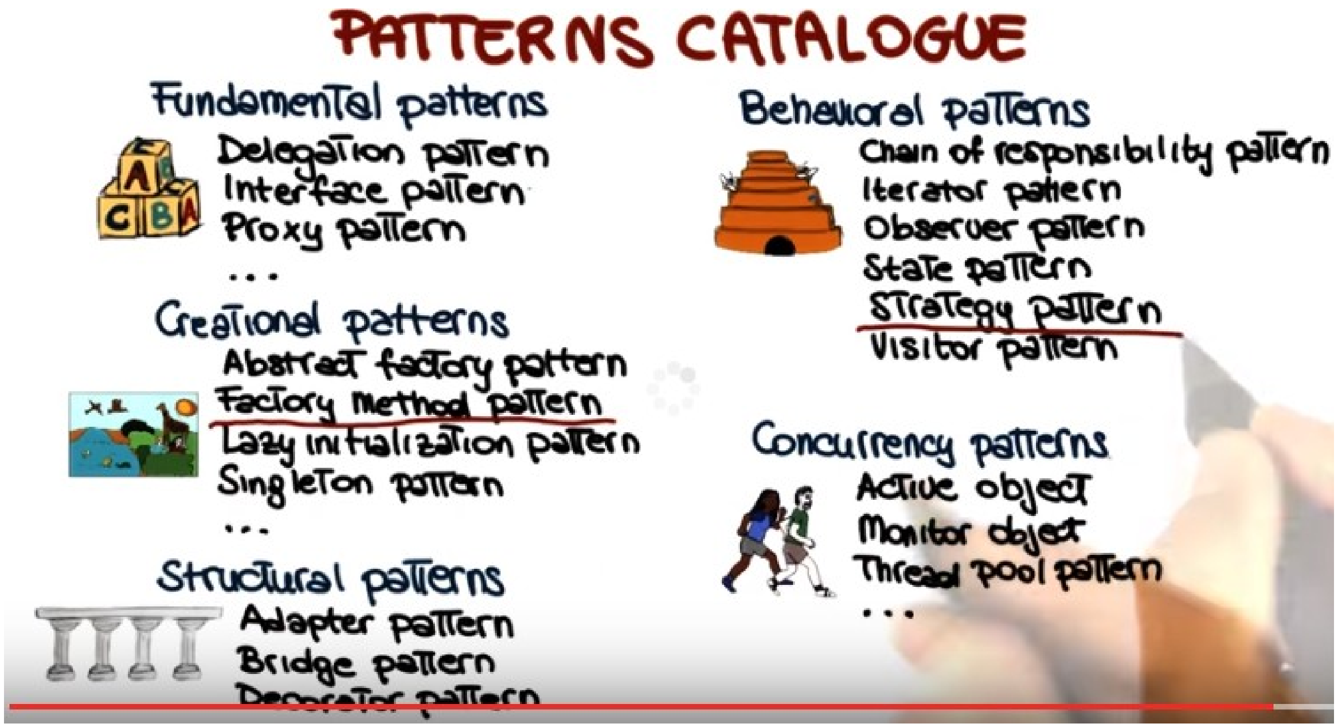
1. In the last lesson, we talked about design, and we saw how difficult it can be to come up with a good and effective design for a given software system. To help address these difficulties, we will discuss design patterns, which can support design activities by providing general, reusable solutions to commonly occurring design problems. Similar to architectural styles, design patterns can help developers build better designed systems by reusing design solutions that worked well in the past and by building on those solutions.

2. Let's start our decision of design patterns by looking at the history of patterns. As you know, I like to give this sort of historical perspective on how and when concepts were defined. In this case, we have to go back to 1977, when Christopher Alexander, an American professor of architecture at UC Berkeley, introduces the idea of patterns, successful solutions to problems, in his book called a Pattern Language. The book contain about 250 patterns. And the idea is that occupants of a building should be able to design it. And the patterns in the book provide a way to do that. And this idea of design patterns, so, a formal way of documenting successful solutions to problems, inspired several other disciplines. In particular, in 1987, Ward Cunningham and Kent Beck leveraged this idea of Alexander's of patterns in the context of an object oriented language. And in this specific the language was a small talk. Some of you might know the language. So what Cunningham and Beck did, was to create a 5 pattern pattern language for guiding novice small talk programmers. So they did an experiment and had several developers using their patterns, and the experiment was extremely successful. The users were able to create elegant designs using the provided patterns. And in case you are interested in reading about it, Cunningham and Beck reported the results in the article, Using Pattern Languages for Object Oriented Programs, which was published at the International Conference on Object Oriented Programming, Systems, Languages, and Applications, also called OOPSLA, in 1987. At the same time, Eric Gamma was working on his dissertation, whose topic was the importance of patterns and how to capture them. Between 1987 and 1992, there were several workshops related to design patterns. And in 1992, Jim Coplien compiled a catalog of C++ items, which are some sort of patterns, and he listed this catalog of patterns in his book, which was titled Advanced C++ Programming Styles and Idioms. Finally, in 1993 and 1994, there were several additional workshops focused on patterns. And this workshop brought together many patterns folks, including these 4 guys, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. These guys are also known as the gang of 4. And the result of this collaboration was the famous book Design Patterns: Elements of Reusable Object Oriented Software. So this is basically The Book on design patterns. If you want to buy a book on design pattern, this is the one you should get.

3. This book contains a patterns catalog which is a number of design patterns classified by purpose. And there are five main classes of patterns. There are fundamental patterns which are the basic patterns. There are creational patterns which are the patterns that support object creation. Then there are structural patterns and these are patterns that help compose objects, put objects together. The next class of patterns are behavioral patterns and these are patterns that are mostly focused on realizing interactions among different objects. Finally, there are concurrency patterns and these are patterns that support, as the name says, concurrency, so they're more related to concurrency aspects. And for each of these classes there are a number of specific patterns, and here I'm just listing some of them. Clearly we cannot cover in one lesson all of these patterns, but what I want to do is to cover at least a few of those to give an idea of what patterns are and how they can be used. In particular, we will see in detail the Factory Method Pattern and the Strategy Pattern. And we will also discuss a few more patterns at a higher level.



PATTERNS CATALOGUE

Fundamental patterns
- Delegation pattern
- Interface pattern
- Proxy pattern
- ...

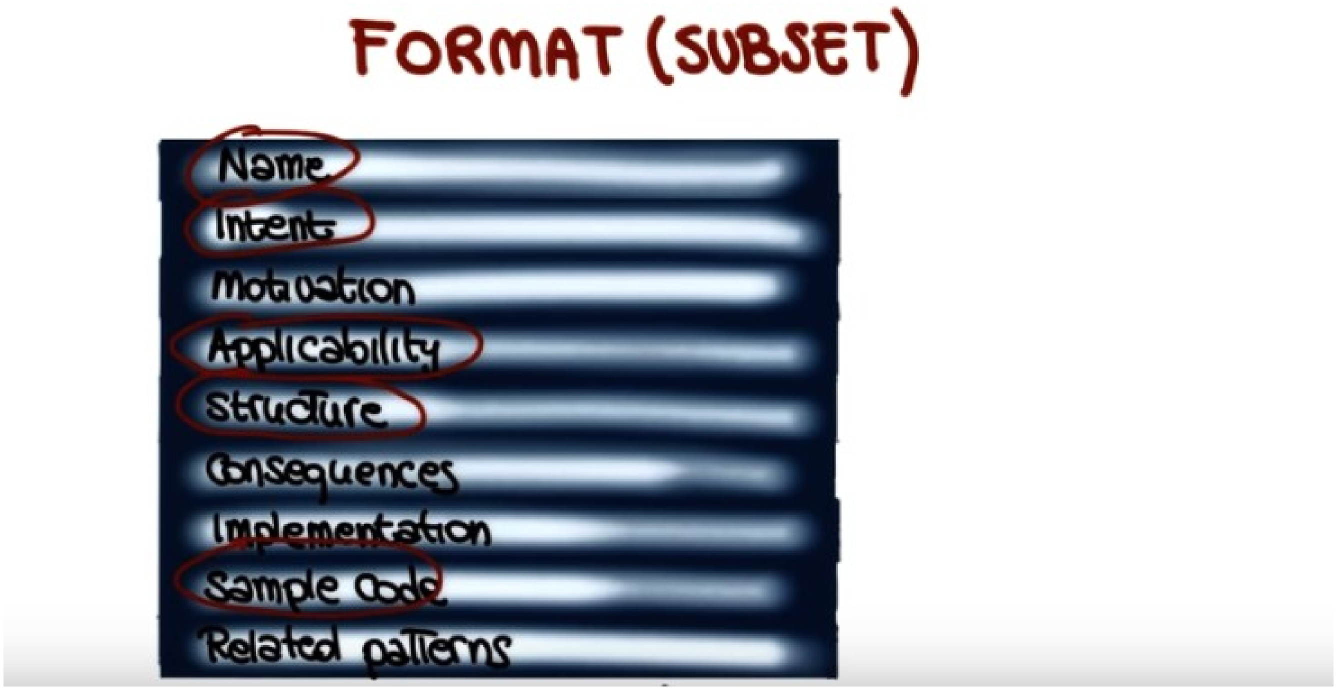
Creational patterns
- Abstract factory pattern
- Factory method pattern
- Lazy initialization pattern
- Singleton pattern
- ...

Structural patterns
- Adapter pattern
- Bridge pattern
- Decorator pattern

Behavioral patterns
- Chain of responsibility pattern
- Iterator pattern
- Observer pattern
- State pattern
- Strategy pattern
- Visitor pattern

Concurrency patterns
- Active object
- Monitor object
- Thread pool pattern
- ...

4. So let's start by seeing how patterns are defined. So what is the format of the pattern definitions. If we look at the Gang of Four's book we can see that these definitions contain a lot of information. In fact, what I'm listing here is just a subset of this information. In this lesson, what I want to do is to focus on four essential elements of a design pattern. It's name, the intent which is the goal of the pattern. The patterns applicability which is the least of situations or context in which the pattern is applicable. I also want to cover the structure and participants. Which is the static model that describes the elements, so normally the classes or the object involved in the pattern. In addition to that the structure also describes the relationships, responsibilities and collaborations among these classes or objects. Finally what I want to cover is sample code. So examples that illustrate the use of patterns.


FORMAT (SUBSET)

Name

Intent

Motivation

Applicability

Structure

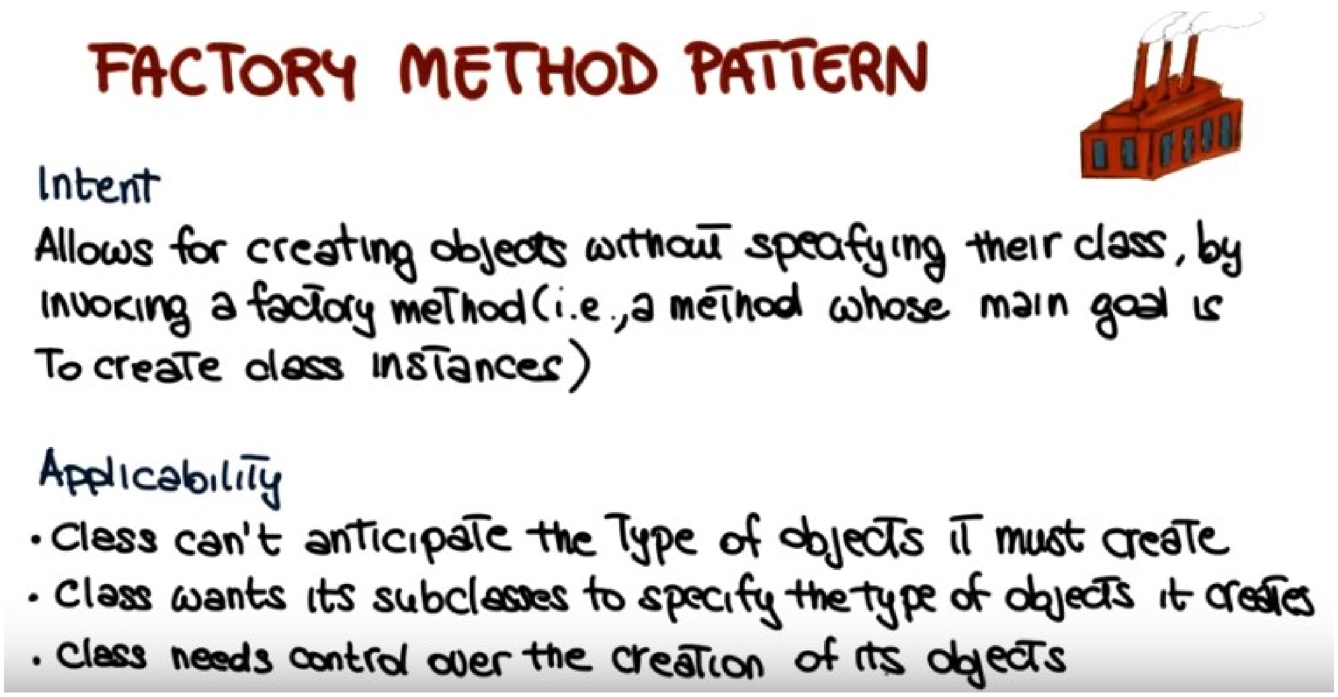
Consequences

Implementation

Sample Code

Related patterns

5. Let's now look at the first design pattern that we will discuss, the factory method pattern. And I'm going to start by discussing the intent of the pattern and its applicability. As far as the intent is concerned, the factory method pattern allows for creating objects without specifying their class, by invoking what we call a factory method (由後面的 example 可知, 這個 factory method 不在這個 class 中, 而是在別的 class 中. 看了後面的 example 就明白 了). And what that is, is a method whose main goal is to create class instances. So when is this pattern useful? So when is it applicable? For example, it is applicable in cases in which a class cannot anticipate the type of object it must create. That is, the type of an object is not known at compile time, is not known until the code runs. A typical example of this, are frameworks. So if you ever used a framework, you will know that, normally, frameworks only know about interfaces and abstract classes. So the exact type of the objects of these classes is only known at runtime. Second case in which the factory method pattern is applicable, is when a class wants its subclasses to specify the type of objects it creates. And we'll see an example of this in a minute. Finally, factory method patterns are applicable when a class needs control over the creation of its objects. And in this case, one possible example is when there is a limit on the number of objects that can be created. Special example, it's a singleton. If you're familiar with a singleton, a singleton is a class for which only one instance can be created. The factory method pattern is perfect in these cases, because it allows to control how many objects gets created. So in this case, it would allow the creation only of a single object. And from the second time that it is invoked, it will just return the object that was previously created. Now let's go ahead and see how this pattern actually works, and let's do that by discussing the structure and the participants for the pattern. The structure that is represented here, using the UML notation, includes three classes, the Creator, the ConcreteCreator, and the Product. The Creator provides the interface for the factory method. So this here, is the interface for the factory method that, when invoked, returns an object of type Product. The ConcreteCreator provides the actual method for creating the Product. So this method is a concrete implementation of this interface. Finally, the Product is the object created by the factory method. So summarizing, we have the interface for the factory method, the actual implementation of the summary method, and the object that is created by the factory method, when it is invoked. So let's look at an example of this pattern.



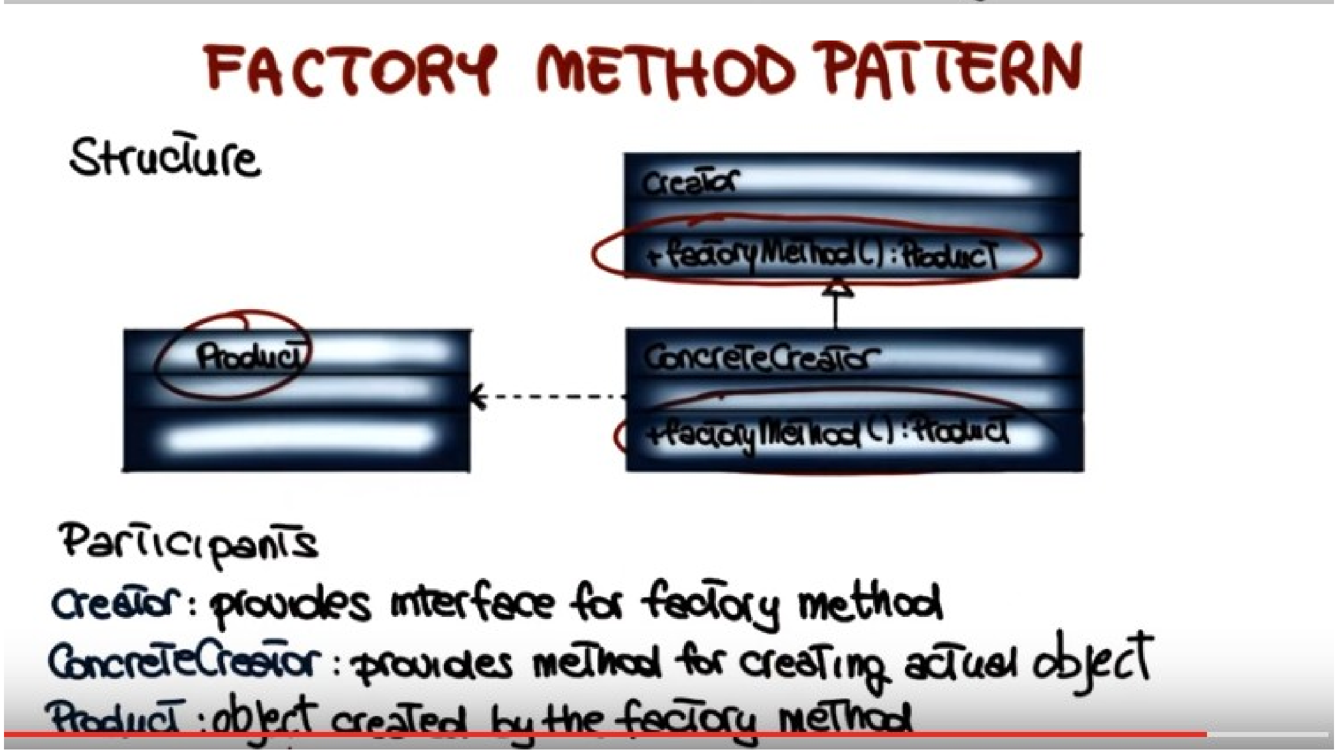
FACTORY METHOD PATTERN

Intent

Allows for creating objects without specifying their class, by invoking a factory method (i.e., a method whose main goal is to create class instances)

Applicability

- Class can't anticipate the type of objects it must create
- Class wants its subclasses to specify the type of objects it creates
- Class needs control over the creation of its objects



FACTORY METHOD PATTERN

Structure

Participants

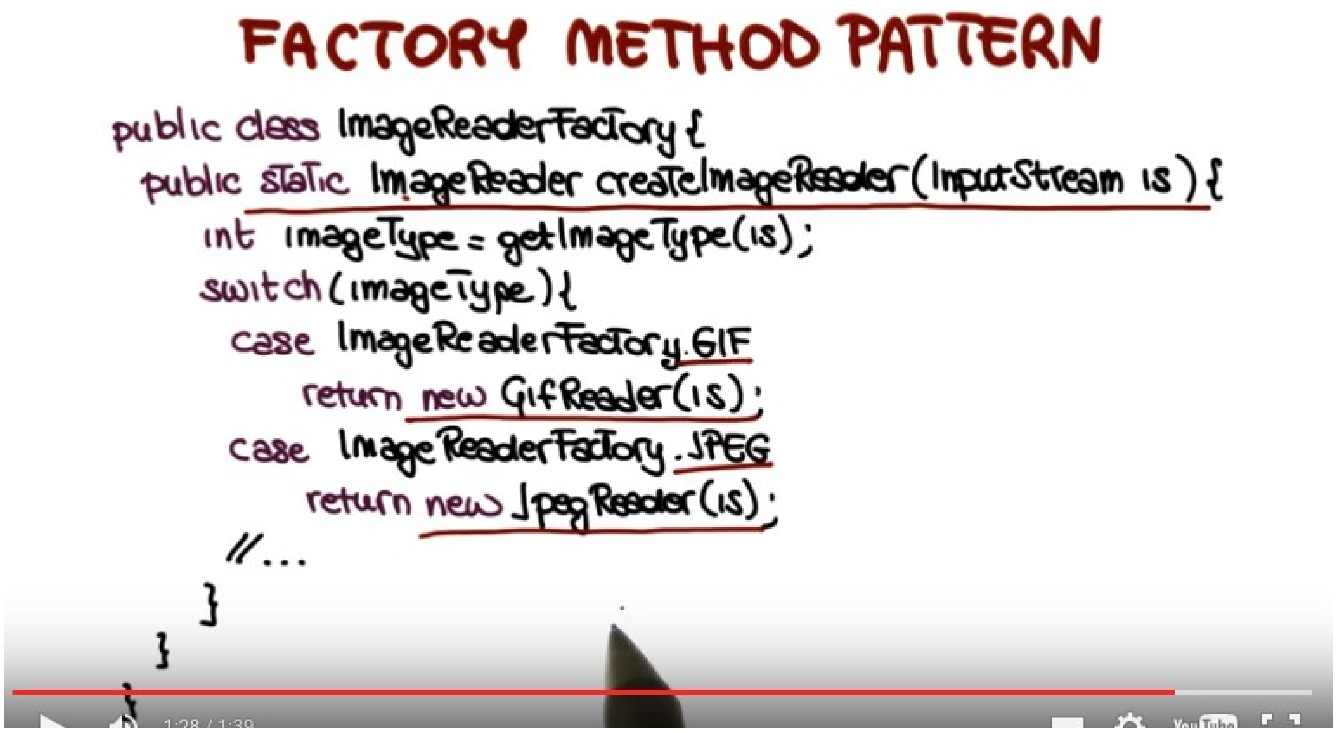
Creator: provides interface for factory method

ConcreteCreator: provides method for creating actual object

Product: object created by the factory method

6. The example I'm going to use consists of a class called ImageReaderFactory which provides the factory method which is this one: createImageReader. As you can see the method takes an InputStream as input and returns an object of type ImageReader, and it's static so that we can invoke it even if we don't have an instance of the ImageReaderFactory. So what does the method do? Well the method first invokes, getImageType, passing the InputStream as a parameter and this method figure out the type of the image that is stored in this Inputstream and it's an integer. Then, based on the value of this integer,

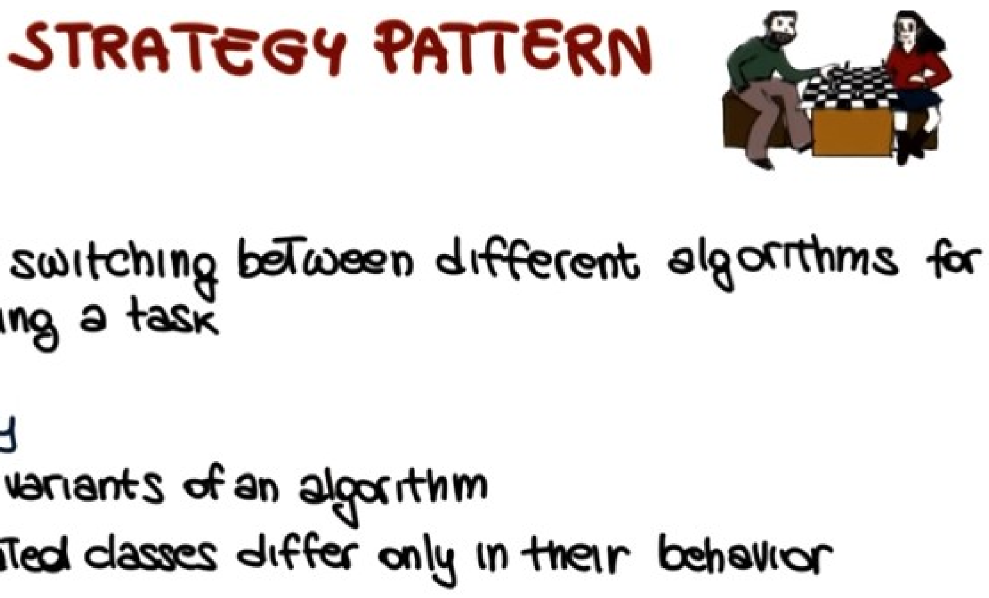
the method does one of several things. If the image type is a GIF, it will invoke the constructor for GifReader passing the stream as a parameter. And what will happen is that the GIF reader will read a GIF from the stream, create a corresponding object and return it. So in this case, the ImageReader object return will be the object representing a GIF as appropriate. Similarly, if the image type is JPEG, then the method will invoke the constructor for JPEG Reader and in this case, this constructor will read from the stream a JPEG, create a corresponding object and return it. And so on for different types of images. So why is this a situation in which it is appropriate to use the factory method pattern? One, because it corresponds exactly to the cases that we saw before, of applicability. This is a case in which we don't know the type of the object that we need to create until we run the code, because it depends on the value of the InputStream. It depends on the content of the InputStream. So, until we read the InputStream, we cannot figure out whether we need to create a GIF, a JPEG or some other type of image. So in this case, we want to do, we want to simply delegate to this classes the creation of the object, once we know what type of object needs to be created. So perfect example of application of a factory method pattern.


FACTORY METHOD PATTERN

```
public class ImageReaderFactory {
  public static ImageReader createImageReader(InputStream is) {
    int imageType = getImageType(is);
    switch(imageType) {
      case ImageReaderFactory.GIF
          return new GifReader(is);
      case ImageReaderFactory.JPEG
          return new JpegReader(is);
      //...
    }
  }
```

7. The second pattern I want to discuss is the strategy pattern, which provides a way to configure a class with one of many behaviors. What does that mean? Well, more precisely, this pattern allows for defining a family of algorithms, encapsulating them into separate classes, so each algorithm in one class, and making these classes interchangeable, but providing a common interface for all the encapsulated algorithms. So in essence, the intent of a strategy pattern is to allow for switching between different algorithms for accomplishing a given task. For example, imagine having different sorting patterns with different space or time tradeoffs. You might want to be able to have them all available and use different ones in different situations. And this pattern is applicable not only when we have different variants of an algorithm, but also when we have many related classes that differ only in their behavior. So let's get more concrete and see how this is done. And I'm going to do it as before, by discussing the structure and the participant for this strategy pattern. In this case, we have 3 types of participants for this pattern, the context, the algorithm, and the concrete strategies. There can be as

many as the number of behaviors that I need to implement. So, let's see what those are. The context is the interface to the outside world. It maintains a reference to the current algorithm and allows for updating this reference at run time. So, basically the outside world will invoke the functionality provided by the different algorithms, by using this interface. And depending on which algorithm is currently selected, that's the one that will be executed when the functionality is involved. The algorithm, also called the strategy, so that's where the pattern gets its name, is the common interface for the different algorithims. So all the algorithms implement this interface. Finally, the concrete strategies are the actual implementations of the algorithms. So if I have 10 different variants of my algorithm, I will implement 10 different concrete strategies. They will all be implementations of this interface.



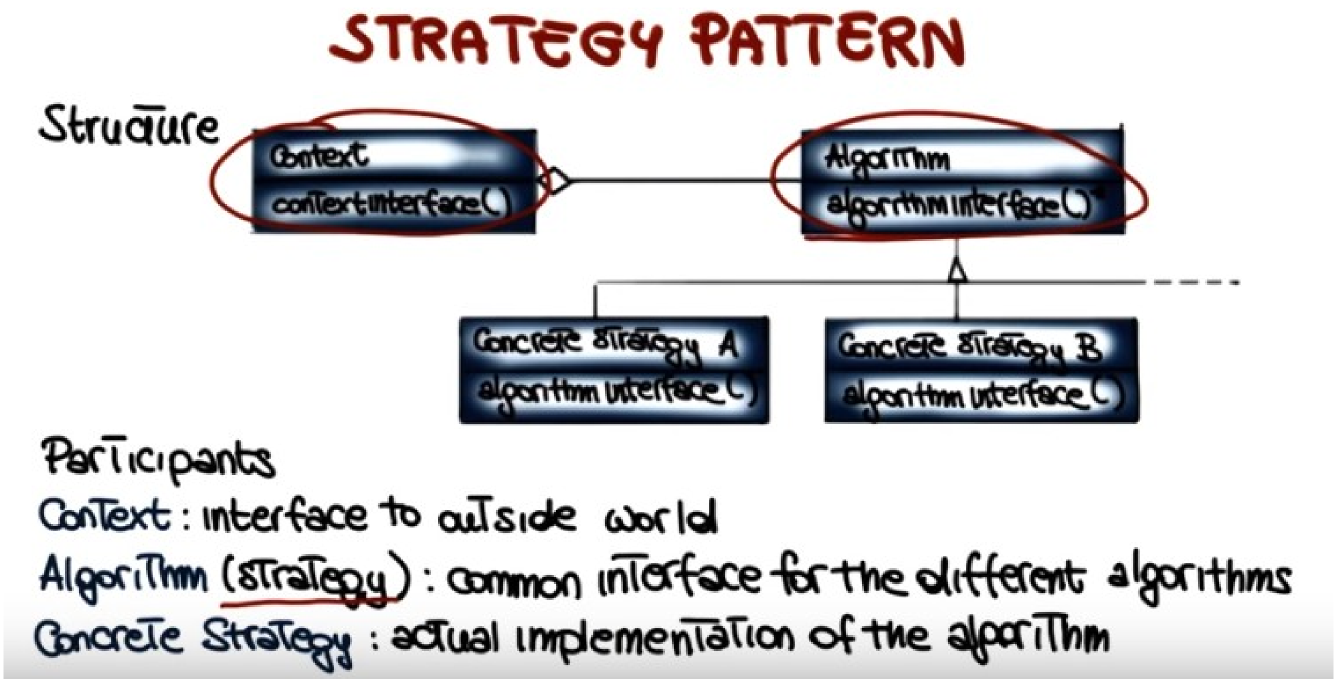
STRATEGY PATTERN

Intent

Allows for switching between different algorithms for accomplishing a task

Applicability

- Different variants of an algorithm
- Many related classes differ only in their behavior



STRATEGY PATTERN

Structure

Participants

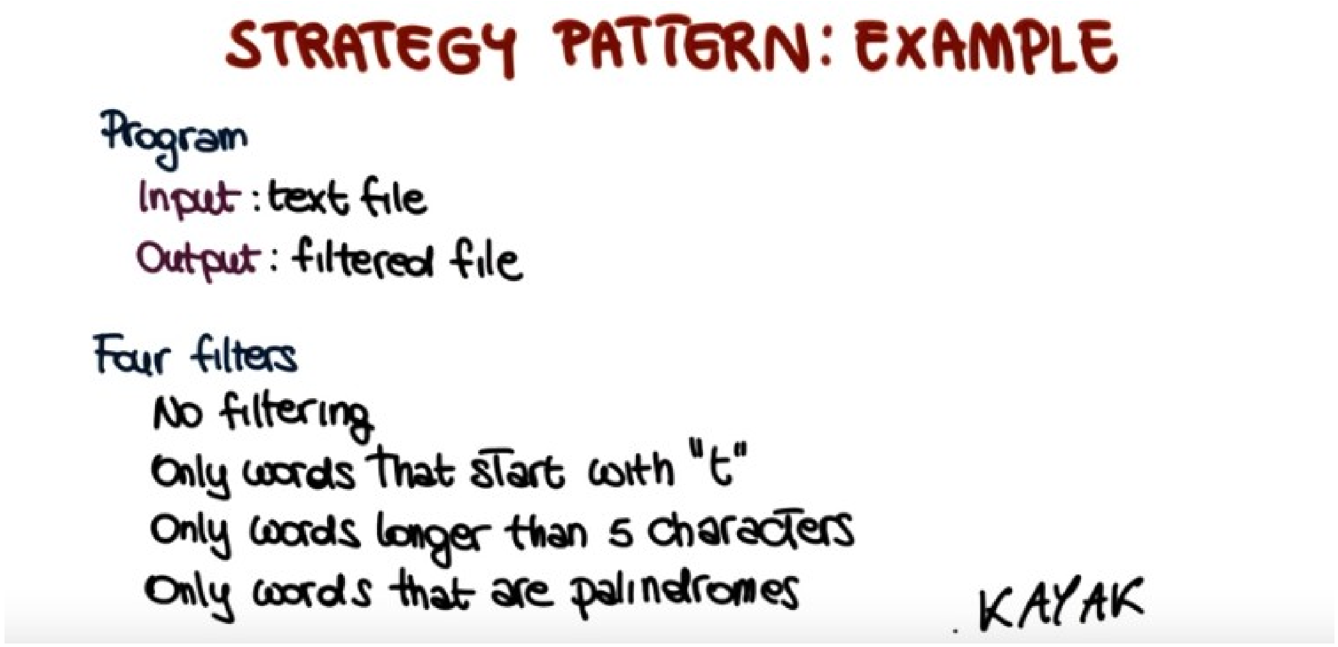
Context: interface to outside world

Algorithm (Strategy): common interface for the different algorithms

Concrete Strategy: actual implementation of the algorithm

8. Now let's see how this whole thing works in practice by using an example. We're going to consider a

program that takes as input a text file and produce it as output, a filtered file. So basically it outputs a subset of the content of this text file based on some filter. And we're going to have a four different types of filters. So the first one is not filtering which means that the whole content of the text file will be produced on the output. The second filter will output only words that starts with t. So you'll take the text file and simply ignore all of the words that do not start with t. So in the output we'll have only those words that starts with letter t. The third filter will producing in the output only words that are longer than five characters. So all the other words will be simply disregarded. And finally, the four filter will produce as output only words in the text file that are palindromes, and in case you don't know what a palindrome is, a palindrome is a word that is the same whether you read it from left to right or from right to left. For example, the word kayak, you can read it in this direction, or in this direction, and it's exactly the same word. So let's see how this program could be implemented using a strategy pattern. And let's do it for real as a demo. What we're looking at here is the editor page for Eclipse, open with the strategy pattern implementation for our example. So what I'm going to do is that, I'm going to look at a different part of implementation. And you will see that, you know, despite the fact that it's slightly longer, it's really fairly simple, it's kind of a straightforward implementation of what we just saw. As I just said, what we are doing is basically building the strategy patterns that allows for changing the strategies with which we're filtering an input file. And we have different strategies, we'll look at those in detail, and we said that the three participants for this pattern are the context, the algorithm, which is the general interface and then the concrete strategies, which are the concrete implementations of this algorithm. So let's start by looking at the context. Which is this class here. And as you can see it contains a reference at the current strategy. We call this the check strategy, which is basically our filter, and when the context is created by default it sets a strategy to the old strategy. The old strategy is the one that accepts all the input, so basically it doesn't filter out anything. And we said that the context is the interface to the outside world, right? So it has to provide the outside world with a way of selecting the strategy, the specific algorithm to be used, and it does that in this case by providing this change strategy method. This method takes a strategy as input, and simply replaces the current strategy with the one specified as a parameter. And at this point, the context also will perform the filtering. The filtering is pretty straightforward, so what it does is that it opens a file that is passed as a parameter so that this the file, the input file to be filtered. And then it reads the file line by line and then split the lines into its composing words and then for each word in each line, what it will do, it will basically invoke the check method in the current strategy, which is basically the filtering method and if the check method returns true, which basically means that the word should be printed, it prints the word. Otherwise, it'll just skip it. So basically the filter will return false for all the words that have to be filtered out. Okay? This is the basic way in which context works. Let's see how this is used in our main method. The main method simply creates the context. Rethink profile from the arguments, and then what he does is simply as a demonstration, it will perform the filtering using all the different filters. So starting from the default one, which is the one that basically doesn't do any filtering that reports all words, then it will switch to the algorithm, that only considers the words that starts with t, and it will do that by invoking a chain strategy and passing this strategy as the argument, and then performing the actual filtering through context. And it will do exactly the same for the strategy that only prints words that are longer than five and the one that only prints words that are palindromes. So now let's look at the actual algorithm. This is the interface, the algorithm interface. And you can see that the only thing that the interface provides is this method, which is the check method, that takes a string as input and will return a bullion. So, basically, it's the bullion that we were seeing before. The one that is true for the words that have to be printed and false for the ones that have to be filtered out. Now, we have all the different implementations of the algorithm, the simplest one is the all algorithm, the simple return is always true, so all the words will be printed. The second one starts with t, and again, without looking at the details of implementations that don't really matter, what it does is basically check in that the first character is t, and returns true in that case and false otherwise. Similarly, for the LongerThan5 algorithm, also in this case, this will implement the check strategy interface, and the check will be performed. By checking that the word is longer than five characters and returning true in that case and false otherwise. And finally the Palindrome check is a little more complicated, but basically it just checks whether the word is a Palindrome and returns true in that case. 'Kay, so as I said, it doesn't really matter too much what is the specific implementations of these matters. What matters is that we have a general interface for the algorithm and then any different concrete implementations of the algorithm that implement different strategies. So again, this allows you to change the behavior of your class without changing class. So we have this context class that does different things when the filter method in involved, depending on what is the current strategy. So the behavior of the class can change dynamically, and it changes dynamically every time that we change the strategy. At this point, the way this whole thing works should be clear, so what we're going to do is that we're going to go to our console, and we're actually going to run the strategy pattern and see what happens. So here I have a file, it's called foo.txt. And if we look at the content of foo, you can see that it's says that this is just a test to assess how well this program performs when used on files of text. And since it checks for palindromes, we will also insert one such word, level. Level is a palindrome, because you can read it from both sides. Okay, so let's see what happens when we run our code. So we're going to run java pattern.strategy.StrategyPattern which is our class, and we going to fetch foo.txt as an input, and let's go back to the beginning of the output to see what happened exactly. You can see here that for the default strategy, which was the old strategy, the whole file is printed, so every word is printed. This is just a test to assess and so on and so forth, 'kay, as expected. For the filter that only prints words that start with t, only words that start with t are printed, again, as expected. Similarly, for the filter that only prints words that are longer than 5, and finally for the one that prints palindromes. And here you can see that we actually have two because the way in which this is implemented we'll also consider single letter words as palindromes because you can read them from both sides. But you definitely will also have level in the output. And in case you want to play with this code yourself, I have made this code and also the implementation for examples of other design partners available as a compressed archive. And the archive is accessible through a URL that is provided in the notes for the cost.



STRATEGY PATTERN: EXAMPLE

Program
Input : text file
Output : filtered file

Four filters
No filtering
Only words that start with "t"
Only words longer than 5 characters
Only words that are palindromes

KAYAK

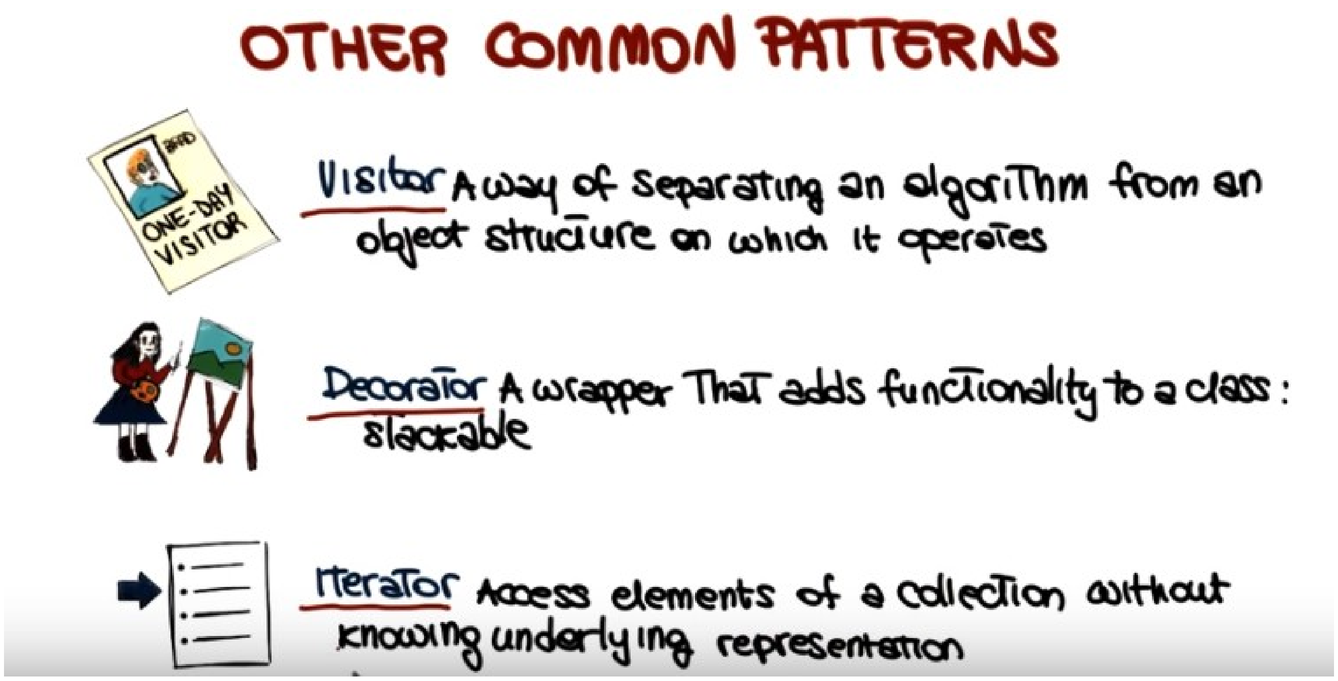
注意以上的各個 filter 之間是互相獨立的, 它們不像我做 collider analysis 時的各個 cuts 那樣疊加, 而是互不相關. 每一個 filter 就是一個 algorithm.

```
package patterns.strategy;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.StringTokenizer;

//Algorithm interface
// CheckStrategy 是一個 interface, 它裡面的 check 函數要在俱體的 algorithm class 中 implement.
check(s)返回 's 是否通過了這個 filter'.
interface CheckStrategy {
   public boolean check(String s);
}

//Algorithm instances
class All implements CheckStrategy {
   @Override
   public boolean check(String s) {
      return true;
   }
}

class StartWithT implements CheckStrategy {
   @Override
   public boolean check(String s) {
      if( s == null || s.length() == 0) {
         return false;
      }
      return s.charAt(0) == 't';
   }
}

class LongerThan5 implements CheckStrategy {
   @Override
   public boolean check(String s) {
      if(s == null) {
         return false;
      }
      return s.length() > 5;
   }
}

class Palindrome implements CheckStrategy {
   @Override
   public boolean check(String s) {
      if(s == null) {
         return false;
      }
```

```
        int length = s.length();
        if(length < 2) {
            return true;
        }
        int half = length/2;
        for(int i = 0; i < half; ++i) {
            if(s.charAt(i) != s.charAt(length - 1 - i)) {
                return false;
            }
        }
        return true;
    }
}

class Context {
    private CheckStrategy strategy;

    public Context() {
        this.strategy = new All();
    }

 //將本 class 中的 strategy 這個 member 設為輸入的 strategy(例如輸入的 strategy 為 new LongerThan5()).
    public void changeStrategy(CheckStrategy strategy) {
        this.strategy = strategy;
    }

    public void filter(String filename) throws IOException {
        BufferedReader infile = new BufferedReader(new FileReader(filename));
        String buffer = null;
        while((buffer = infile.readLine()) != null) {
            StringTokenizer words = new StringTokenizer(buffer);
            while( words.hasMoreTokens() ) {
                String word = words.nextToken();
                if (strategy.check(word)) {
                    System.out.println(word);
                }
            }
        }
    }
}

public class StrategyPattern {

    public static void main(String[] args) throws IOException {
        Context context = new Context();
        String filename = "foo.txt";

        System.out.println("\n* Default:");
        context.filter(filename);
        System.out.println("\n* Longer than 5:");
        context.changeStrategy(new LongerThan5());
        context.filter(filename);

        System.out.println("\n*Palindromes:");
        context.changeStrategy(new Palindrome());
        context.filter(filename);
    }
}
```

9. Before concluding this lesson, let's look at a few more patterns. And although it will take too long to cover them in detail, I would like to at least mention and quickly discuss a few more of these more commonly-used patterns. In fact, some of the patterns that I will discuss, you might have used yourself. Maybe without knowing their name or the fact that they were design patterns. So let's start with a Visitor pattern, which is a way of separating an algorithm from an object structure on which it operates. And a practical result of this separation is the ability to add the new operation to exist in object structures, without modifying the structures. So, basically what this pattern does, is to allow for defining and easily modifying set of operations to perform on the objects of the collection. And the typical usage of this is, for example, when you're visiting a graph, or a set of objects, and you want to perform some operations on these objects. By using a visitor pattern, you can decouple the operation from the objects. Although not straightforward, this pattern is very, very useful. So, I really encourage you to look at it in more detail and get familiar with it. The second pattern I want to mention is the decorator pattern. The decorator pattern is basically a wrapper that adds functionality to a class. So the way in which it works, is that you will take a class, you will build a class that basically wraps this class. So it reproduces the functionality of the original class, but it also adds some functionality. And for all the functionality that was already in the original class, it will simply invoke this functionality and for the new one, you will implement it using the services of the class. And a nice property of the decorator pattern is that it's stackable. So you can add decorators on decorators on decorators, and further increase the functionality provided by your class. The iterator is another very commonly-used pattern. And, you probably use this one yourself because, it's also part of many standard libraries. What the iterator allows you to do, is basically to access elements of a collection without knowing the underlying representation. So the iterator will allow you to just go through a set of objects without worrying about how the objects are stored. So you basically just ask the iterator to give you the first object, the next object and so on. Another very commonly-used pattern is the observer pattern. And this pattern is very useful when you have an object of interest and a set of other objects that are interested in the changes that might occur in this first object. So what the observer pattern allows you to do is to register these objects, so that they let the system know that they're interested in changes in this first object. And then, every time that there is a change, these other objects will be automatically notified. So basically, the observer pattern allows for notifying dependents when an object of interest changes. If you want an example of this, just think about the file system and imagine having a folder. All the views of this folder will want to be notified every time that there's a change in the folder because they need to refresh. So instead of continuously checking the state of the folder, they will just register and basically say, hey, we're interested in knowing when something changes in this folder. And when something changes in the folder, they will be automatically notified. So it will be some sort of a push notification instead of a pull notification, if you are familiar with that terminology. Finally the proxy pattern is a pattern in which a surrogate controls access to an object. In other words, we have our object, and we have our proxy here. So all the requests to the object will go through the proxy. They will then forward them. And all the responses from the object will also go through the proxy. They will then forward

them to the original requester. So what the proxy allows you to do is to control how this object, that is behind the proxy, is actually accessed, for example, by filtering some calls. So in a sense, the proxy allows you for masking some of the functionality of the object that is behind the proxy. And there's many, many, many more useful patterns. That can help you when designing and implementing the system. So once more, I really encourage you to have a look at the book, to look at the resources online, and to really get more familiar with these patterns, and to try to use them in your everyday work.



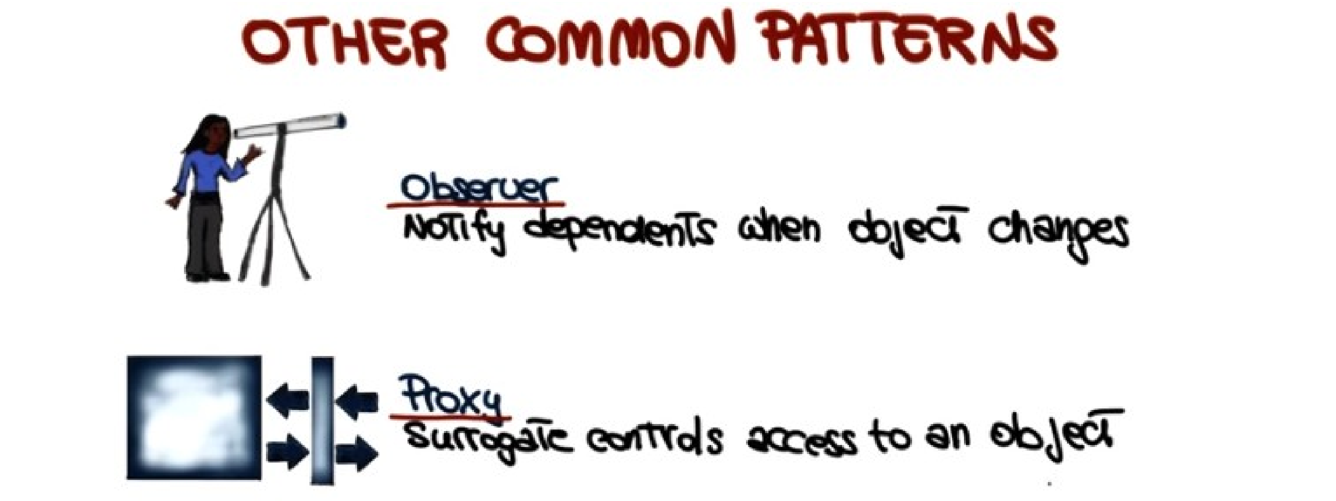
OTHER COMMON PATTERNS

Visitor A way of separating an algorithm from an object structure on which it operates

Decorator A wrapper That adds functionality to a class: stackable

Iterator Access elements of a collection without knowing underlying representation

ONE-DAY VISITOR



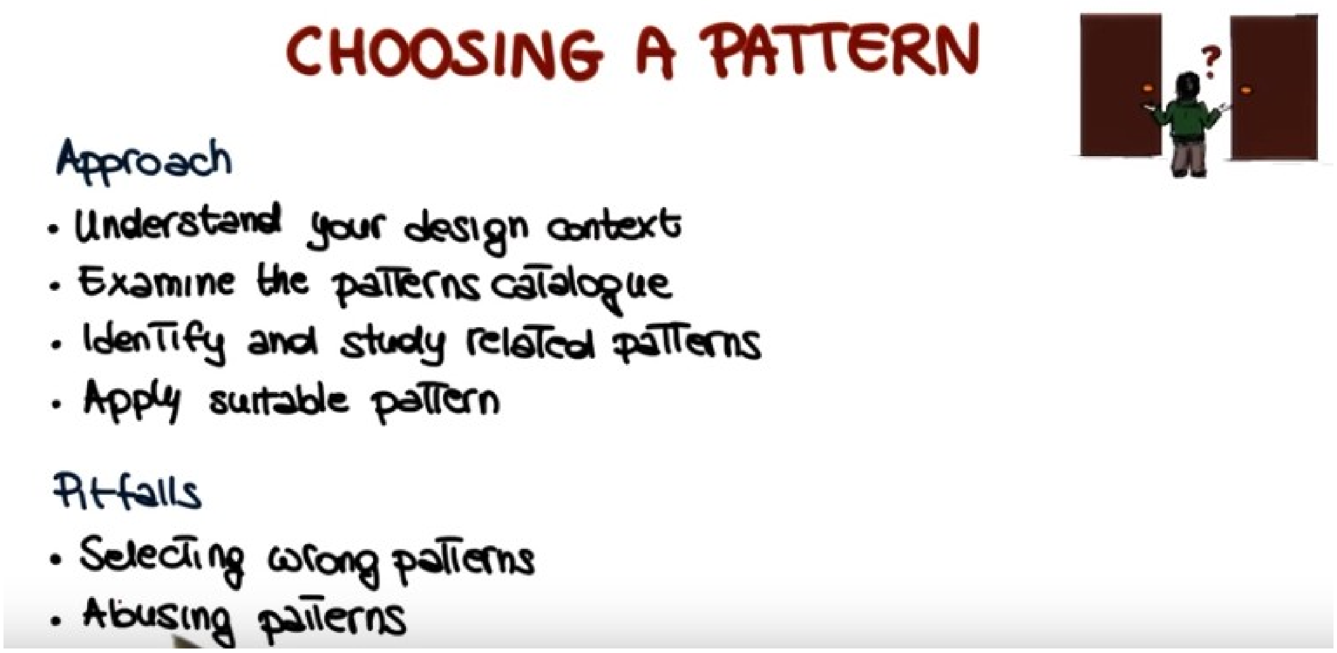
OTHER COMMON PATTERNS

Observer
Notify dependents when object changes

Proxy
Surrogate controls access to an object

10. But with so many patterns, how do we choose a pattern? So this is a possible approach that you can follow. First of all, you want to make sure that you understand your design context. You understand what you're designing and what are the issues involved with this design. With all the problems that you need to solve. At this point, you can examine the patterns catalog, or,if you're already familiar with the catalog, just think about the possible patterns that you could use. Once you identify the patterns that

you can use, you also want to study them and study the related patterns. So normally if you look at any pattern catalog, for each pattern there will also be a list of related patterns. So you can also look at those to see whether maybe some of those might be more applicable. And finally, once you identify the pattern that you think is appropriate, you will apply that pattern. When you do that, just be mindful that there are pitfalls in the use of patterns. One obvious one is the fact that you might select the wrong pattern and make your design worse instead of better. The second one is that if you get too excited about patterns, then you might be abusing patterns, so just using too many patterns, and end up with a design that is more complicated rather than less complicated. So always be careful, spend the time to figure out which one is the right pattern to apply, and make sure that you don't use patterns that you don't actually need.



CHOOSING A PATTERN

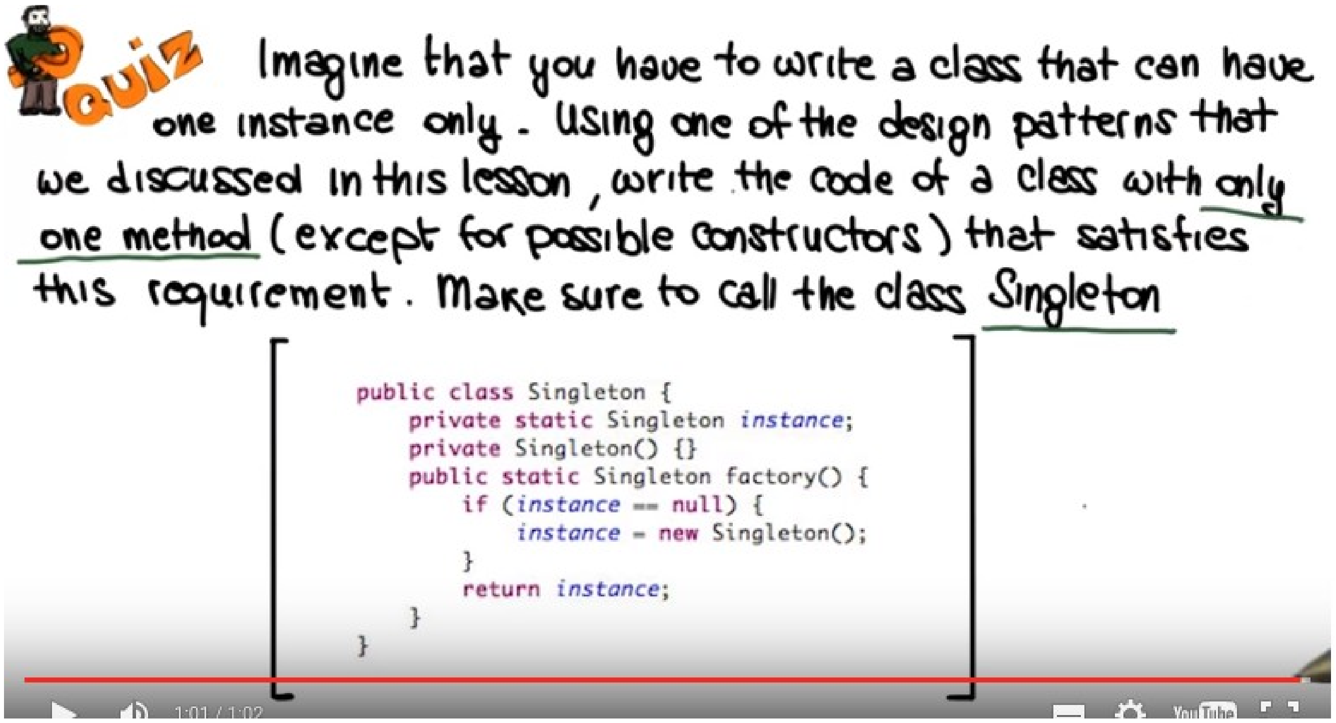
Approach

- Understand your design context
- Examine the patterns catalogue
- Identify and study related patterns
- Apply suitable pattern

Pitfalls

- Selecting wrong patterns
- Abusing patterns

11. Now that we've discussed how to choose a pattern. Imagine that you have to write a class that can have only one instance. So to satisfy this requirement, I would like for you to pick one of the design patterns that we discussed in this lesson, and write the code here that satisfies that requirement. And when you write the code, please make sure that your class has only one method, without counting possible constructors, and that the class is called Singleton. And write your class right here.



QUIZ Imagine that you have to write a class that can have one instance only. Using one of the design patterns that we discussed in this lesson, write the code of a class with only one method (except for possible constructors) that satisfies this requirement. Make sure to call the class Singleton

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static Singleton factory() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

由以上代碼可知, 一個 class 還可以在它裡面定義一個它自己類型的 object 作為 member.

12. As we discussed in the class the right thing to do here was to use the factory pattern. So here is a possible code to solve the problem. Of course that there are different possible solutions. So what we did for this code was to first create a private, static, Singleton object called instance, which is the one that will keep track of the only instance that can be created on the class. Then we define the default constructor, the constructor that doesn't take any parameter as private. In this way other classes cannot create instances of Singleton without calling our factory method, and finally we create the factory method. And the factory method is very simple. The method will first check whether an instance of the class was already created. If it was created, it would just return that instance. Otherwise, it will create a new instance and assign it to that instance member variable and then return the newly created instance. So with this code you're guaranteed that other classes cannot bypass the factory method, because the default constructor is private. And the that the factory method will create one and only one instance of the class, which is exactly what our requirements were.

13. To conclude this lesson, I want to discuss the concept of negative design patterns, that is, patterns that should be avoided. Interestingly, negative patterns were also mentioned in Christopher Alexander's book, so in the first formulation of patterns. So negative design pattern are basically guidelines on how not to do things. In consoles with patterns, the guidelines on how to do things. So basically, what the negative design patterns do is, they enable recurring design defects to be avoided. And as we will see in this class extensively, in mini-course four, negative patterns are also called anti-patterns or bad smells, or bad code smells. So in mini-course four we will see several examples of bad smells and what we can do to eliminate them.

NEGATIVE DESIGN PATTERNS

- Also in Christopher Alexander's book
- How not to (design, manage, etc.)

- Also called anti-patterns and bad smells