
		this.seniority = seniority;
		this.monthsDisabled = monthsDisabled;
	}

	// TODO: add a function here with the signature: public boolean notEligibleForDisability()

	public double disabilityAmount() {
		// TODO: replace these checks with a single function call
		if (seniority < 2)
			return 0;
		if (monthsDisabled > 12)
			return 0;
		if (isPartTime)
			return 0;

		// in a real program, there would be other logic for determining
		// the amount of disability the employee is eligible for here
		// the following is a dummy return value to make the compiler happy
		// and differentiate between the above cases and others
		return 1;
	}

}
```

Correct Solution:

```
public class Employee {
	private boolean isPartTime;
	private int seniority;
	private int monthsDisabled;

	public Employee(boolean isPartTime, int seniority, int monthsDisabled) {
		this.isPartTime = isPartTime;
		this.seniority = seniority;
		this.monthsDisabled = monthsDisabled;
	}

	// TODO: add a function here with the signature: public boolean notEligibleForDisability()
	public boolean notEligibleForDisability() {
		return (seniority < 2 || monthsDisabled > 12 || isPartTime) ? true : false;
	}

	public double disabilityAmount() {
		// TODO: replace these checks with a single function call
		if(notEligibleForDisability())
			return 0;

		// in a real program, there would be other logic for determining
		// the amount of disability the employee is eligible for here

		// the following is a dummy return value to make the compiler happy
		// and differentiate between the above cases and others
		return 1;
	}

}
```

11. Let's now see a related refactoring, which is the decompose conditional refactoring. What happens here is that in some cases, the complexity of the conditional logic in a program can make a method hard to read and understand. Specifically we might have one or more particularly complex conditional statements. And similar to what we discussed for the previous refactoring, the conditional, if it's too complex, might tell you what happens, but obscure why it happens. To address this issue, you can do a similar thing to what we did in the previous refactoring. You can transform the condition into a method and then replace the condition with a call to that method. And if you give the right name to the method, as we saw in the last example, that can make the code much clearer and much easier to understand. In

addition here you can also do something else. Let's assume that those are the then and else part of the conditional are complex. We can do the same thing with them. So what we can do, we can modify the then and else part of the conditional by extracting the corresponding code, making also that one into a method, suitably naming the method, and then having the call to the method only in the then and else part of the conditional statement. So let's see how that works with an example. Here we have the matter that computes some charge. And it computes the charge based on some corrective uses of the date that is provided as input, or it's imagined or is just, you know, one of the fields in the class. So as you can see, there is a conditional here that checks that if the dates is before the beginning of the summer, so before summer start. Or it's after the summer end. Then it compute the charge using some winterRate. Otherwise, if we are in the summer, it will compute the quantity, the charge using a summerRate. And this is just a small example, so it might not look that complex. But, you know, just project this on more realistic code, on larger code. You can end up with the conditions that are hard to understand. And even in this case, even such a small piece of code, you have to kind of look at the conditionals, figure out what does it mean for the date to be before the summer start and after the summer end. We can make this much clearer. So, how can we do it? By applying this refactoring as we described. Let's see what happens when I apply the decompose conditionals refactoring to this method. The first thing I will do is to take this condition, create a method that perform exactly the same check, give it a meaningful name. In this case I called it notSummer, which is pretty self-explanatory, and then replacing the condition with a call to that matter. As you can see here, there's a clear improvement in the code, because here I just need to look at this statement and I see right away that the check. What the check is doing is just checking whether the date is in the summer or not. So, much easier than having to interpret this condition. And the second thing I do is to take the code that computes the charge and also in this case, creating suitable methods that compute the winterCharge and the summerCharge. And I called them exactly winterCharge and summerCharge which again is self explanatory. And then I replace this computation with a call to that method. So again, when I look at this code, I can clearly see that the charge is computed using some sort of winterCharge calculation and then using some sort of summerCharge calculation. And if I don't want to know how this is exactly computed, that's all I need to know to understand what this method does. Easier and faster than looking at this method and figuring out what it does. And if I need to look at the details, exactly like in the previous case, I can just go and look at the implementation of winterCharge and summerCharge. But I will be looking at that in its specific context. So, without having to understand everything at once. So in this way, you make it clear both why you're doing something, because it is notSummer, and what exactly you're doing. You're computing a winterCharge, or a summerCharge.



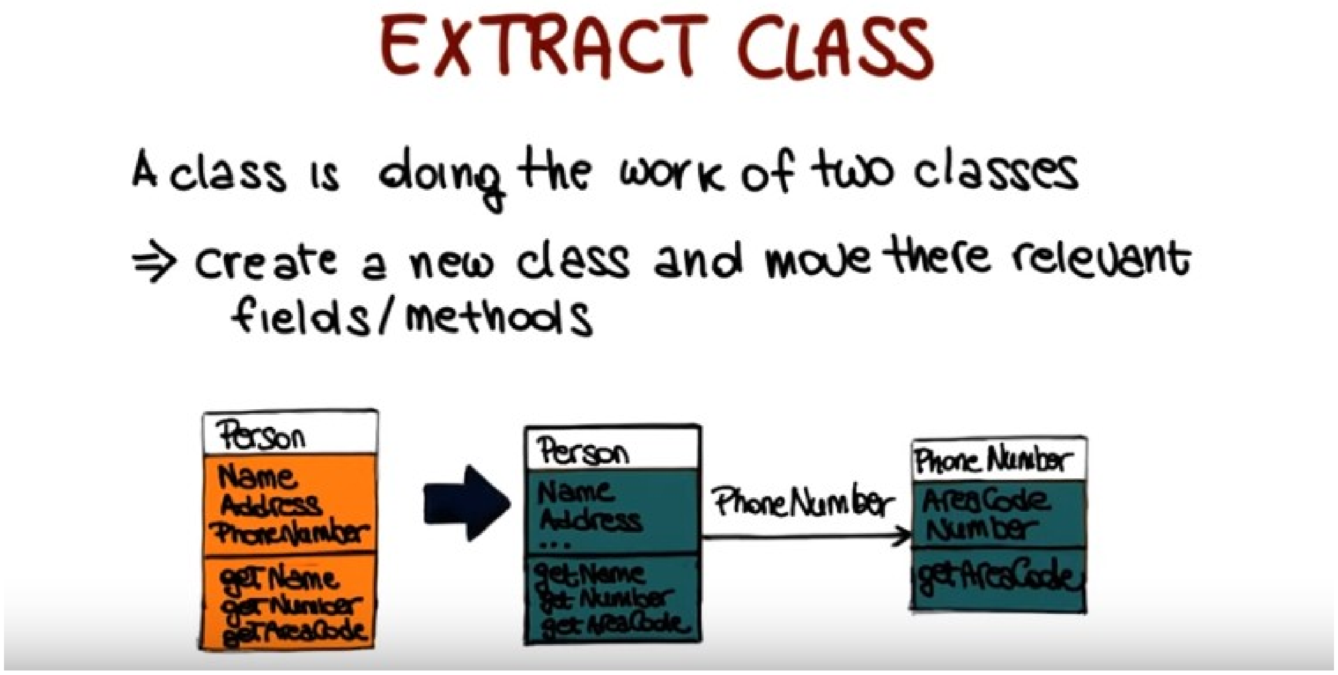
DECOMPOSE CONDITIONALS

A conditional statement is particularly complex
⇒ extract methods from conditions
modify THEN and ELSE part of the conditional

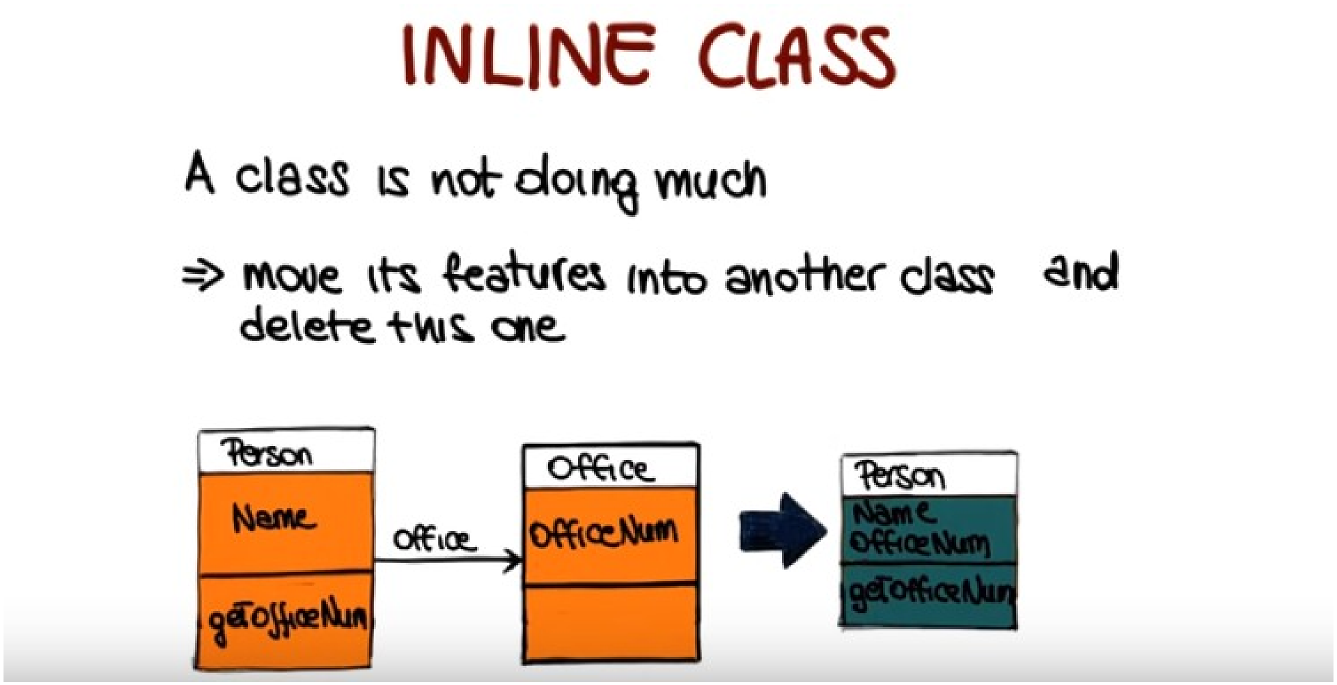
```
if(date.before(SUMMER_START)||date.after(SUMMER_END))
   charge = quantity * winterRate + winterServiceCharge;
else
   charge = quantity * summerRate;
```

```
if (not Summer(date))
   charge = winterCharge(quantity)
else
   charge = summerCharge(quantity)
```

12. We are now going to talk about the extract class refactoring. When a software system evolves, we might end up with classes that really do the work of more than one class because we keep adding functionality to the class. Therefore also they're too big, too complicated. In particular, we might end up with a class that is doing the work of two classes. Typical examples are classes with many methods and quite a lot of data, quite a lot of fields. In this case, it's normally good idea to split the class into two, so what you will do, you will create a new class and move there the relevant fields and methods from the original class. So as to have two classes, each one implementing a piece of the functionality. Let's look at an example. In this case we're going to use a UML like representation for the class. We have this class Person that ends up representing also a phone number. And imagine that we add up these pieces, you know, a little bit at the time so we end up with something that really is doing the job of the person and of the phone number. So what we can do, we can actually do exactly what we described here. We split this class into a Person class, and the Phone Number class. And then we establish a use relation, so we have a reference of the phone number class into this class. And by separating the telephone number behavior into its own class, I once more improved the structure of the code, because now I have classes that are more cohesive, and do exactly one thing.

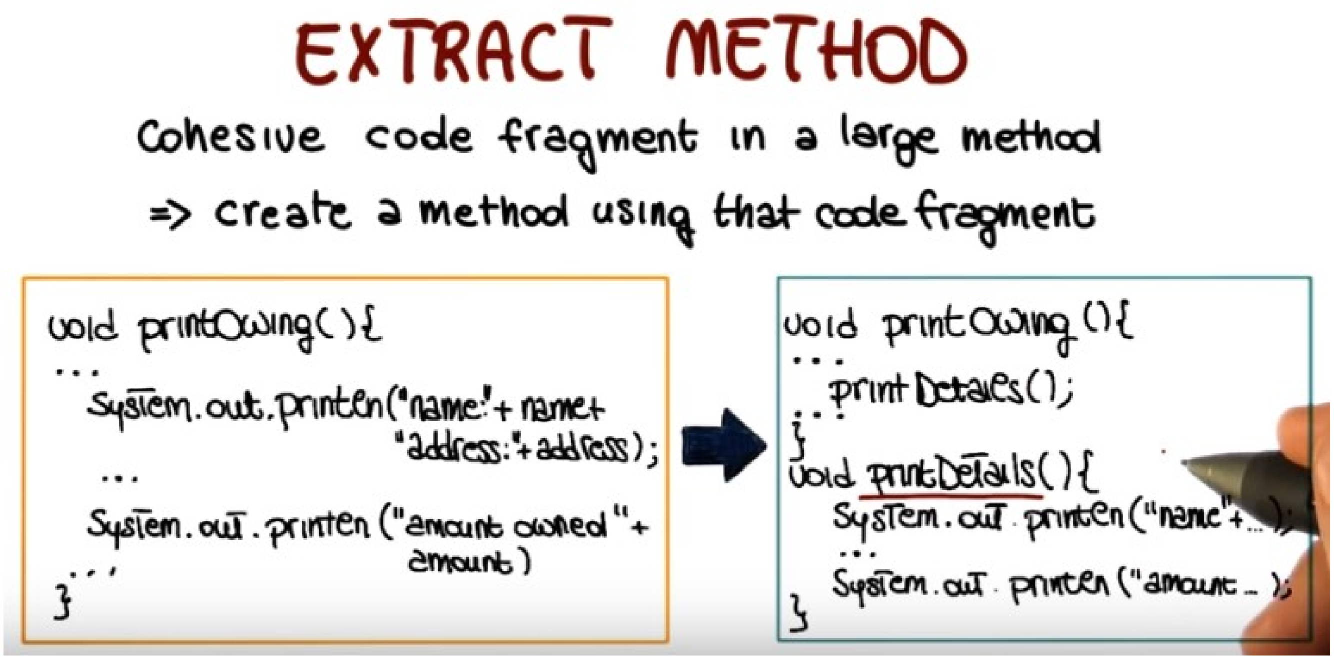


13. This new refactoring called inline class is the reverse of the extract class refactoring. And note that this is kind of a general situation in the sense that it is often the case that the refactoring also has a reverse refactoring that does exactly the opposite. So basically, un-dos, in a sense, the operation of the other refactoring. In this case, the motivation for the refactoring is that during system evolution, we might end up with one or more classes that do not do much. In this case what you want to do is to take the class that is not doing that much and move its features into another class. And then delete the original class. So lets use an example similar to the one we've used for the previous refactoring to illustrate how this works. Here we have in this case, two classes, person and office. And the person class is using the office class, but this latter class, the office class, only contains a phone number. So it doesn't really do that much. What we can do is therefore to fold the office class into the person class, by simply moving its only field into the class. And so the result will be this person class that also contains the information about the office number, and overall a simpler design for the code.



14. The next re-factoring which is also the last one that we'll see, extract method is one of the most commonly used re-factoring. As it is applicable in many, many situations. The starting point is a method that is too long and contains cohesive code fragments, that really serve a single very specific purpose. So we start from a cohesive code fragment in a large method. What we can do in this case, is to create a method using that code fragment. And then replacing the code fragment with a call to that method. Let's look at this with an example. Here over this method called print owing, and what it does, imagine that it does a lot of operations here that I'm just not listing, and then it's got a set of print statements. That are just printing a lot of details about the owing information. And then again, a lot of code after that. So what I could do in this case to simplify. The method is to transform this set of statements. They are cohesive in the sense that they do just one thing, they just print these details into a method, and then I had, replace the statements with a call to that method. Which is actually something similar to what we did as part of some the previous re-factoring's. Here I'm showing the result. So here is the method that I extracted. As you can see. It contains the code that was previously here. I give you the meaningful name, I called it printDetails so it's clear what it does. And now the print owning method is simpler. Because I still have the remaining code the one I didn't touch. But now this potentially long list of details. Of prints, of details is not replaced by a single method code. So a gain similar to the previous refactorings that we saw. If we just look at the printing method, it's very easy to figure out what this part does. Oh, print some details. And once more I really want to stress this. If you don't care about how, this is implemented and knowing that this print some details is enough. Then you're done. You don't need to understand anything more. It's clear, it's self explanatory. And if you'll need to look at what print details does, you just go and look at print details. And you look at it in isolation. So it's easier to understand what this does without having to think the rest of the code. So once more the color we factor in is just to improve your design, made the code more readable Make the code more maintainable. And also keep in mind all of these, are kind of small examples. You also always have to think about the effect that this can have on larger codebases. It can really improve a lot.

The understandabililty, and maintainability of your code. So in general, it's design.



EXTRACT METHOD

Cohesive code fragment in a large method

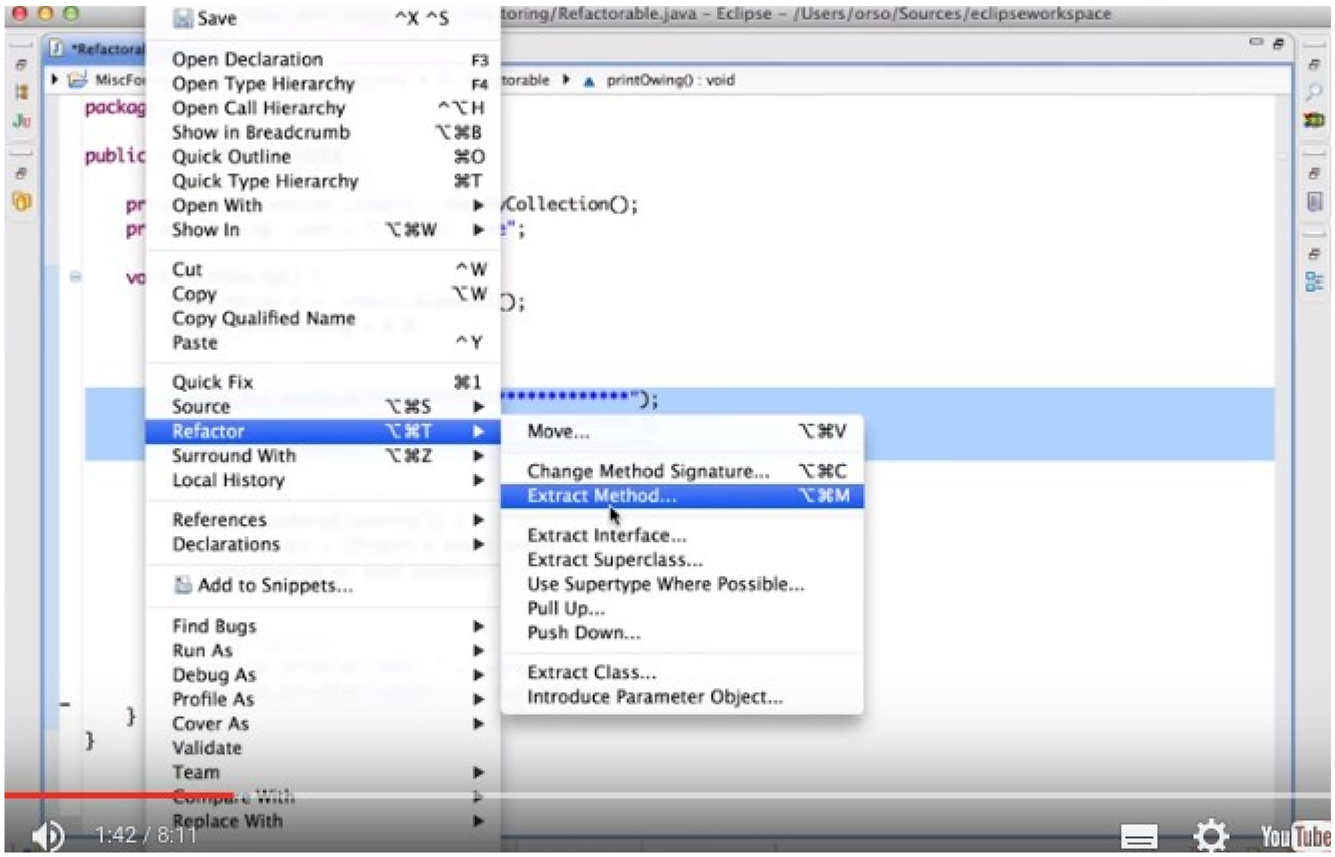
=> create a method using that code fragment

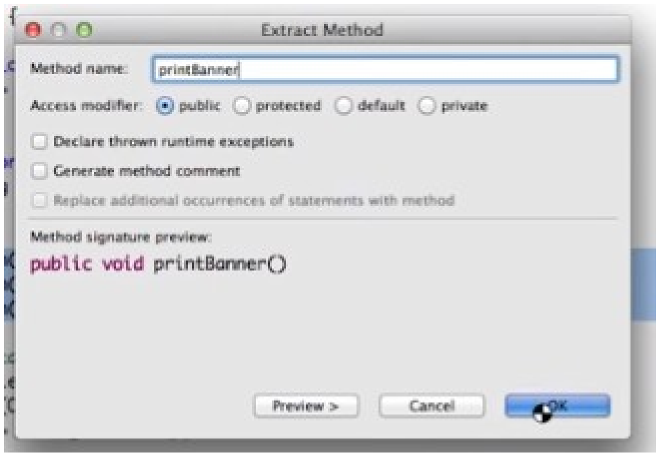
```
void printOwing(){
...
  System.out.printen("name:"+ name+
                     "address:"+address);
  ...
  System.out.printen ("amount owned"+
                      amount)
...
}
```

```
void printOwing(){
...
  printDetails();
..
}
void printDetails(){
  System.out.printen("name"+...);
  ...
  System.out.printen("amount...);
}
```

15. So now we saw, this set of re-factoring's. They're nice, but how can we actually perform re-factoring's? In some cases you'll have to do it by hand. And you'll do it in that case in small steps, so that you can check at every step that you didn't introduce any area. But there's also many cases in which at least for the more standard re-factoring's, you can just apply, you can just use a tool that actually supports re-factoring. I'm going to show you how that works, into a specific IDE, Eclipse through a demo. To show you how Eclipse, can help in performing re-factoring, in an automated way, I just opened the Eclipse editor and I maximized it. So that we can look at the code more clearly. And as you can see here, I have this class. It's called Re-factorable, it's a pretty indicative name. And what we're going to do, we're going to try to apply the extract method re-factoring to this class. And in particular, to parts of this print owing method. So this is a matter than will print owing's, as the name says. And it will do several things such as, for example, printing a banner first, then calculating the outstanding debts, and then printing some details. So the starting point for an extract method re-fractoring, is the identification of some cohesive code fragment. And here, for instance, we can see that, if we can see there, these three print statements. They are basically printing some banner, for the method. And I also put a comment here just to make that even more explicit. So this is a perfect case in which we might want to just extract this part, create an independent method, so that we can make the code more readable and maintainable. So I select, the part of the code, that I want to put in my method. I invoke the contextual menu, and as you can see there is a re-factor entry here. Here are some re-factoring's [UNKNOWN], re-factoring's that I can apply, and I'm going to select extract method. When I do that, Eclipse is going to ask me to specify a method name. I'll just call this one print banner. And as you can see, as soon as I do that, Eclipse will show me the preview, for the method that will be generated. I'm going to leave the access modifier. To public and I'm not going to change anything else. So, now when I click Ok. As you can see Eclipse modified my code so that now I have the Print Banner method down here that does exactly what that piece of code was doing before. And I just have an invocation of the Print Banner method, up here, where the code was before. And of course, this is something that we could have done by hand. It's pretty easy to do, but it's even easier, to do it using

Eclipse's capabilities. And this will become even more apparent, when we consider slightly more complex case. So here, if we look at this piece of code for instance, we can that see this code prints some details, about the always. And the reason why this case is likely more complicated, is because this code needs to know about the value of outstanding. And whereas that underscore name, is a member of the class, and therefore will be available to the method. Outstanding is a locker variable, so a method different from print, oh it wouldn't know anything about outstanding. So let's see what happens when we try to apply a re-factoring for this code. So we go again here to the re-factor menu, we select extract method, we will pick a name again. So let's call it [SOUND] print details, since this is what the code does. And as you can see here, Eclipse was able to figure out, that outstanding has to be a parameter, of this method. So if you look at the signature here, this will be very clear. So outstanding has to be passed to the matter because it's a locker variable of the print owing method. so it will not be visible to the other methods otherwise. So since eclipse figured it out, all I have to do, is to press Ok. And at this point what I will have here is my new method, for in details that takes outstanding as a parameter. And does exactly what the code was doing before. And here, where the code was, I will have my print details invocation, with outstanding as a parameter. So now, let's continue to extract methods. And let's look at a even more complex case. which is, the one involving this piece of code. So this piece of code, as you can see, will calculate the value of the outstanding debt. Will calculate the owing's, and the way in which it does that, is by considering all the orders, that are part of this enumeration. That is the declared here, and it will compute for each one, of these orders, the amount, and then added to outstanding. So what is the additional complication here? Well, the additional complication here is that this code needs to know, not only about outstanding. It also needs to know, about this enumeration, because this one is also a local variable. And in addition to that, this code also has some side effects. So outstanding, is modified as a consequence of the execution of this code. So how can we do that in the extracted method? Well lets see what the clips will do and what the clips will suggest. It will try to again re-factor this code and extract the method. In this case as you can see. The clips does two things. First of all, it figures out as before, that there are some parameters, that are needed for this method to operate correctly. The enumeration e, as we said, and the outstanding variable. In addition, if you look at the method signature Eclipse will also figure out that this method has to return, a double value. So what does this value correspond to? This value corresponds to a new value of outstanding. So if we, give a name to this method, so we just use the name, [SOUND] that I put in the comment over there. We click Ok, and this will create a method by extracting the code. And here, where the method used to be, we will have that the value of outstanding is updated. Based on the return value of calculate outstanding. So in the end if we look at this code, you can see that if we just focus, on this code it's very easy to understand what it does. It prints the banner, it calculates an outstanding value, and then it prints some details. And in case we don't care, as I said before, about the details of what these methods do, we're done. And if we care about the details we can look at each matter individually. And get exactly the same information that we got before, in a sort of a separation of concerns kind of way, by focusing on one problem at a time. So now let me do one last thing. So let me modify the code, slightly. So i'm going to go back, to the version of the code before re-factoring. So this is what we had. And I'm going to add, an additional variable here, [SOUND] called count, which I initialize to zero. Here I'm going to increase, [SOUND] the value of count at every iteration. And finally, here I'm going to print out the value of count. Okay, now that I have this code up. So let's imagine that I want to, again as I did before, extract this matter. So, I'm going to give you a second. Have a look at this and see, if you see any problem with that. Feel free to stop the video, if you need more time. So the problem here is that I have two side effects. Both outstanding and count are modified. And therefore it's not really possible to extract this method, and preserve the semantics of this code. Let's see if Eclipse will be able to figure that out. And it does. If we try to extract the matter here, you'll tell us that's an ambiguous return value. The selected block contains more than one assignment to look at variables. And the affected variables are outstanding, just a Double and Count which is an integer. So it will refuse to extract the method. So at this point if we wanted to do that we have we'll have to do the re-factoring a different way, but I don't really want to get there. I want to conclude here, and I hope this this little demo helped you realize how useful it can be to use an id that supports re-factoring that can automate this important task. And I also encourage you to try to play with this, and try to use different re-factoring's, on your code. So as to get familiar with the kind of re-factoring's that are supported by the ID. And also with the re-factoring's themselves and how should be used.





Method name: printBanner

Access modifier: [✓ public] protected default private

Declare thrown runtime exceptions

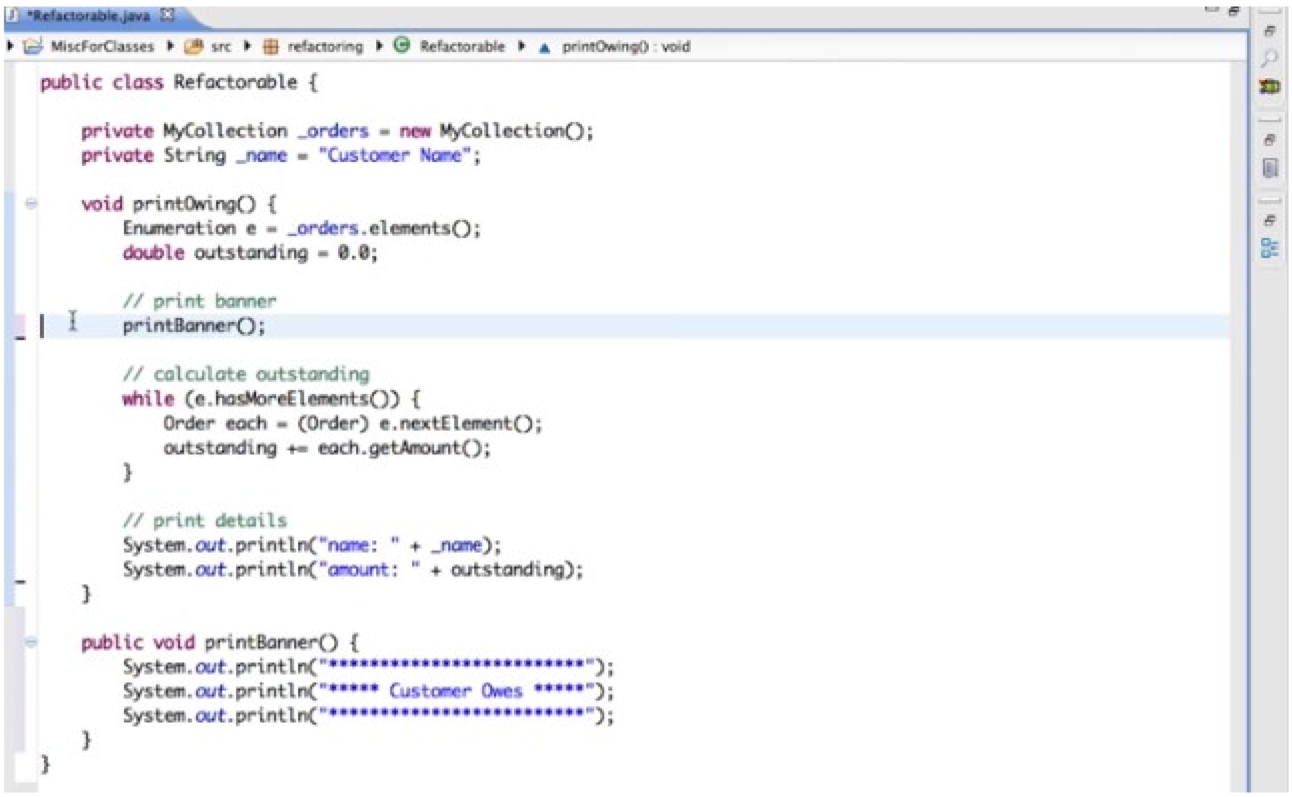
Generate method comment

Replace additional occurrences of statements with method

Method signature preview:

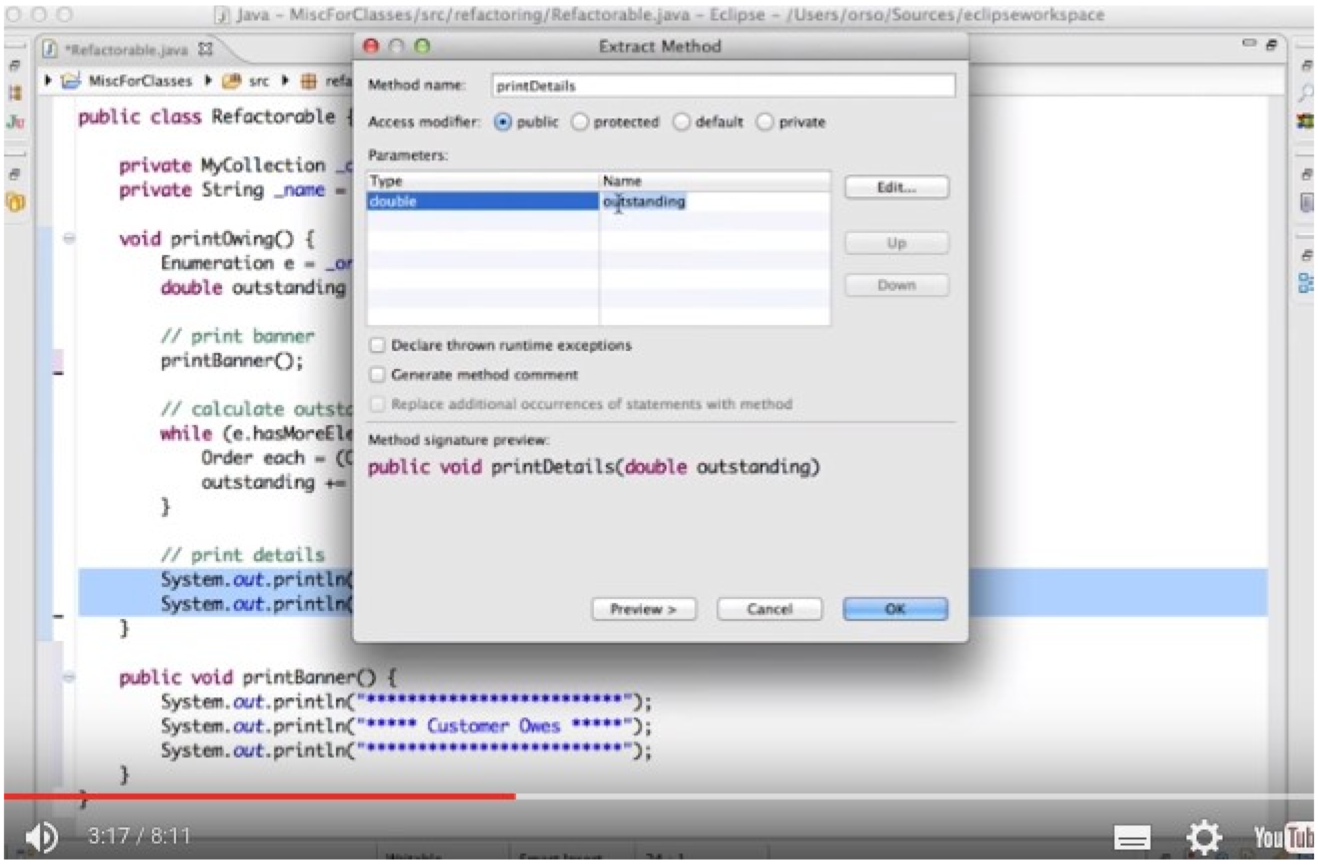
```
public void printBanner()
```

然後原代碼就自動變成了下面那樣(將 print 語句移到了新的函數 printBanner()中, 原來的 print 語句處, 被替換成了調用 printBanner()函數)



```
public class Refactorable {

    private MyCollection _orders = new MyCollection();
    private String _name = "Customer Name";

    void printOwing() {
        Enumeration e = _orders.elements();
        double outstanding = 0.0;

        // print banner
        printBanner();

        // calculate outstanding
        while (e.hasMoreElements()) {
            Order each = (Order) e.nextElement();
            outstanding += each.getAmount();
        }

        // print details
        System.out.println("name: " + _name);
        System.out.println("amount: " + outstanding);
    }

    public void printBanner() {
        System.out.println("**************************");
        System.out.println("***** Customer Owes *****");
        System.out.println("**************************");
    }
}
```

以下是弄另外兩句 print. 此時 outstanding 自動變成了 parameter.



Java – MiscForClasses/src/refactoring/Refactorable.java – Eclipse – /Users/orso/Sources/eclipseworkspace

Extract Method

Method name: printDetails

Access modifier: [✓ public] protected default private

Parameters:

Type	Name
double	outstanding

Declare thrown runtime exceptions

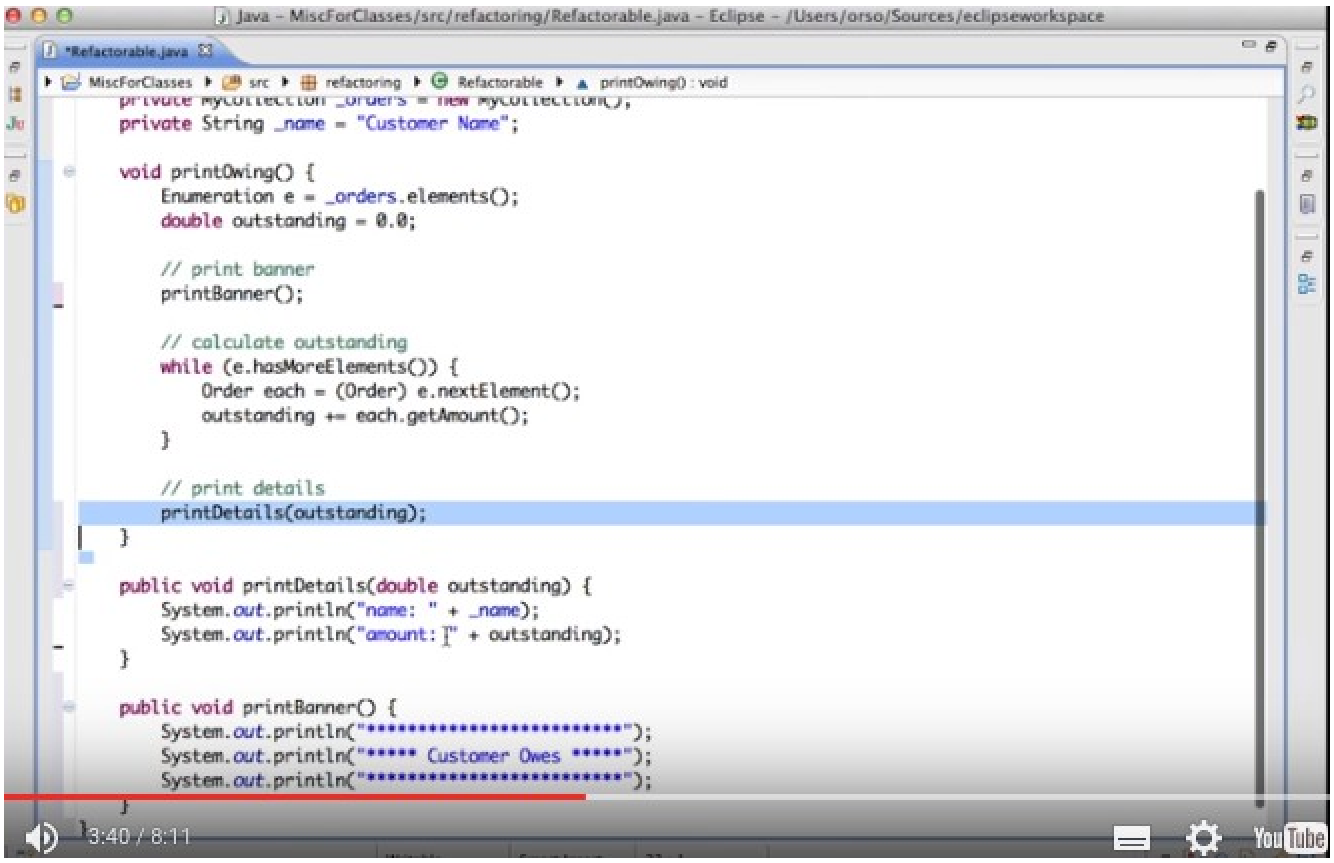
Generate method comment

Replace additional occurrences of statements with method

Method signature preview:

```
public void printDetails(double outstanding)
```

```
public class Refactorable {

    private MyCollection _o
    private String _name =

    void printOwing() {
        Enumeration e = _or
        double outstanding

        // print banner
        printBanner();

        // calculate outsta
        while (e.hasMoreEle
            Order each = ((
            outstanding +=
        }

        // print details
        System.out.println(
        System.out.println(
    }

    public void printBanner() {
        System.out.println("**************************");
        System.out.println("***** Customer Owes *****");
        System.out.println("**************************");
    }
}
```



Java – MiscForClasses/src/refactoring/Refactorable.java – Eclipse – /Users/orso/Sources/eclipseworkspace

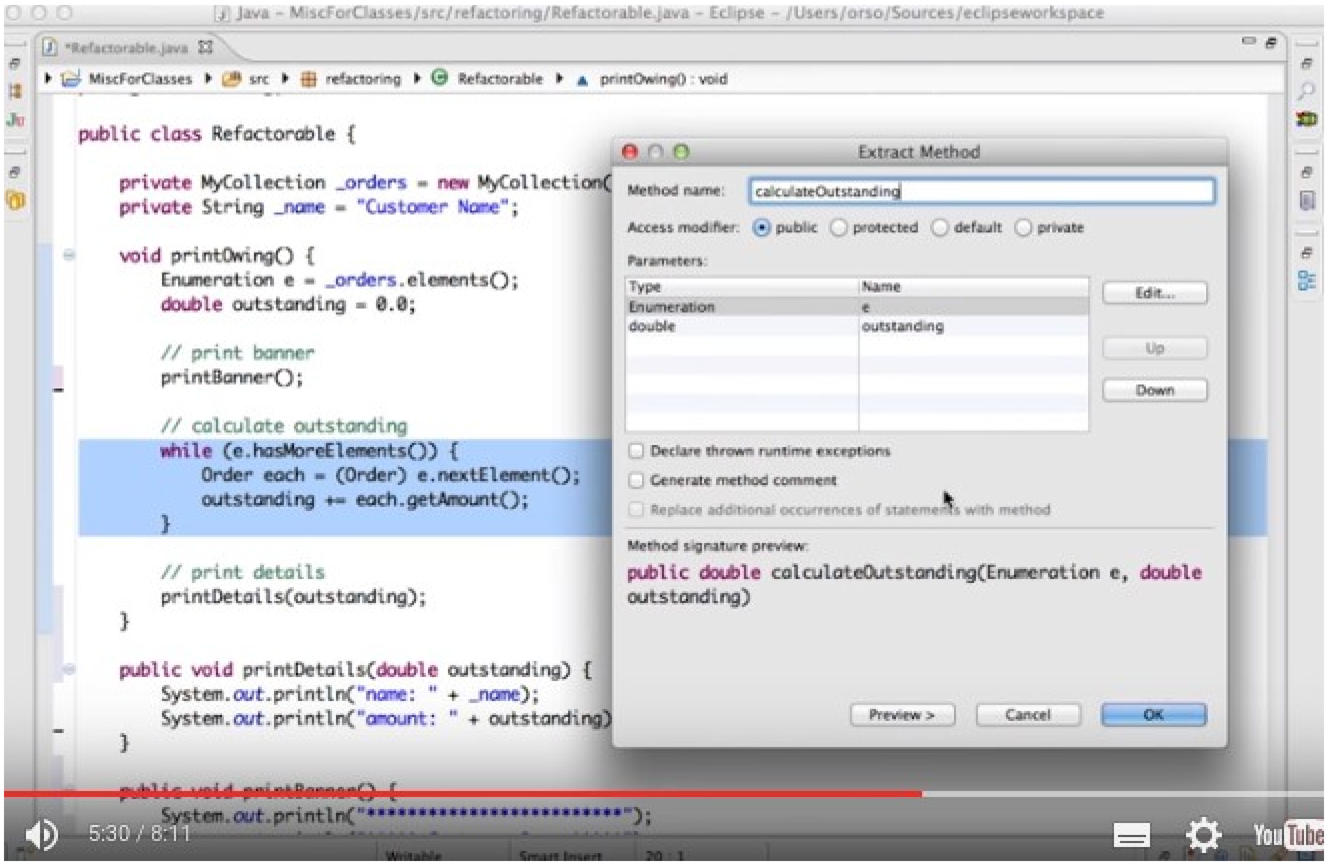
*Refactorable.java

MiscForClasses ▸ src ▸ refactoring ▸ Refactorable ▸ printOwing() : void

```java
    private MyCollection _orders = new MyCollection();
    private String _name = "Customer Name";

    void printOwing() {
        Enumeration e = _orders.elements();
        double outstanding = 0.0;

        // print banner
        printBanner();

        // calculate outstanding
        while (e.hasMoreElements()) {
            Order each = (Order) e.nextElement();
            outstanding += each.getAmount();
        }

        // print details
        printDetails(outstanding);
    }

    public void printDetails(double outstanding) {
        System.out.println("name: " + _name);
        System.out.println("amount: " + outstanding);
    }

    public void printBanner() {
        System.out.println("**************************");
        System.out.println("***** Customer Owes *****");
        System.out.println("**************************");
    }
}
```

3:40 / 8:11

然後將那個 while 弄成一個函數, 此時 outstanding 是該函數的返回值.



```
public class Refactorable {

    private MyCollection _orders = new MyCollection(
    private String _name = "Customer Name";

    void printOwing() {
        Enumeration e = _orders.elements();
        double outstanding = 0.0;

        // print banner
        printBanner();

        // calculate outstanding
        while (e.hasMoreElements()) {
            Order each = (Order) e.nextElement();
            outstanding += each.getAmount();
        }

        // print details
        printDetails(outstanding);
    }

    public void printDetails(double outstanding) {
        System.out.println("name: " + _name);
        System.out.println("amount: " + outstanding)
    }

    public void printBanner() {
        System.out.println("****************************");
```

Extract Method

Method name: calculateOutstanding

Access modifier: (● public) ○ protected ○ default ○ private

Parameters:

Type	Name
Enumeration	e
double	outstanding

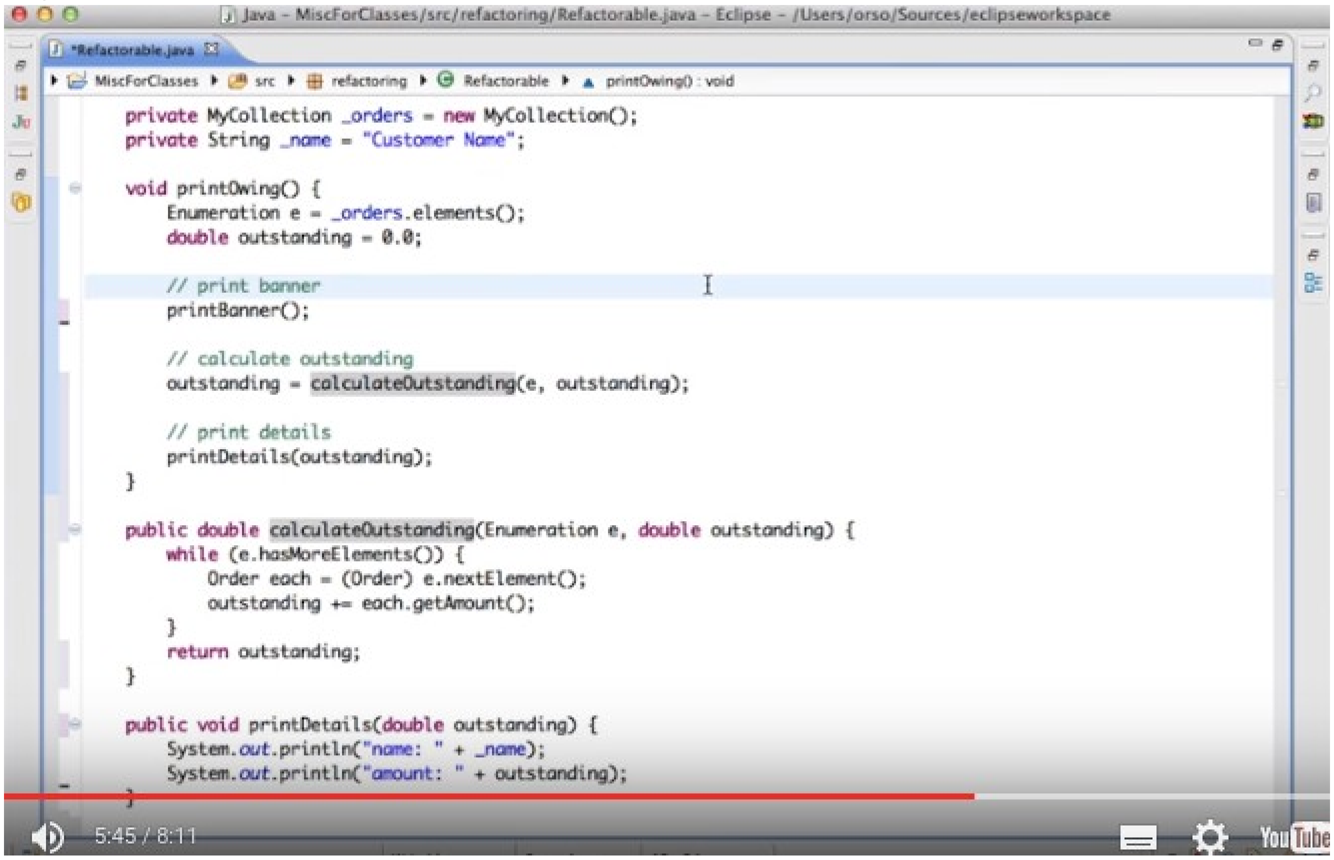
☐ Declare thrown runtime exceptions
☐ Generate method comment
☐ Replace additional occurrences of statements with method

Method signature preview:

```
public double calculateOutstanding(Enumeration e, double
outstanding)
```

Java - MiscForClasses/src/refactoring/Refactorable.java - Eclipse - /Users/orso/Sources/eclipseworkspace
*Refactorable.java
MiscForClasses ▸ src ▸ refactoring ▸ Refactorable ▸ printOwing() : void
Edit...
Up
Down
Preview >
Cancel
OK
5:30 / 8:11
Writable
Smart Insert
20 : 1

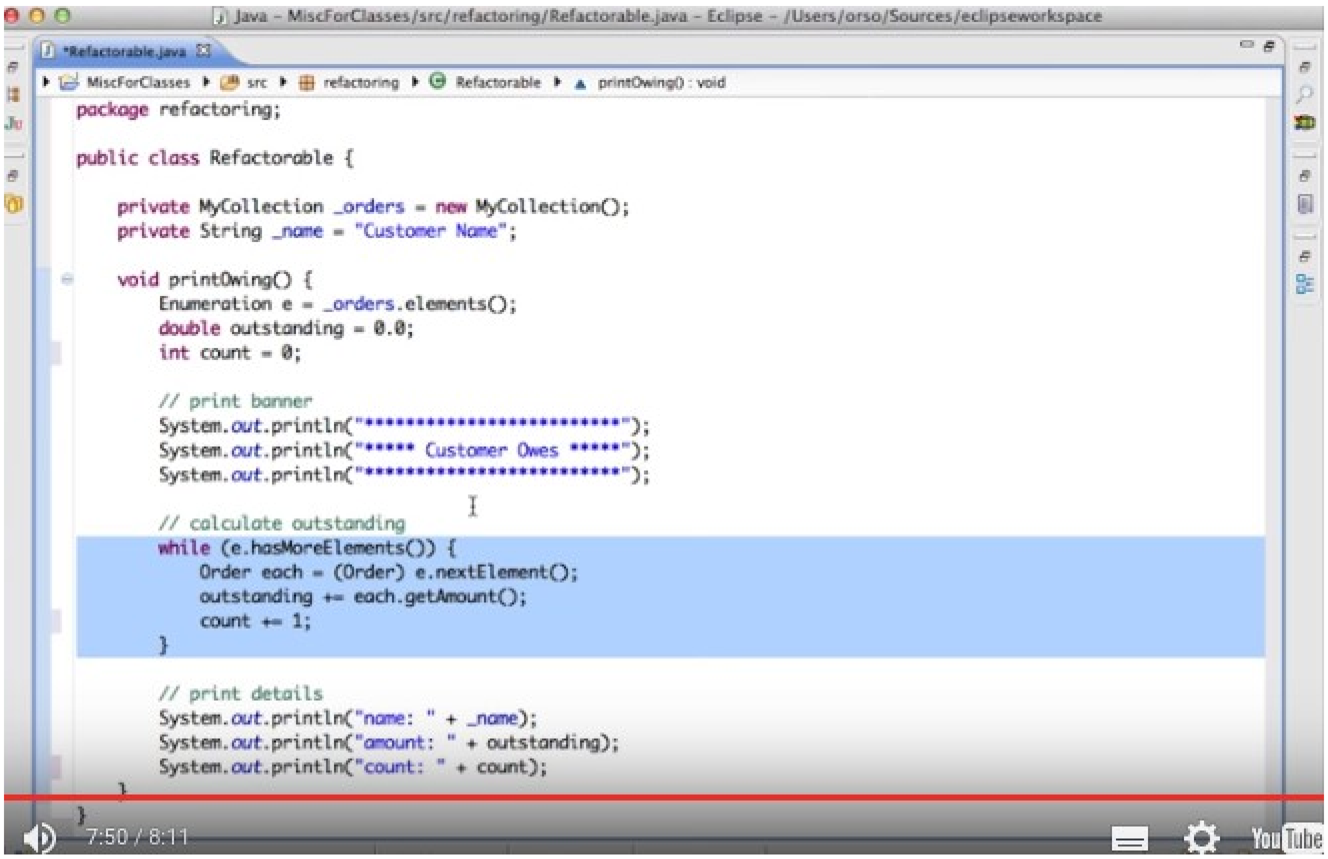
結果:



Java - MiscForClasses/src/refactoring/Refactorable.java - Eclipse - /Users/orso/Sources/eclipseworkspace

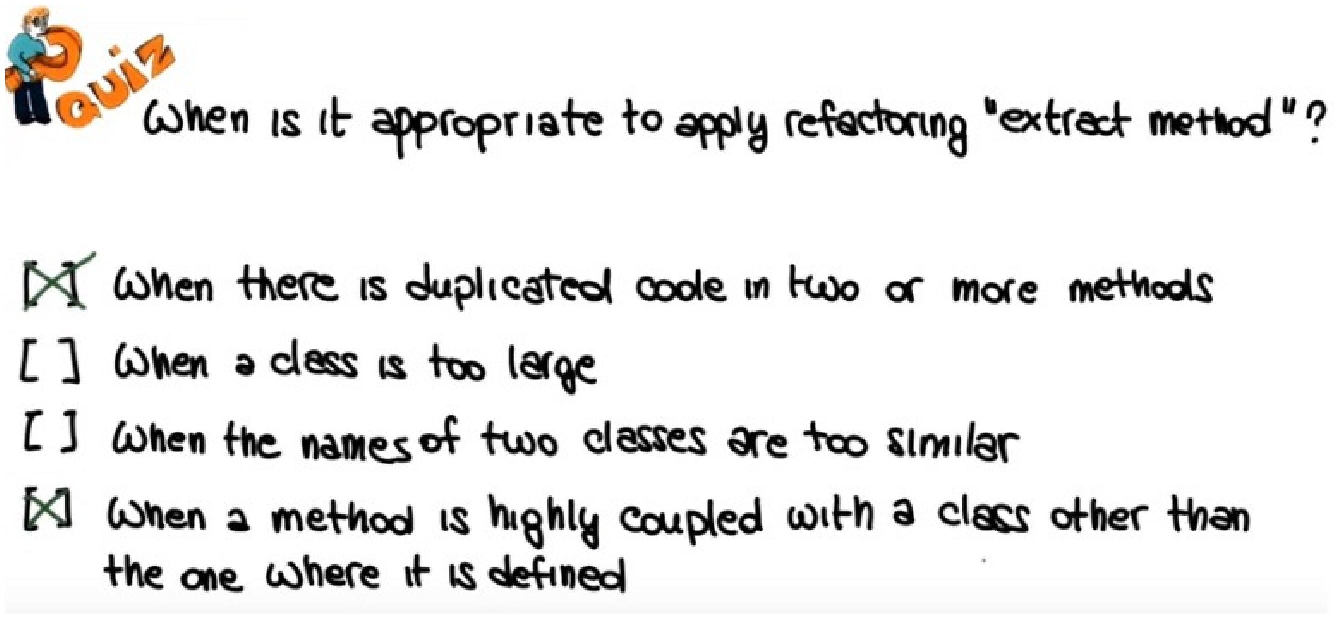
```
    private MyCollection _orders = new MyCollection();
    private String _name = "Customer Name";

    void printOwing() {
        Enumeration e = _orders.elements();
        double outstanding = 0.0;

        // print banner
        printBanner();

        // calculate outstanding
        outstanding = calculateOutstanding(e, outstanding);

        // print details
        printDetails(outstanding);
    }

    public double calculateOutstanding(Enumeration e, double outstanding) {
        while (e.hasMoreElements()) {
            Order each = (Order) e.nextElement();
            outstanding += each.getAmount();
        }
        return outstanding;
    }

    public void printDetails(double outstanding) {
        System.out.println("name: " + _name);
        System.out.println("amount: " + outstanding);
    }
```

以下是在 'factor 之前的原代碼' 中加入一個 count 變量, 故新加了三句: int count = 0, while 中的 count+=1, print details 中加了輸出 count. 下圖中的代碼, 並沒講如何 refactor, 只是提到了會出現這種情況.


Java - MiscForClasses/src/refactoring/Refactorable.java - Eclipse - /Users/orso/Sources/eclipseworkspace
*Refactorable.java
MiscForClasses ▸ src ▸ refactoring ▸ Refactorable ▸ printOwing() : void
7:50 / 8:11

```
package refactoring;

public class Refactorable {

    private MyCollection _orders = new MyCollection();
    private String _name = "Customer Name";

    void printOwing() {
        Enumeration e = _orders.elements();
        double outstanding = 0.0;
        int count = 0;

        // print banner
        System.out.println("**************************");
        System.out.println("***** Customer Owes *****");
        System.out.println("**************************");

        // calculate outstanding
        while (e.hasMoreElements()) {
            Order each = (Order) e.nextElement();
            outstanding += each.getAmount();
            count += 1;
        }

        // print details
        System.out.println("name: " + _name);
        System.out.println("amount: " + outstanding);
        System.out.println("count: " + count);
    }
}
```

16. After the demo I would like to have a little quiz about the extract method refactoring. And I would like to ask you when is it appropriate to apply the extract method refactoring. Here I have a set of possible scenarios. First one is when there is duplicated code in two or more methods. When a class is too large. When the names of two classes are too similar. Or when a method is highly coupled with a class other than the one where it is defined. So as usual, please mark all that apply.



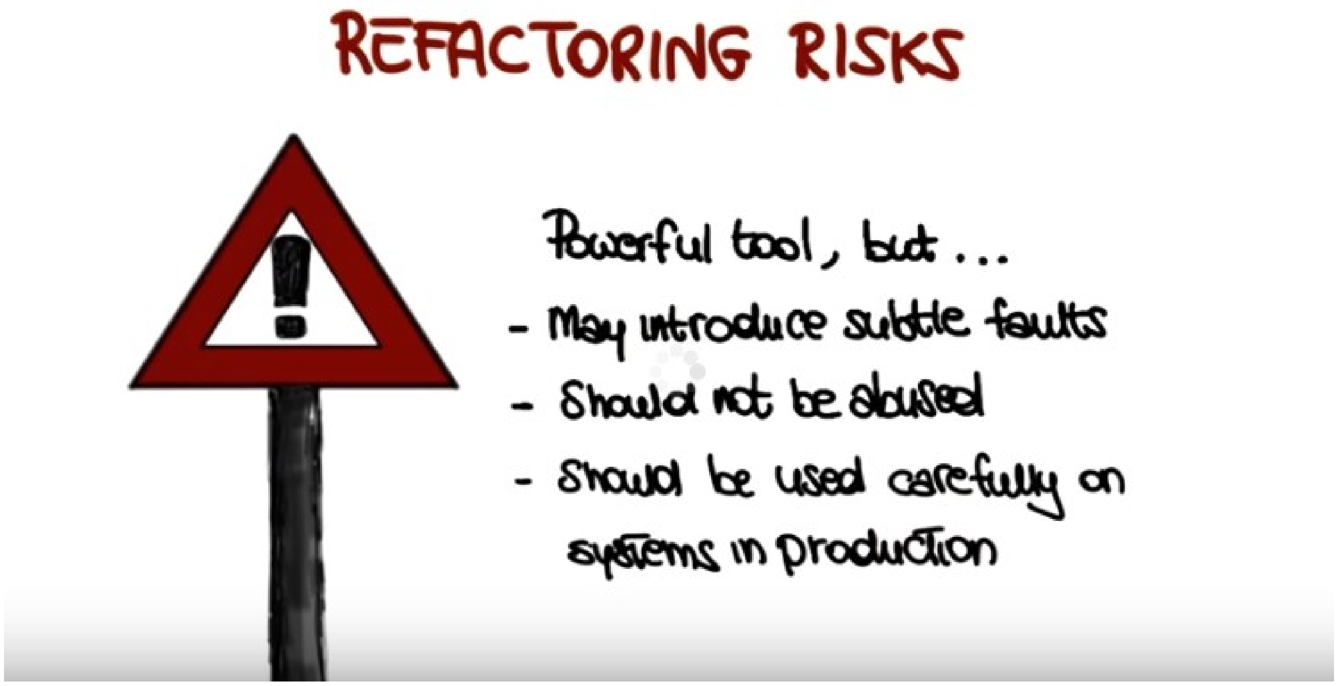
When is it appropriate to apply refactoring "extract method"?

[✓] When there is duplicated code in two or more methods
[ ] When a class is too large
[ ] When the names of two classes are too similar
[✓] When a method is highly coupled with a class other than the one where it is defined

17. The first scenario is the typical case in which it is recommended to use the extract method refactoring, when there is duplicated code in two or more methods and we want to take this code and factor is out, and basically have the two methods called a third method, which is the one we create using the refactoring. When a class is too large, normally we don't want to apply the extract. Extract method. Instead, in this cases, it is usually more appropriate to use the extract class or extract subclass refactorings. Analogously, when the names of two classes are too similar, extracting a method will normally not help much. And all we need to do in case having too similar names is actually a problem. Is to rename one of the two classes, or both, if we wish. Finally, it is definitely appropriate to apply the extract method of refactoring in cases in which a method is highly coupled with a class other than the one where it is defined. In this case, which we will discuss also later in the lesson, the extract method of refactoring allows us to extract part of the metal to With the other class. Then we can take the matter that we just extracted and move it to the class where it actually belongs. So the extract method is one of the two refactorings that it is appropriate to apply in these cases.

18. Now that we saw a number of refactorings, we also saw how refactorings can be performed automatically within an ID, I'd like to make you aware of some risks involved with the user refactorings. Refactorings are a very powerful tool, but you also have to be careful, first of all when you do more complex refactorings, you may also introduce subtle faults. What, we normally call regression errors. You might change something in the class. You might think that that's a behavior preserving transformation when considering the whole code, and instead your change is affecting the behavior of some of the other parts of the code. So, it's introducing a regression that will cause some other functionality, some other piece of functionality some other feature, to work incorrectly. So you always have to be careful, and as we saw at the beginning one way to avoid that is to run tests. Every time you make a refactoring every time you change your code and refactor your code. So is to get the least some confidence in the fact that your refactoring is indeed behavior preserving. Also consider the refactoring should not. Be abused. Refactoring should be performed when it's needed. It's useful to improve the design of your code when you see problems with the design of the code. Shouldn't just be applied for the final code because you can apply, for example, easily within a tool. So be careful not over doing it when you refactor. And for the same reason that we mentioned at the beginning, you should be particularly careful when you're using refactoring for systems that are in production. Because

if you introduce a problem, before the system goes in production, then you might be able to catch it earlier, with testing. Or before it's released. But, if you introduce a problem for a system in production, then you have to issue a new version of the code. You'll be affecting, you might be affecting some users, because the code fails on their machine. So, you have to be twice as careful, when you are doing refactoring, when you're changing your code for a system that is already in production.



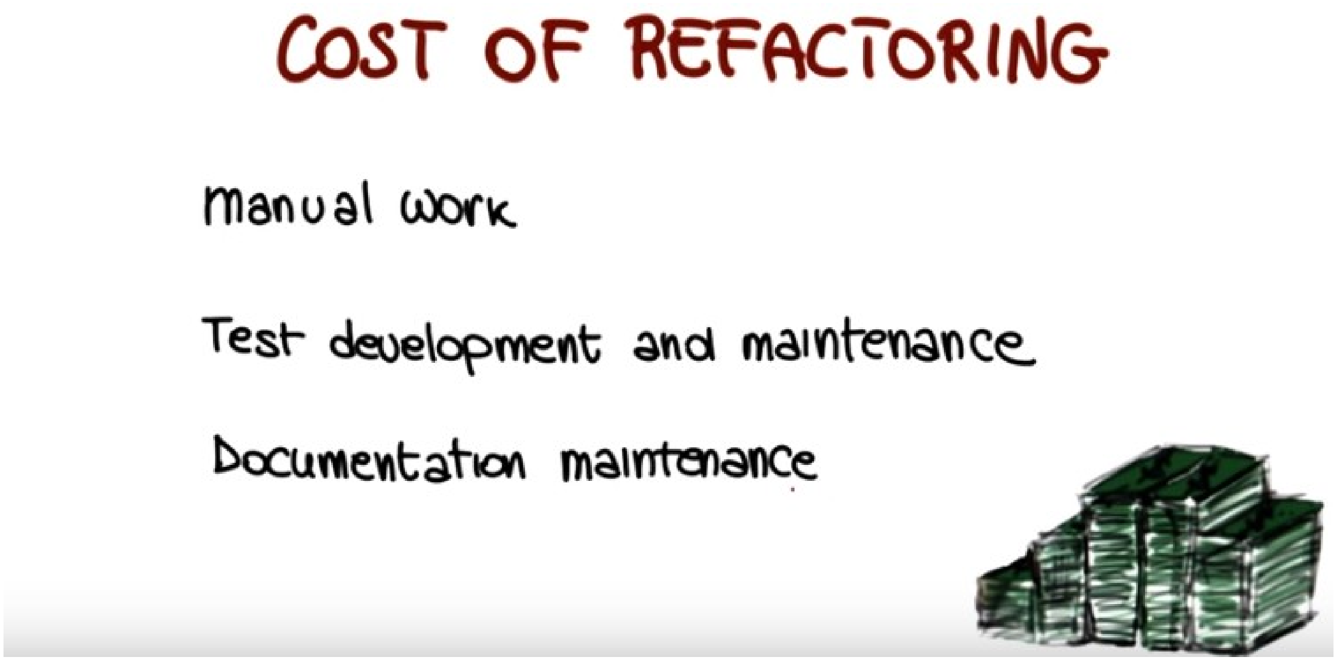
REFACTORING RISKS

Powerful tool, but ...

- May introduce subtle faults
- Should not be abused
- Should be used carefully on systems in production

19. Let's also talk about the cost of refactoring. Refactoring might be free or almost free if you're using a tool to do refactoring as we did in our demo. But that's not always the case. In many cases, refactoring involves quite a bit of manual work if you're doing some manual refactoring. And how much that costs depends on how well the operations on the source code are supported. You might have partial support from an ID. You might have complete support, in which case it's greater. Or you might have no support, in which case you have to be very careful about how you change your code and how you check that you didn't change the behavior of the code. There's also an additional cost associated with refactoring. Remember that refactoring relies heavily on testing after each small step of refactoring. So you might have to develop test cases, specifically to check your refactoring. And even if

you have an existing test because, for example, you're working some agile context and therefore you develop a lot of UNIX test cases before writing your code. And therefore you have a good regression test with it you can use every time you modify your code. Nevertheless, when you refactor and you change your code, you might need to update your test so it's not only the development of the test cases but also it's maintaining the test cases. And if you have a lot of test cases, you have to maintain more test cases. So that's a cost that is not directly visible but can affect quite a bit the overall cost of refactoring and the overall cost of system development therefore. And finally, you should not under estimate the cost of documentation maintenance. Applying refactoring may involve changes in interfaces, names, for example, names of classes. And when you make this kind of changes, you might need to update the documentation, and that's also cost. It's something that takes effort and therefore should be considered.



COST OF REFACTORING

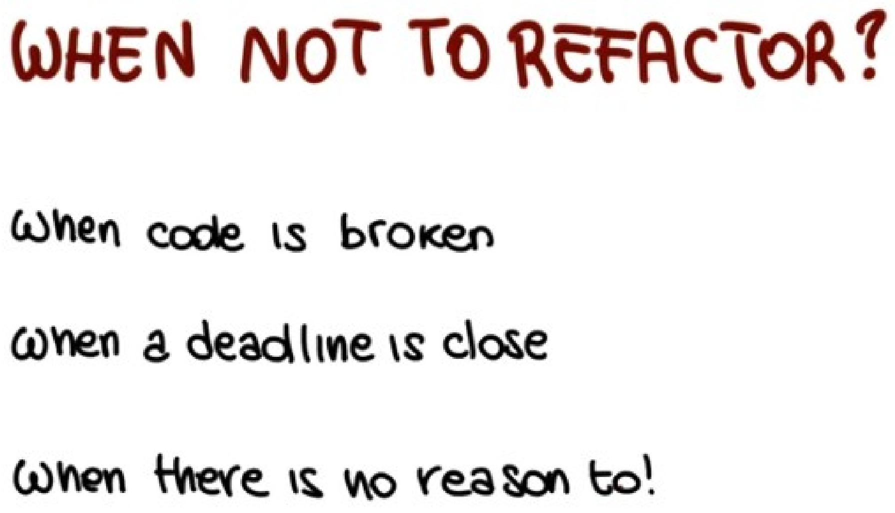
manual work

Test development and maintenance

Documentation maintenance

20. Now I want to conclude this discussion on refactoring by telling you when you should not refactor. One first clear case is when your code is broken. I want to make it very clear, refactoring is not a way to fix your code in terms of its functionality. It's a way to improve the design of your code. So if your code does not compile or does not run in a stable way, it's probably better to throw it away and rewrite it rather then trying to refactor it. By definition refactoring should maintain the functionality of the system. It should be behavior preserving. So if the code was broken before, it, it's probably going to be broken afterwards as well. You may also want to avoid refactoring when a deadline is close. Well, first of all, because refactoring might take a long time and therefore might introduce risks of being late for the deadline. And also, because of what we said before about introducing problems, you don't want to introduce problems that might take you time to fix right before a deadline. So if the deadline is too

close, you might want to avoid refactoring the code at that point. And finally, do not refactor if there is no reason to. As we said before, you should refactor on demand. You see a problem with the design of your code, with the structure of your code, it's okay to refactor. If the code is fine, there is no reason to refactor. I know that refactoring is fine, but you don't want to do it all the time. The next thing I want to discuss, after discussing when not to refactor, is when to refactor without an indication that will tell us that it's time to refactor the code. And that leads us to the discussion of a very interesting concept.

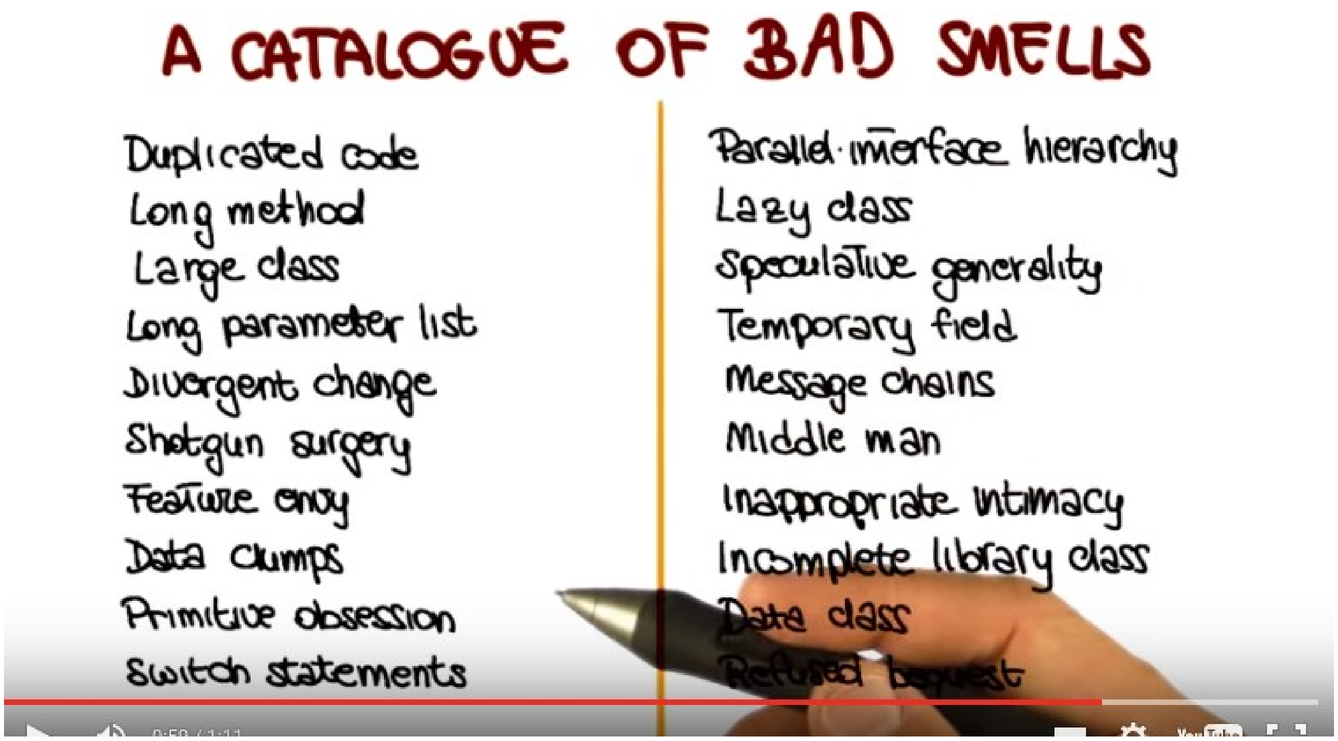

WHEN NOT TO REFACTOR?

When code is broken

When a deadline is close

When there is no reason to!

21. The concept of bad smells. What are bad smells? Well, we mentioned earlier that refactoring is typically applied when there is something that does not look right, does not feel right in the code. And that's exactly what bad smells are. Bad smells, or code smells if you wish, are symptoms in the code of a program that might indicate deeper problems. So there might be parts of my system, classes in my systems, that just don't smell right, and it feels like there's, there might be something wrong with them. And if you are an experienced developer just like Brad, you'll be able to figure out there is something wrong with the classes. You'll be able to smell that there's something wrong and you'll do something about it. And I want to mention one more, just to make sure that we're all on the same page here. That these bad smells are usually not bugs and don't prevent the program from functioning. They however indicate weaknesses in the design of the system that might cause problems during maintenance. In other words, they might make the code less maintainable, harder to understand, and so on. Just like refactorings, there's also many possible different bad smells. So what I'm providing here is just a possible list of some very common bad smells. And you can find plenty of information on this online. So what I want to do next is just to cover before finishing the lesson a few of those to show you some examples of smells and what you can do about them.



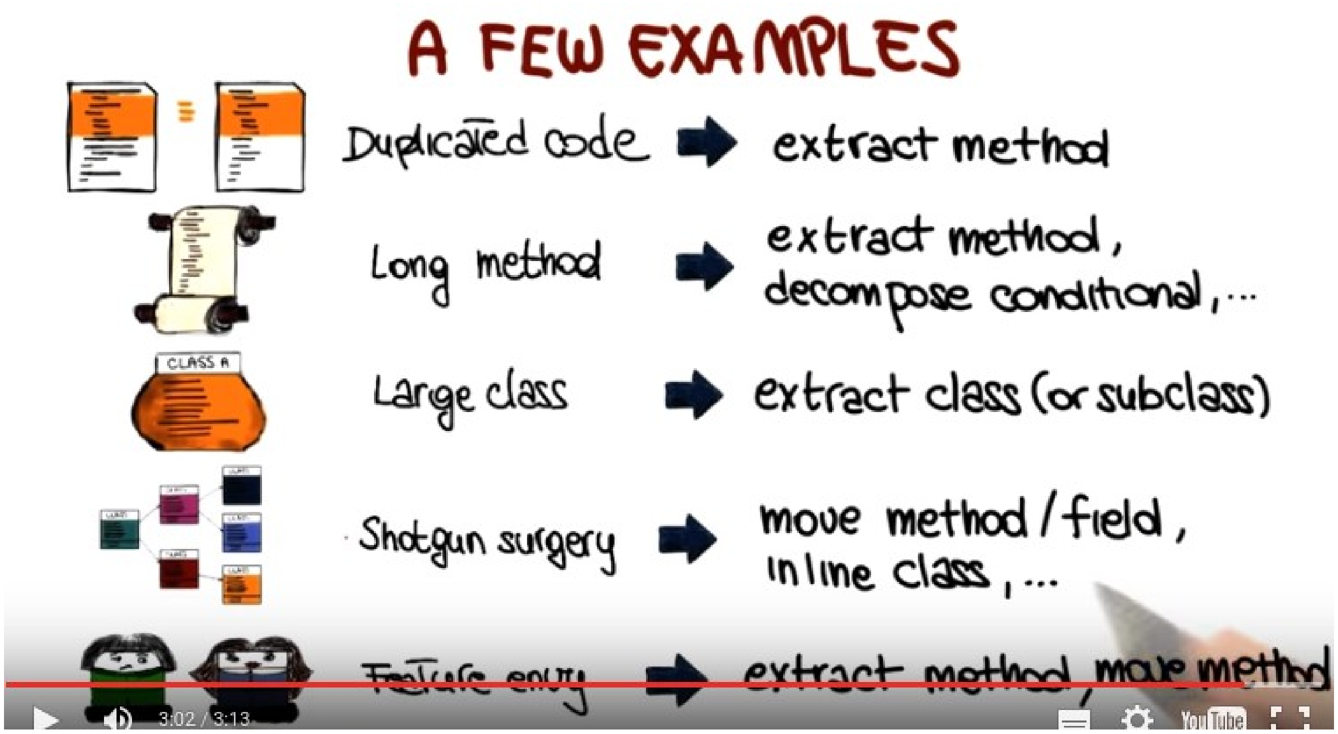
A CATALOGUE OF BAD SMELLS

Duplicated code	Parallel interface hierarchy
Long method	Lazy class
Large class	Speculative generality
Long parameter list	Temporary field
Divergent change	Message chains
Shotgun surgery	Middle man
Feature envy	Inappropriate intimacy
Data clumps	Incomplete library class
Primitive obsession	Data class
Switch statements	Refused bequest

注意上圖中有 switch statements(左下).

22. The first example I want to mention is this called duplicated code. So what happens here what the symptom is, is that you have the same piece of code. The same fragment of code or code structure replicated in more than one place. And that's pretty common when we do for example copy and paste programming. Is something that we mention at the beginning of the lessons. So for example we are just

instead reimplementing a piece of functionality we know we already have. We simply copy from a different part of the code. So what do you do if you have duplicated code? This can be a problem over time because we might end up with a lot of duplication in your system. You can use the extract method. Refactoring that we just saw, and basically create a method that has exactly the same function as this fragment of code and then replace the fragment of code with an invocation to run and you will do it in all the places where the code is duplicated. That simply finds the code and favors reuse, because there can be more places that benefit from that additional method. Especially if it implements some popular piece of functionality. Another example of best mal a typical one is the long method. So you have a very long method with a lot of statements. And we know that the longer procedure, the more difficult it is to understand it and maintain it. So what I'm going to do in this case is to factor in such as an extract method or a decompose conditional to make the code simpler, shorten it. And extract some of the functionality into other methods. So basically break down the method in smaller methods that are more cohesive. Another typical example of best mail which is something that can happen very commonly during maintenance, is that you keep adding functionality to a class and you end up with a large class. So class is clearly to big. It contains too many fields too many methods, and is just too complex to understand. This case the obvious solution is to use the extract class or subclass and basically break down the class in multiple classes. Each one with a more cohesive piece of functionality. So, the classes are more cohesive, are more understandable, and the overall structure The structure of the system is improved. Shotgun surgery is an interesting smell and the case here is we are in a situation and you, probably will happen to you, it definitely happened to me, in which every time you make some kind of change to the system you have to make many little changes. All over the place to many different classes. And this can be a symptom of the fact that the functionality is spread among these different classes. So there's too much coupling between the classes and too little cohesion within the classes. Also in this case you can use refactoring, for example by using the move method or move field or inline class to bring the pieces of related functionality together. So that your resulting classes are more cohesive, you reduce the dependencies between the different classes, and you address this problem. Because at this point, each class is much more self-contained and therefore it can be modified by itself without having to affect the rest of the system. The last smell I want to mention is one I really like, is the feature envy, and it refers to a method that seems more interested In a class other than the one it belongs to. So for example this method is using a lot of public fields of another class, is calling a lot of methods of the other class. And so in this case the solution is really clear. What you want to do it to perform the extract method refactoring and then the move method refactoring so as to take the jealous method out of the class where it doesn't belong and get it home. To the class where it really belongs and once more the effect of this is that you decrease the coupling between the two classes and therefore you have a better system and also you eliminate the envy. Which is always a good thing.



Duplicated code → extract method

Long method → extract method, decompose conditional, ...

Large class → extract class (or subclass)

Shotgun surgery → move method / field, inline class, ...

Feature envy → extract method, move method

23. The first example I want to mention is this called duplicated code. So what happens here what the symptom is, is that you have the same piece of code. The same fragment of code or code structure replicated in more than one place. And that's pretty common when we do for example copy and paste programming. Is something that we mention at the beginning of the lessons. So for example we are just instead reimplementing a piece of functionality we know we already have. We simply copy from a different part of the code. So what do you do if you have duplicated code? This can be a problem over time because we might end up with a lot of duplication in your system. You can use the extract method. Refactoring that we just saw, and basically create a method that has exactly the same function as this fragment of code and then replace the fragment of code with an invocation to run and you will do it in all the places where the code is duplicated. That simply finds the code and favors reuse, because there can be more places that benefit from that additional method. Especially if it implements some popular piece of functionality. Another example of best mal a typical one is the long method. So you have a very long method with a lot of statements. And we know that the longer procedure, the more difficult it is to understand it and maintain it. So what I'm going to do in this case is to factor in such as an extract method or a decompose conditional to make the code simpler, shorten it. And extract some of the functionality into other methods. So basically break down the method in smaller methods that are more cohesive. Another typical example of best mail which is something that can happen very commonly during maintenance, is that you keep adding functionality to a class and you end up with a large class. So class is clearly to big. It contains too many fields too many methods, and is just too complex to understand. This case the obvious solution is to use the extract class or subclass and basically break down the class in multiple classes. Each one with a more cohesive piece of functionality. So, the classes are more cohesive, are more understandable, and the overall structure The structure of the system is improved. Shotgun surgery is an interesting smell and the case here is we are in a situation and you, probably will happen to you, it definitely happened to me, in which every time you make some kind of change to the system you have to make many little changes. All over the place to many different classes. And this can be a symptom of the fact that the functionality is spread among these

different classes. So there's too much coupling between the classes and too little cohesion within the classes. Also in this case you can use refactoring, for example by using the move method or move field or inline class to bring the pieces of related functionality together. So that your resulting classes are more cohesive, you reduce the dependencies between the different classes, and you address this problem. Because at this point, each class is much more self-contained and therefore it can be modified by itself without having to affect the rest of the system. The last smell I want to mention is one I really like, is the feature envy, and it refers to a method that seems more interested In a class other than the one it belongs to. So for example this method is using a lot of public fields of another class, is calling a lot of methods of the other class. And so in this case the solution is really clear. What you want to do it to perform the extract method refactoring and then the move method refactoring so as to take the jealous method out of the class where it doesn't belong and get it home. To the class where it really belongs and once more the effect of this is that you decrease the coupling between the two classes and therefore you have a better system and also you eliminate the envy. Which is always a good thing.

24. So let's look at this one by one. The fact the program takes too long to execute is not really a bad smell. It probably indicates some problem with the code and the fact that we might need to modify the code to make it more efficient, but it's not something that we will normally classify it as a bad smell, so we're not going to mark it. The second one, conversely, is definitely a bad smell. The fact that the method is too long is a typical example of bad smell and one in which we might want to apply some refactoring, for example, the extract method or the decomposed conditional refactorings. There's definitely nothing wrong with the fact that the classes cat and dog are subclasses of class animal. Actually, that sounds pretty appropriate, so this is not a problem and definitely not a bad smell. Whereas the fact that every time we modify method M1, we also need to modify method some other method M2 as a typical example of bad smell. So this can actually be considered a specific example of what we just called "shotgun surgery." So it is a case in which we might want to use, for instance, the move method refactoring to fix the issue.