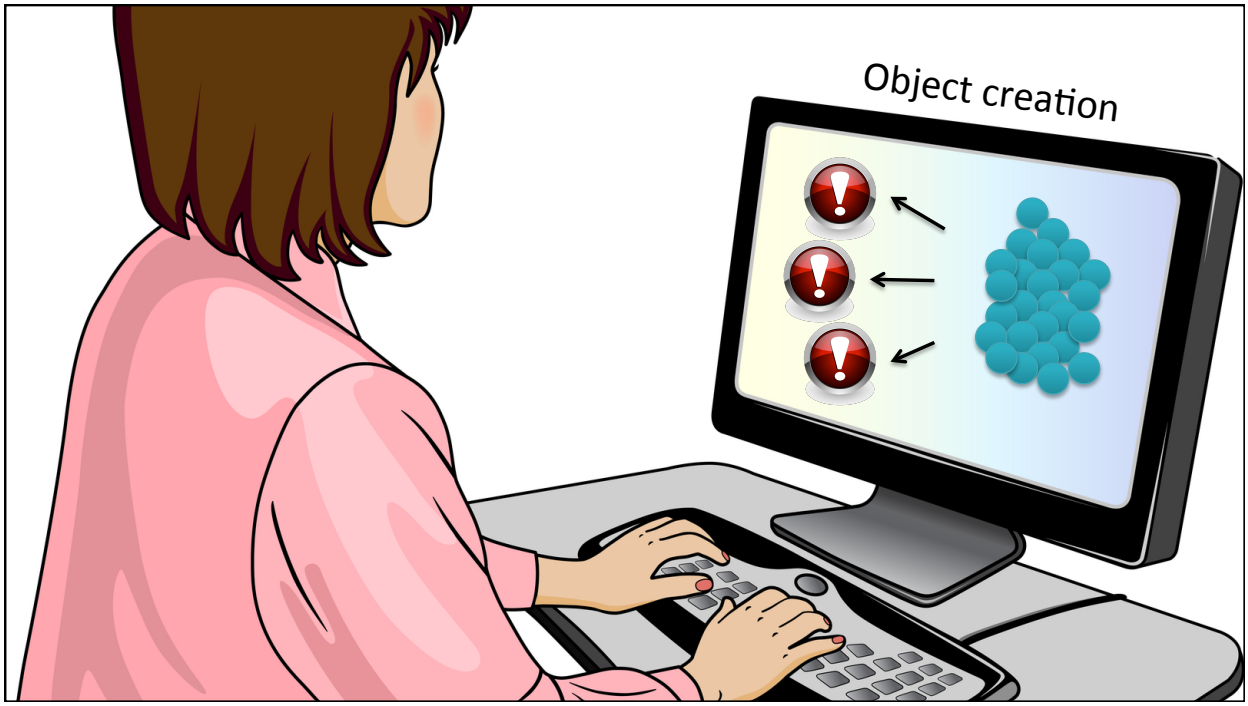
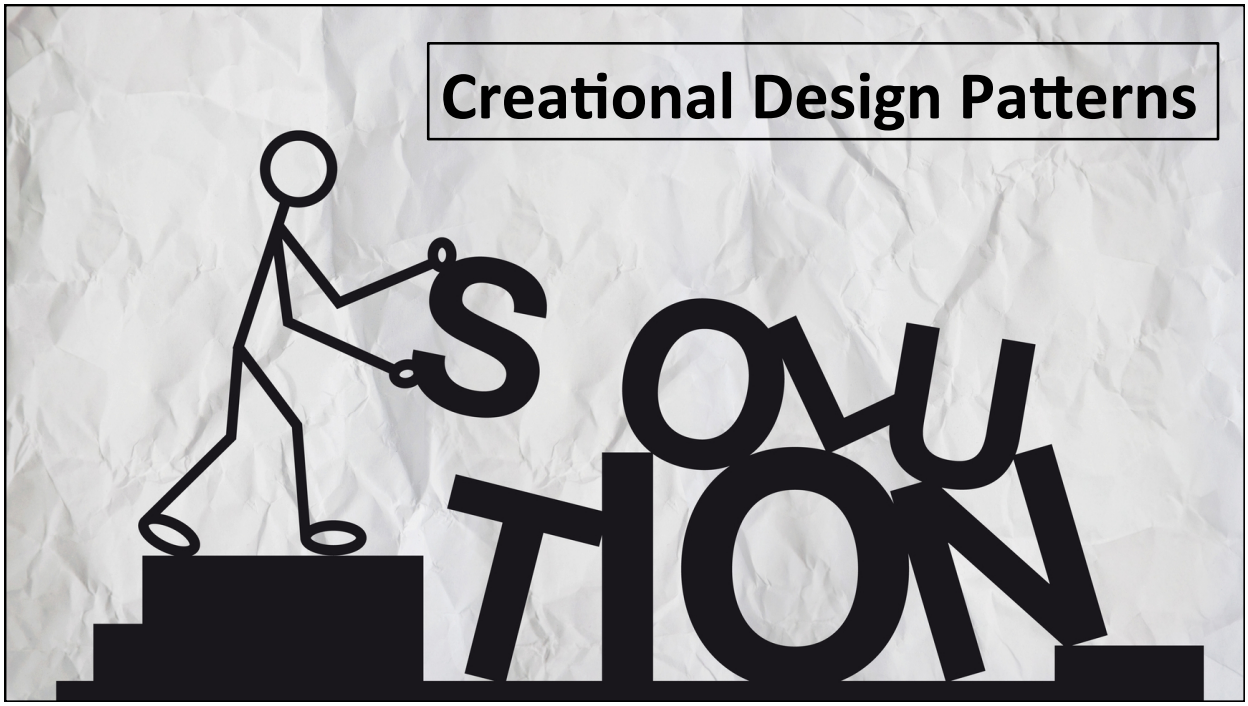

Creational Patterns
Part of the Design Patterns
www.elqoo.com


Object creation


Creational Design Patterns
SOLUTION


Content
Creational Design Patterns
1 Singleton
PAGE 5
www.elqoo.com


Content
Creational Design Patterns
1 Singleton
2 Builder
PAGE 6
www.elqoo.com

Content

Creational Design Patterns

PAGE 7

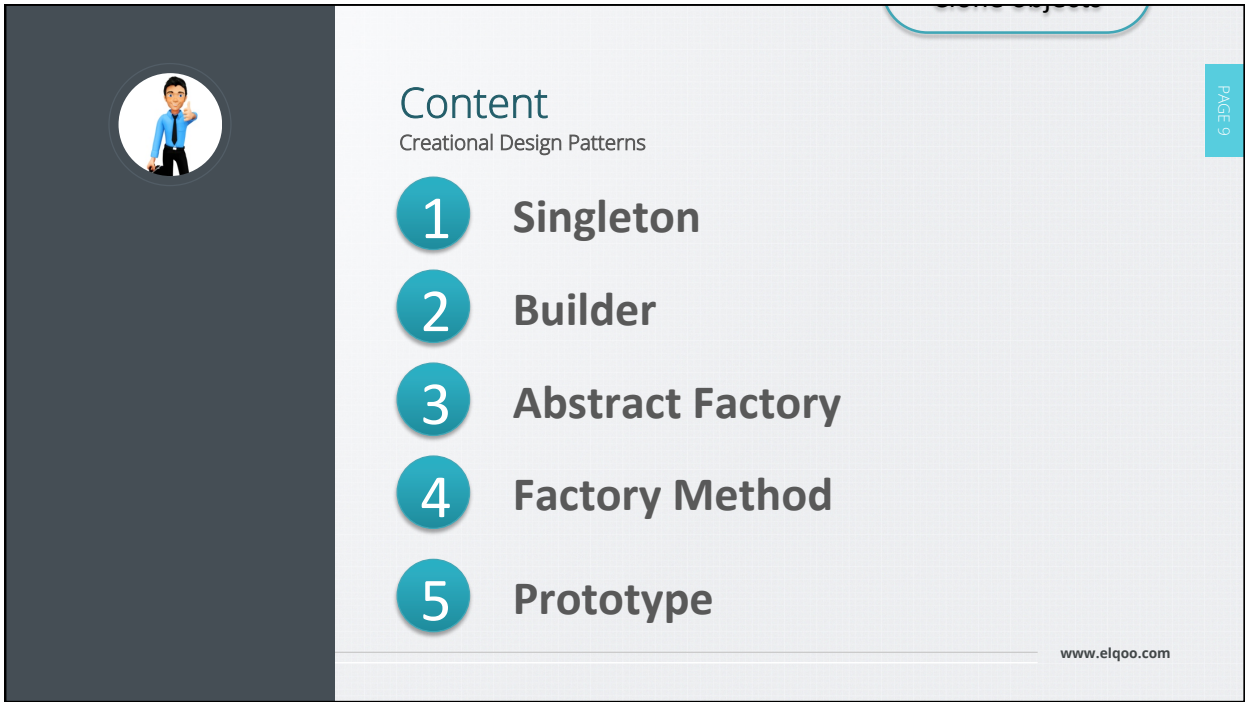
1. Singleton
2. Builder
3. Abstract Factory

Content

Creational Design Patterns

PAGE 8

1. Singleton
2. Builder
3. Abstract Factory
4. Factory Method


Content
Creational Design Patterns
1 Singleton
2 Builder
3 Abstract Factory
4 Factory Method
5 Prototype
PAGE 9
www.elqoo.com


- SINGLETON -
Single object instances

Problem Statement

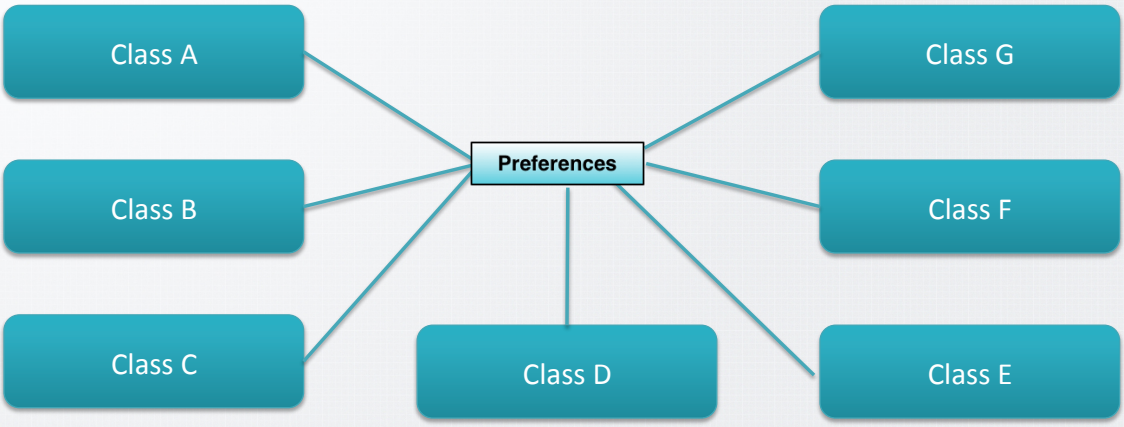
Brad

- Hello Brad
- Hi Suzy
- We would like to manage preferences in different parts of our application
- I'll get on it

Suzy

Problem Statement Overview

Preferences need to be accessible from all over the application



Problem Statement Detail

Detailed explanation of the problem statement

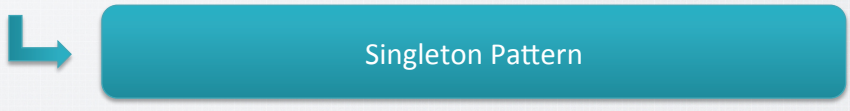
- **Issues**
  - Multiple classes require the same object instance
  - There can **only be one** object for the entire application
  - It must be guaranteed that there is only one object
- **Examples**
  - One FileSystem, Window Manager

Problem Solution

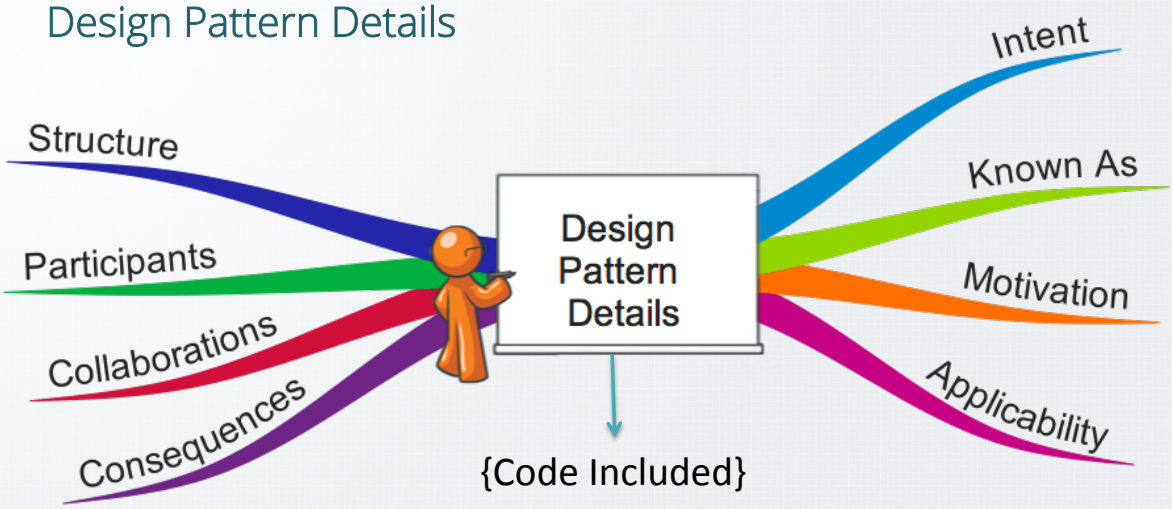
Reason or logic behind the singleton pattern

- **Solution 1**
  - Create **a global variable**

  → Doesn't ensure multiple object creation
- **Solution 2**
  - Let class manage its one instance



Design Pattern Details



Singleton Pattern

Intent and known as

Intent

Ensure a class only has **one instance**, and provide a **global point of access to it**.

Apply Singleton Pattern

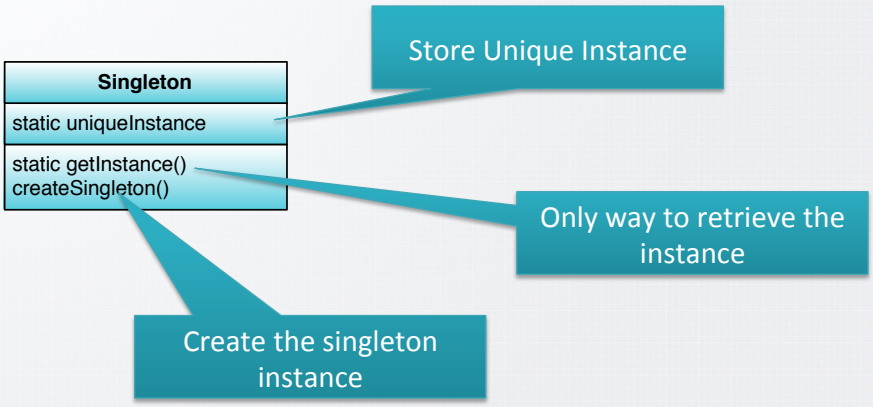
Applicability

- **When**
  - Only **one instance** of class **required**
  - Must be **one access point**
  - Need to **manage object instances**

One instance + One access point

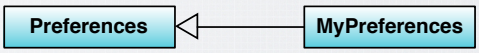
Singleton Pattern Structure

Layout Structure of the singleton pattern



Singleton Pattern Consequences (1)

- **Benefits**
  - **Controlled access** to one instance
  - Reduce name space → Avoids global variables
  - The ability to subclass the singleton class



Singleton Pattern Consequences (2)

- **Benefits**
  - **Configure** the number of **instances** you need
- **Drawbacks**
  - **State** of the singleton must be **shareable** between program executions

Conclusion

- **Singleton pattern is great**
  - Manage **number of instances** at runtime
  - Provide **single** unique access

- Builder -

Building complex objects

Problem Statement

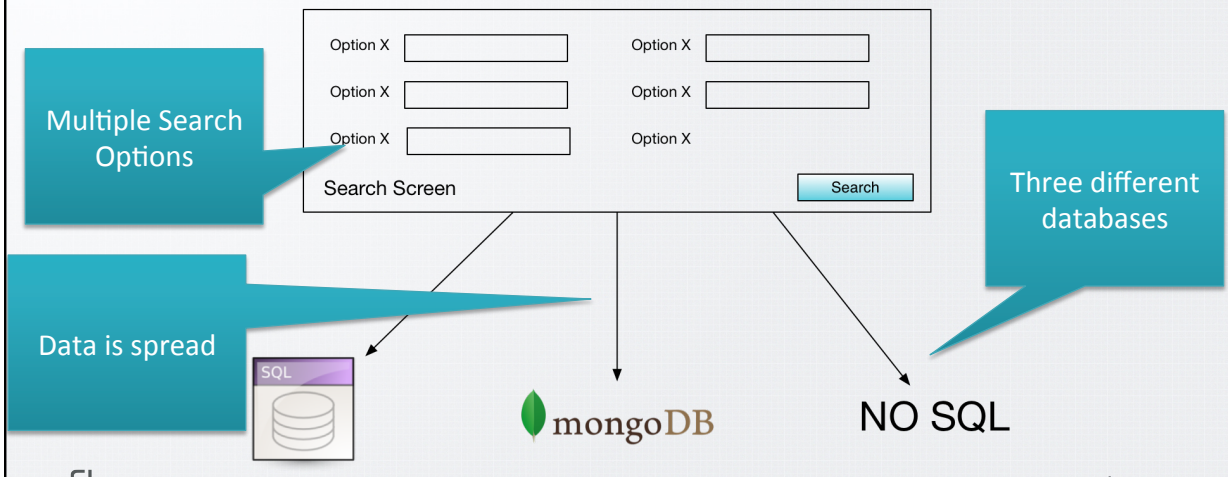
Brad

- Hello Brad
- Hi Suzy
- We need a flexible search screen for our customer application.
- I'll get on it.

Suzy

Problem Statement Overview

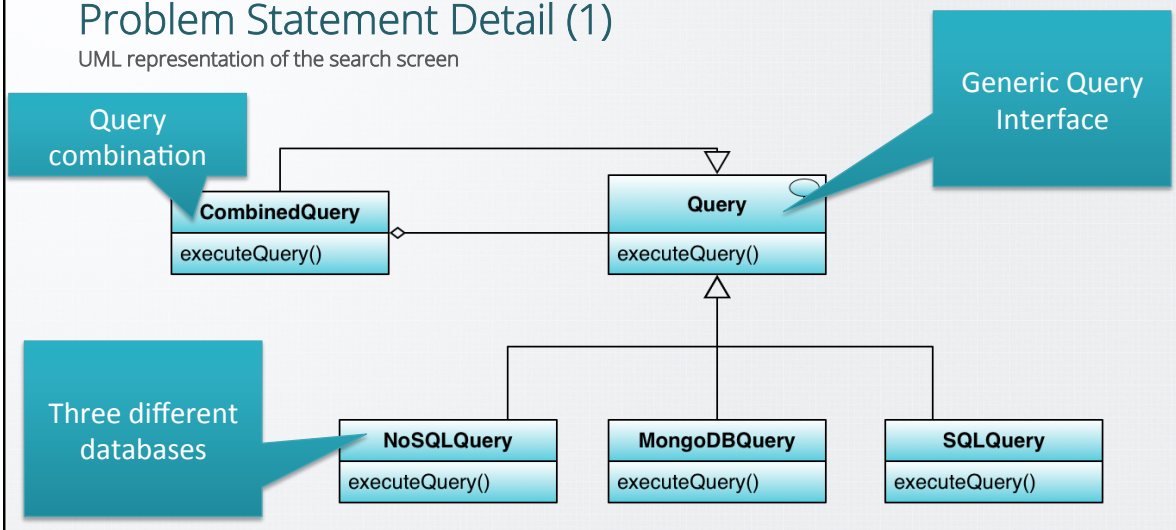
Designing a search screen that connects to multiple data sources



How would you implement the query?

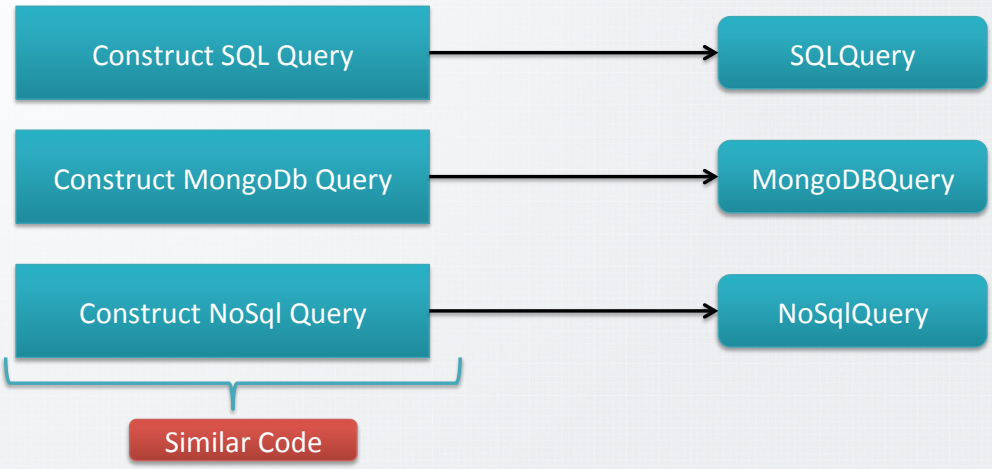
Problem Statement Detail (1)

UML representation of the search screen



Problem Statement Detail (2)

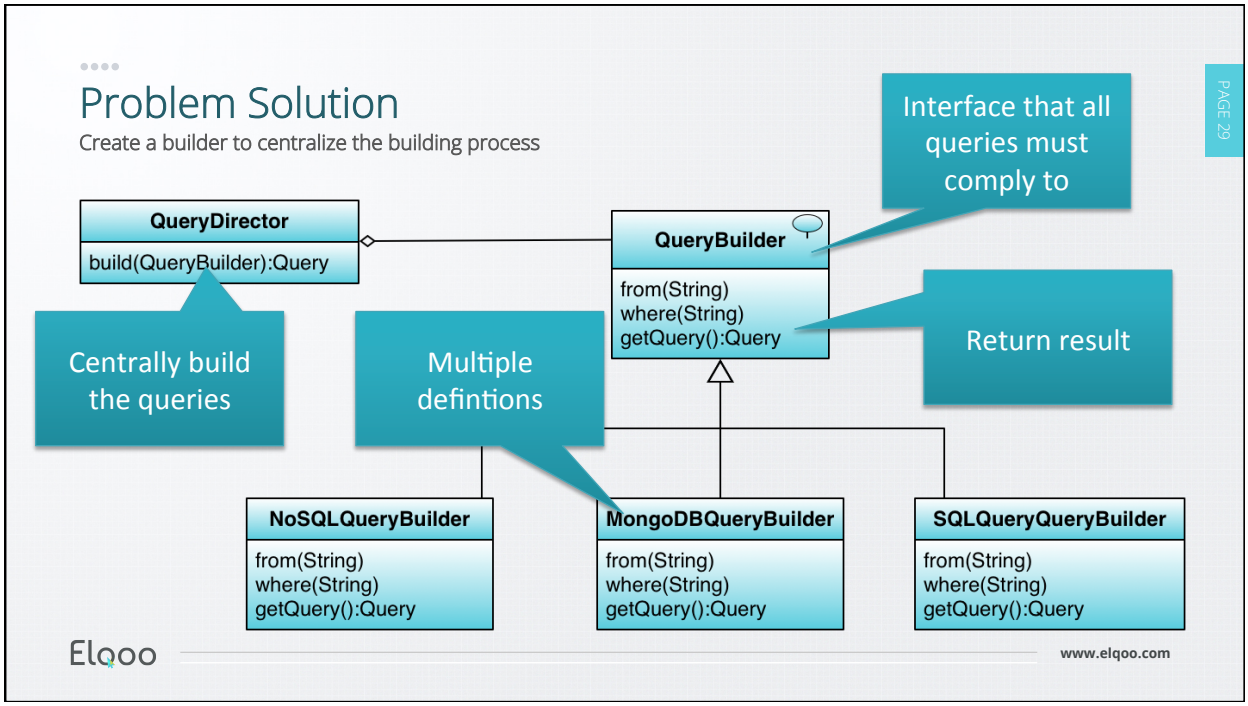
Copy of the construction logic

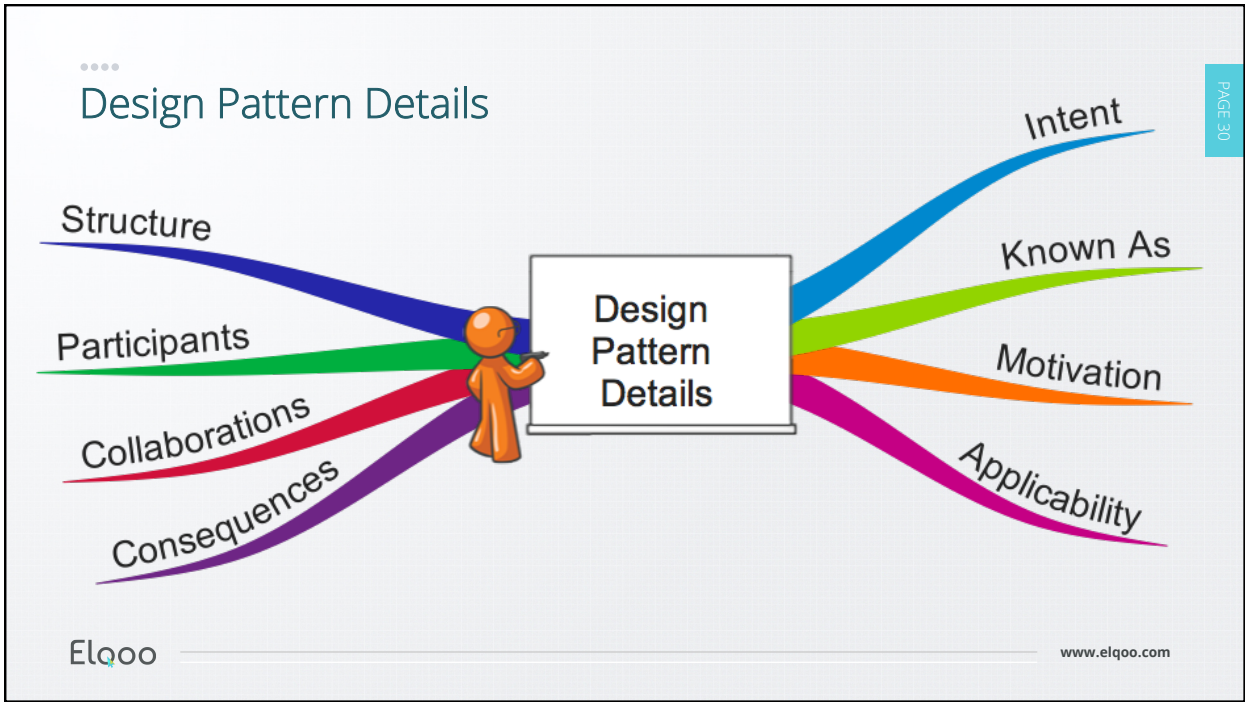


Problem Statement Detail (3)

Detailed explanation of the screen design problems

- **Issues**
  - Search screen results in three objects
    - One for each database
  - Need to **construct** the three objects **over and over again**
- **Future proof**
  - Ideally we would like to **support a fourth database** without changing too much code


Problem Solution
Create a builder to centralize the building process
PAGE 29
Interface that all queries must comply to
QueryDirector
build(QueryBuilder):Query
QueryBuilder
from(String)
where(String)
getQuery():Query
Centrally build the queries
Multiple defintions
Return result
NoSQLQueryBuilder
from(String)
where(String)
getQuery():Query
MongoDBQueryBuilder
from(String)
where(String)
getQuery():Query
SQLQueryQueryBuilder
from(String)
where(String)
getQuery():Query
Elqoo
www.elqoo.com


Design Pattern Details
PAGE 30
Structure
Participants
Collaborations
Consequences
Design Pattern Details
Intent
Known As
Motivation
Applicability
Elqoo
www.elqoo.com

Builder Pattern

Intent and known as

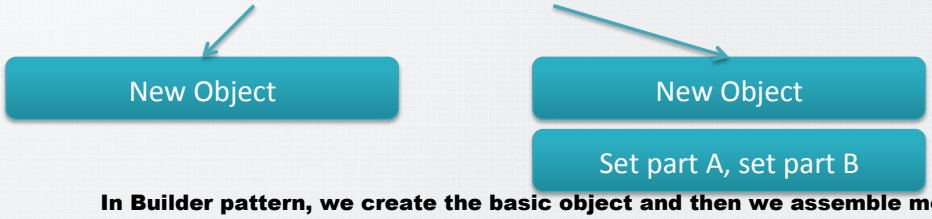
Intent

Separate the **construction** of a **complex object** from its **representation** so that the **same construction** process can create **different representations**.

Apply Builder Pattern

Applicability

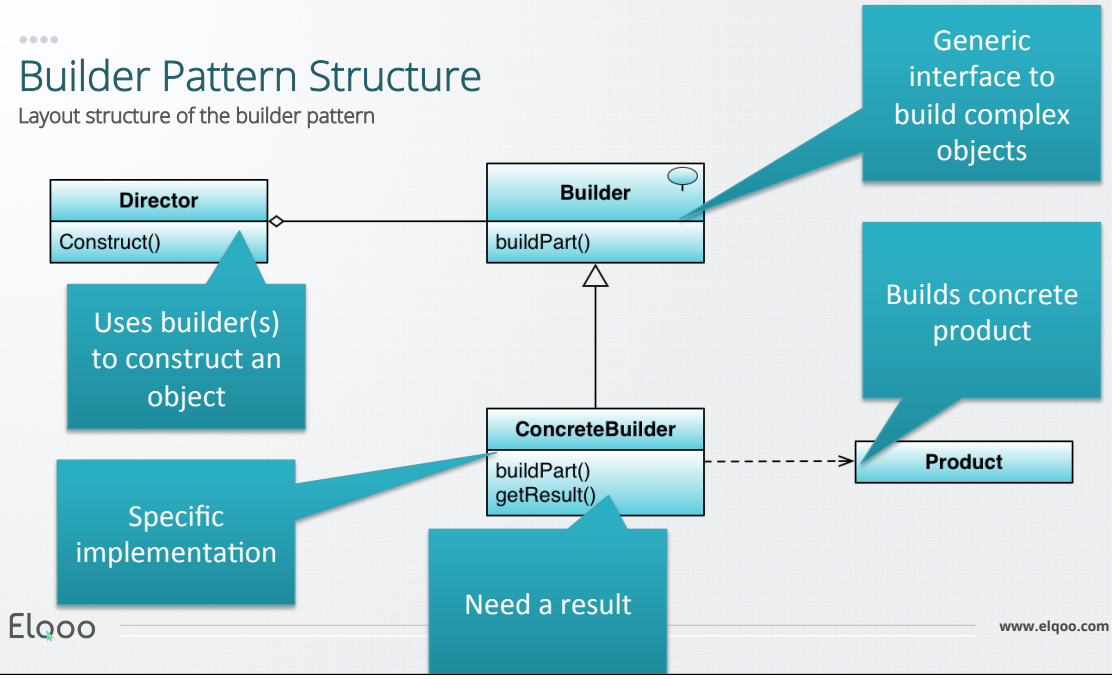
- **Use**
  - **Separate construction** with internal representation
  - One process → **multiple object representation**
  - Object construction <> object assembling



In Builder pattern, we create the basic object and then we assemble more parts on that object.

Builder Pattern Structure

Layout structure of the builder pattern

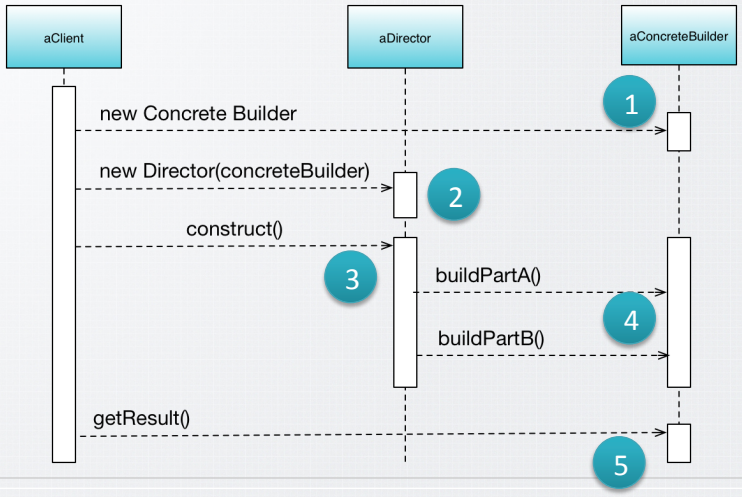


PAGE 33

Elqoo www.elqoo.com

Builder Pattern Collaboration

How objects work with a builder



PAGE 34

Elqoo www.elqoo.com

Builder Pattern Consequences

Benefits of the builder pattern

- **Benefits**
  - **Uniform production** creation via an interface
  - Abstract building process
  - **Loose coupling**
    - Construction
    - Representation
  - **Finer control** on the build process → Allow multiple steps

Conclusion

- **Builder pattern is great**
  - **One build process** for multiple similar objects
  - **Advance control** over the build process

There are three major issues with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

Too Many arguments to pass from client program to the Factory class that can be error prone because most of the time, the type of arguments are same and from client side its hard to maintain the order of the argument.
Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to send as NULL.
If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that is confusing.
We can solve the issues with large number of parameters by providing a constructor with required parameters and then different setter methods to set the optional parameters. The problem with this approach is that the Object state will be inconsistent until unless all the attributes are set explicitly.

Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object.

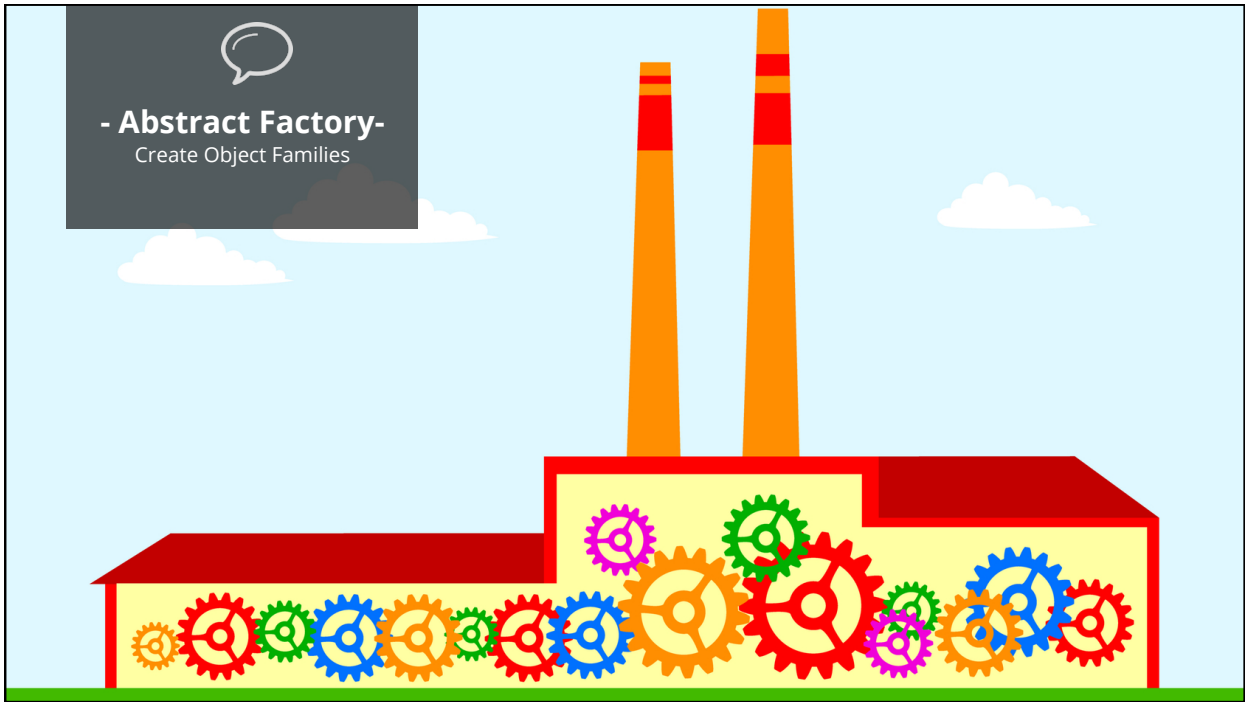
```
package com.syne.Builder;

public abstract class FoodBuilder {
	public abstract FoodBuilder dessert();
	public abstract FoodBuilder edable();
	public abstract FoodBuilder drink();
	protected String dessert;
	protected String drink;
	protected String edable;
	public Food buildFood() {
		// TODO Auto-generated method stub
		Food food = new Food();
		food.setDessert(dessert);
		food.setDrink(drink);
		food.setEdable(edable);
		return food;
	}

}
```

```
package com.syne.Builder;

public class Food {
	private String dessert;
	private String edable;
	private String drink;
	public Food() {
		super();
		// TODO Auto-generated constructor stub
	}
	public String getDessert() {
		return dessert;
	}
	public void setDessert(String dessert) {
		this.dessert = dessert;
	}
	public String getEdable() {
		return edable;
	}
	public void setEdable(String edable) {
		this.edable = edable;
	}
	public String getDrink() {
		return drink;
	}
	public void setDrink(String drink) {
		this.drink = drink;
	}
	@Override
	public String toString() {
		return "Food [dessert=" + dessert + ", edable=" + edable + ", drink="
				+ drink + "]";
	}

}
```

```
package com.syne.Builder;

public class KidFoodBuilder extends FoodBuilder{

	public FoodBuilder dessert() {

		dessert="Coke";
		return this;
	}

	public FoodBuilder edable() {
		// TODO Auto-generated method stub
		edable = "French Fries";
		return this;
	}

	public FoodBuilder drink() {
		// TODO Auto-generated method stub
		drink="coke";
		return this;
	}
}
```

```
package com.syne.Builder;

public class AdultFoodBuilder extends FoodBuilder{

	public FoodBuilder dessert() {

		this.dessert="Sweets";
		return this;
	}

	public FoodBuilder edable() {
		// TODO Auto-generated method stub
		this.edable = "Mutton";
		return this;
	}

	public FoodBuilder drink() {
		// TODO Auto-generated method stub
		drink="alcohol";
		return this;
	}
}
```

```
package com.syne.Builder;

public class WaiterDirector {
	public static void main(String args[])
	{
		FoodBuilder foodBuilder = new AdultFoodBuilder();
		System.out.println(foodBuilder.dessert().drink().buildFood());
	}

}
```



Problem Statement

Brad

- Hello Brad
- Hi Suzy
- We want to use different themes in our application
- No problem, pink too?

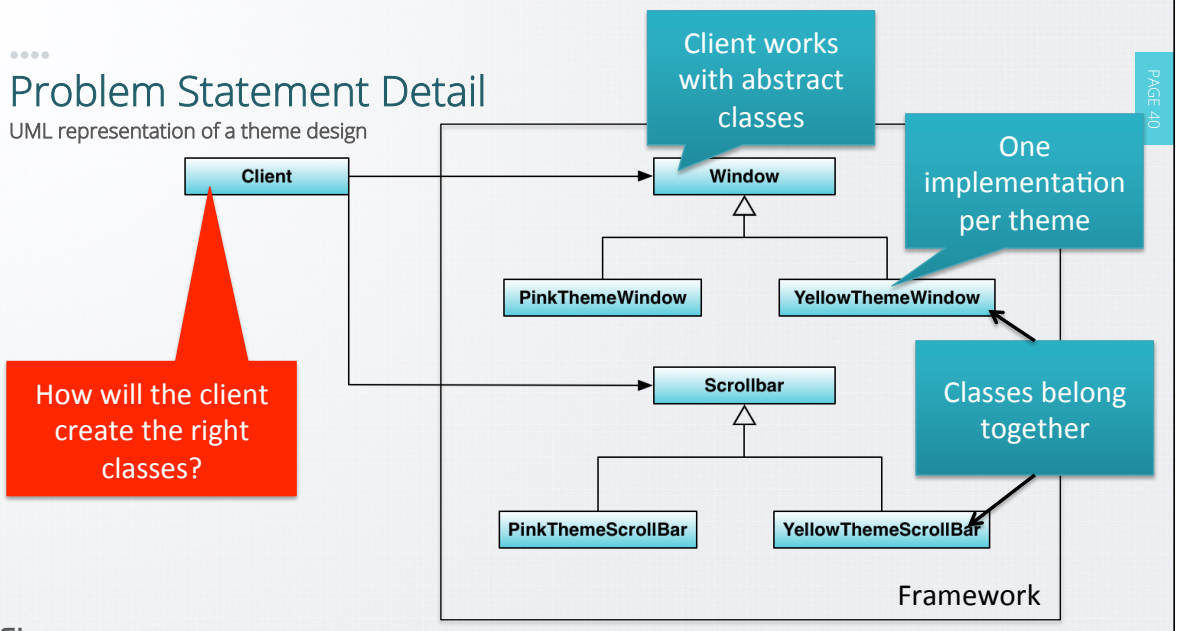
Suzy

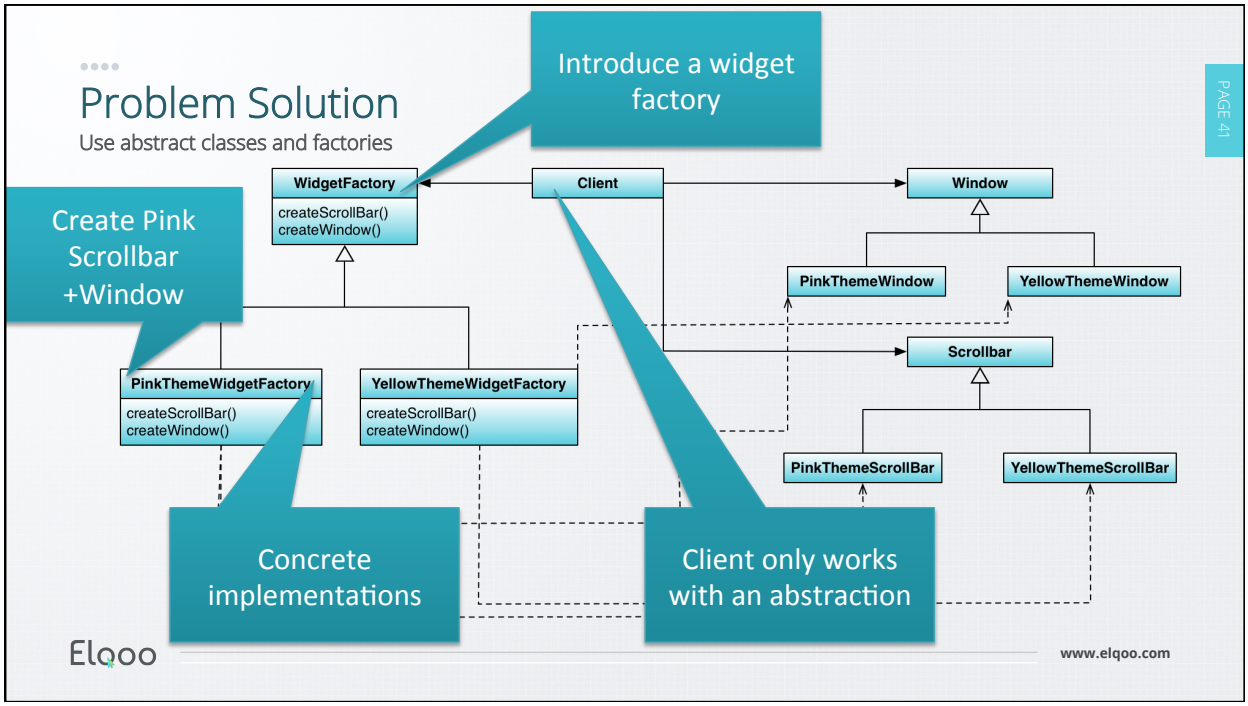
Problem Statement Overview

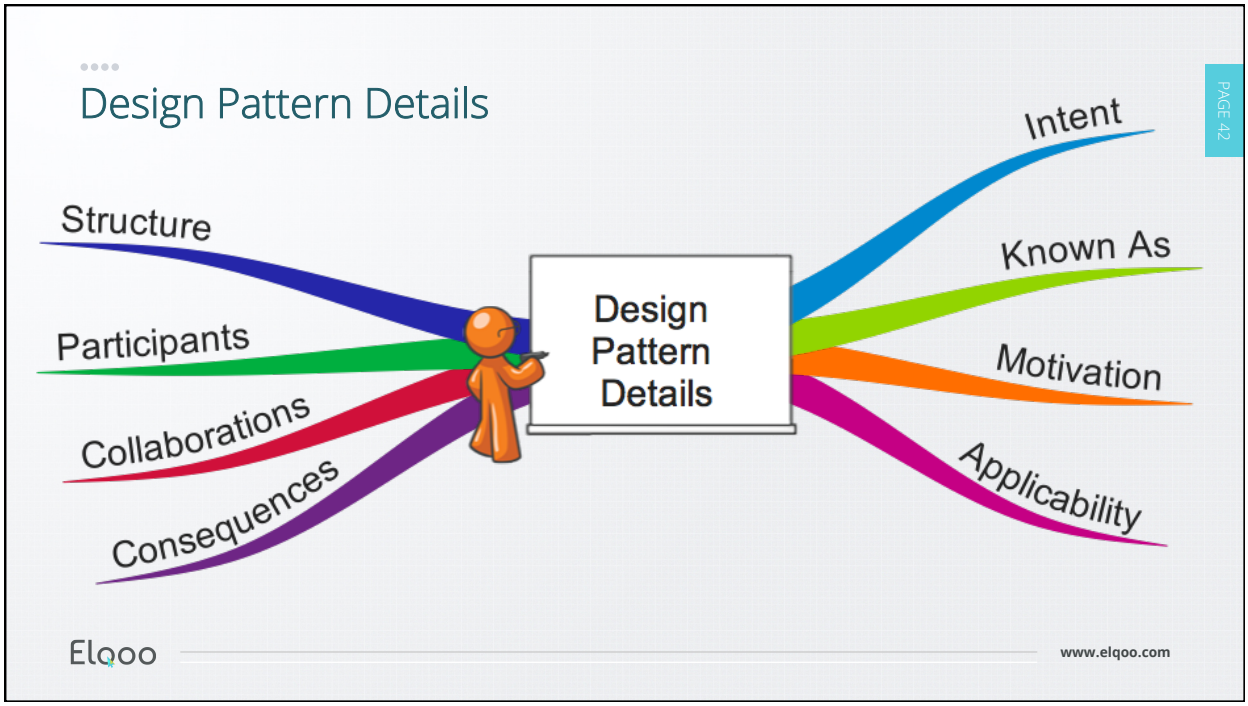
- **Requirement**
  - **Easy change** theme
  - Add new themes
- **Avoid**
  - **Hard coded** themes

Problem Statement Detail

UML representation of a theme design




Problem Solution
Use abstract classes and factories
PAGE 41
Introduce a widget factory
Create Pink Scrollbar +Window
WidgetFactory
createScrollBar()
createWindow()
Client
Window
PinkThemeWindow
YellowThemeWindow
Scrollbar
PinkThemeWidgetFactory
createScrollBar()
createWindow()
YellowThemeWidgetFactory
createScrollBar()
createWindow()
PinkThemeScrollBar
YellowThemeScrollBar
Concrete implementations
Client only works with an abstraction
Elqoo
www.elqoo.com


Design Pattern Details
PAGE 42
Structure
Participants
Collaborations
Consequences
Design Pattern Details
Intent
Known As
Motivation
Applicability
Elqoo
www.elqoo.com

Abstract Factory Pattern

Intent and known as

Intent

Provide an **interface for creating families** of related or dependent objects **without specifying their concrete classes**.

Known As

Kit

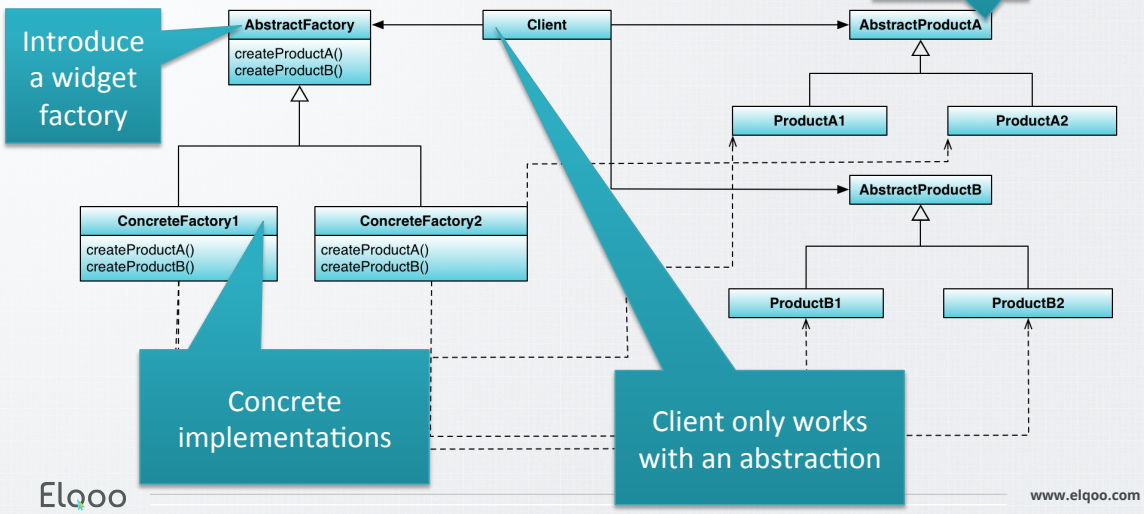
Apply Abstract Factory Pattern

Applicability

- **Use**
  - **Creation** of products **independent** from the **application**
  - Configuration **of product families** is required
  - Hide product implementation → only provide interface

Abstract Factory Pattern Structure

Layout structure of the abstract factory pattern



Abstract Factory Pattern Consequences

Considerations for the abstract factory pattern

- **Benefits**
  - **Control** the classes of object to be created
  - Exchanging product families easy
  - Promote **consistency** among products
- **Drawbacks**
  - **Addition** of **new products** is difficult → extend factory interface

Conclusion

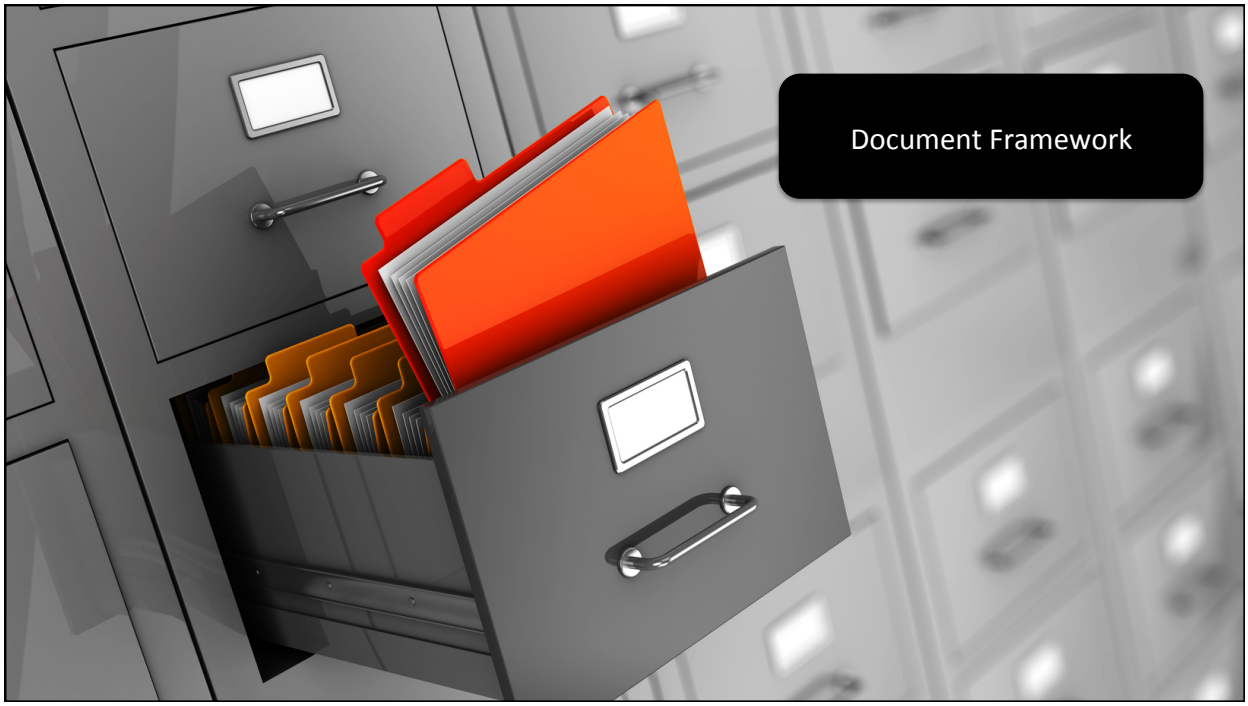
- **Abstract Factory pattern is great**
  - **Creating product families**
  - **Centralize** creation logic

Problem Statement

Brad

- Hello Brad
- Hi Suzy
- Can you help our developers only focus on the new functionality to implement for our document framework?
- No problem

Suzy



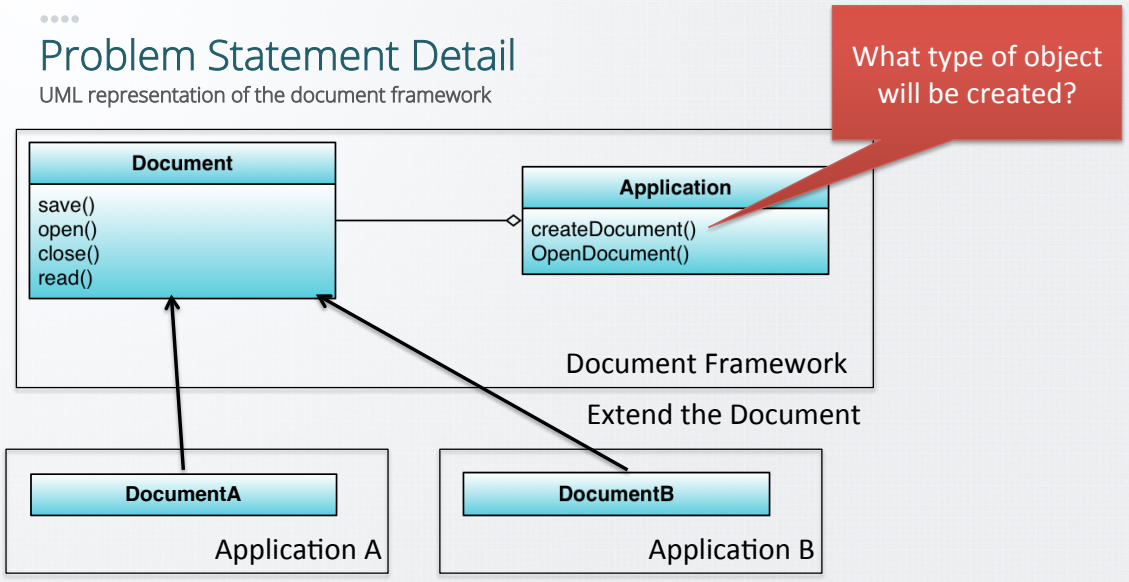
Problem Statement Overview

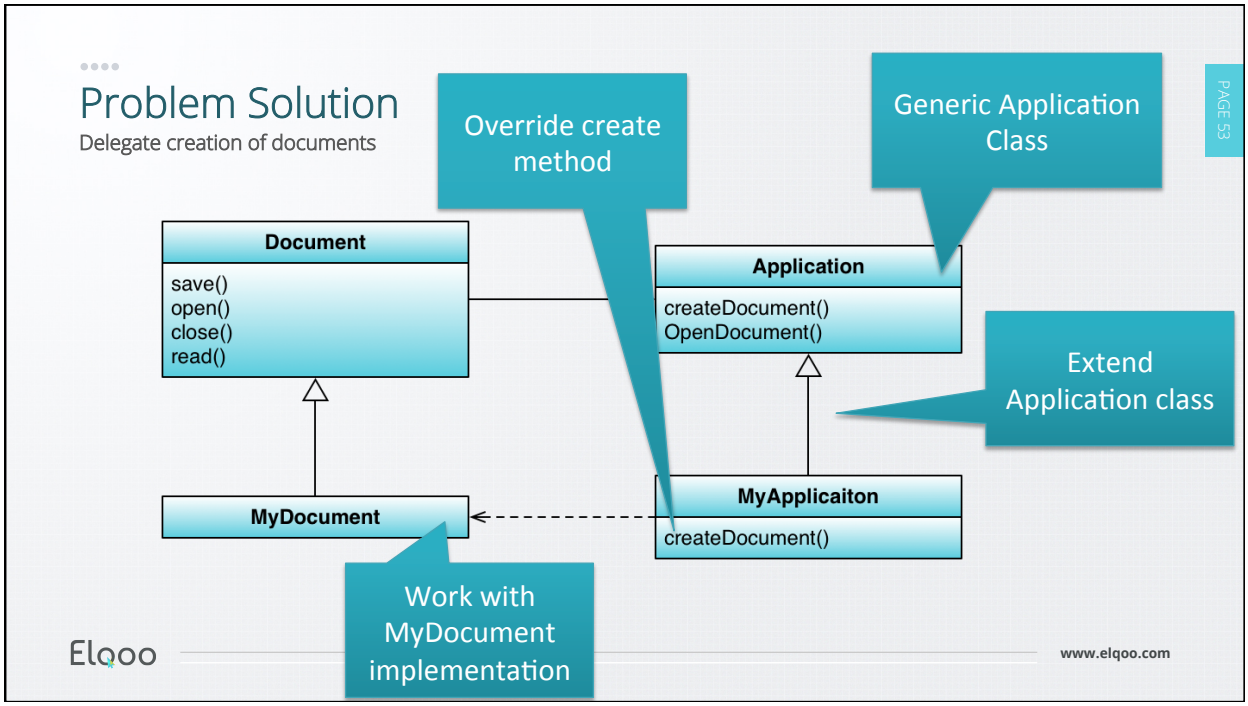
Required functionality for the document framework

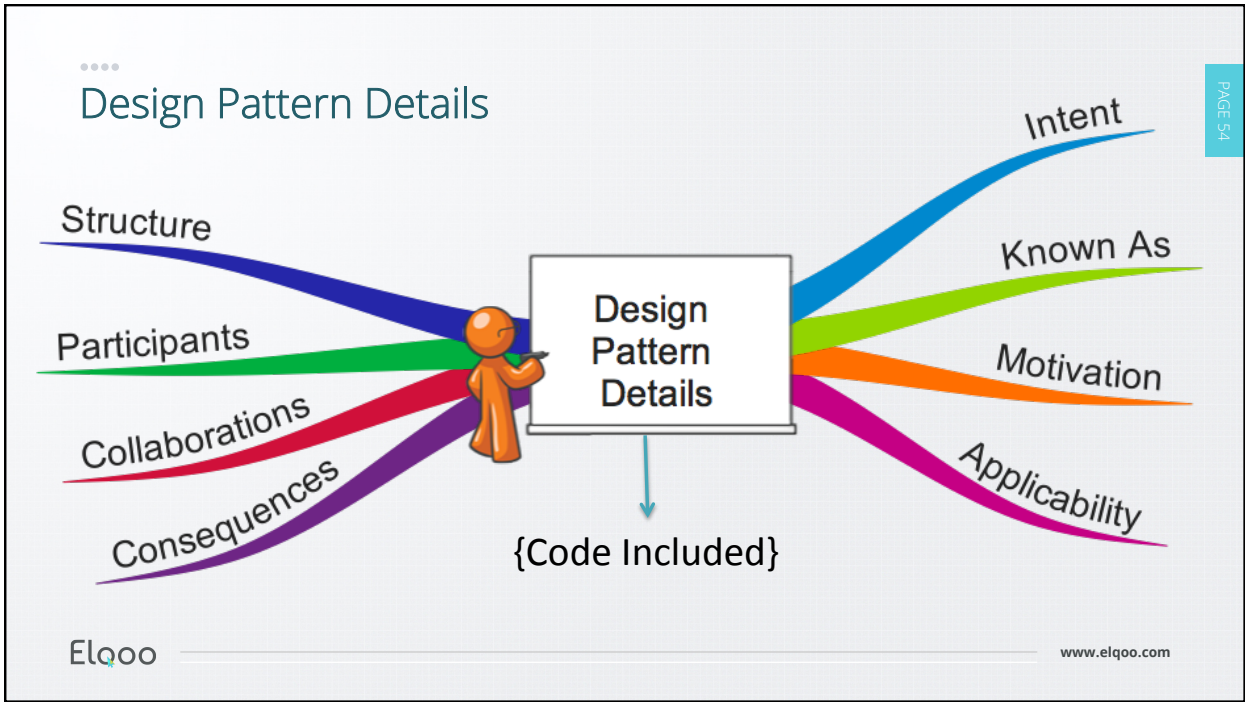
- **Framework Requirement**
  - **Flexible** create new documents
  - **Document** type **per application**
  - Re-use document code

Problem Statement Detail

UML representation of the document framework




Problem Solution
Delegate creation of documents
PAGE 53
Override create method
Generic Application Class
Document
save()
open()
close()
read()
Application
createDocument()
OpenDocument()
Extend Application class
MyDocument
MyApplicaiton
createDocument()
Work with MyDocument implementation
Elqoo
www.elqoo.com


Design Pattern Details
PAGE 54
Intent
Known As
Motivation
Applicability
Structure
Participants
Collaborations
Consequences
Design Pattern Details
{Code Included}
Elqoo
www.elqoo.com

Factory Method Pattern

Intent and known as

Intent

Define an **interface** for **creating an object** but let subclasses decide which class to instantiate. Factory Method lets a class **defer instantiation** to **subclasses**.

Known As

Virtual Constructor

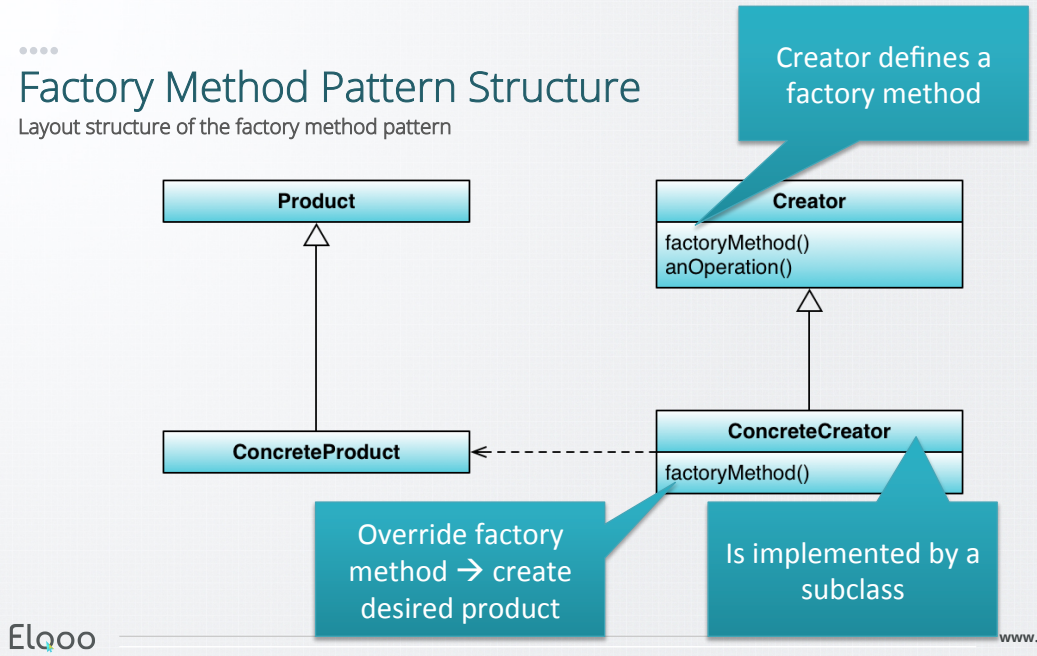
Apply Factory Method Pattern

Applicability

- **When**
  - Class can't **expect** the type of object it must **create**
  - **Subclasses** must decide what types of **objects are created**

Factory Method Pattern Structure

Layout structure of the factory method pattern



Factory Method Pattern Consequences

Considerations for the factory method pattern

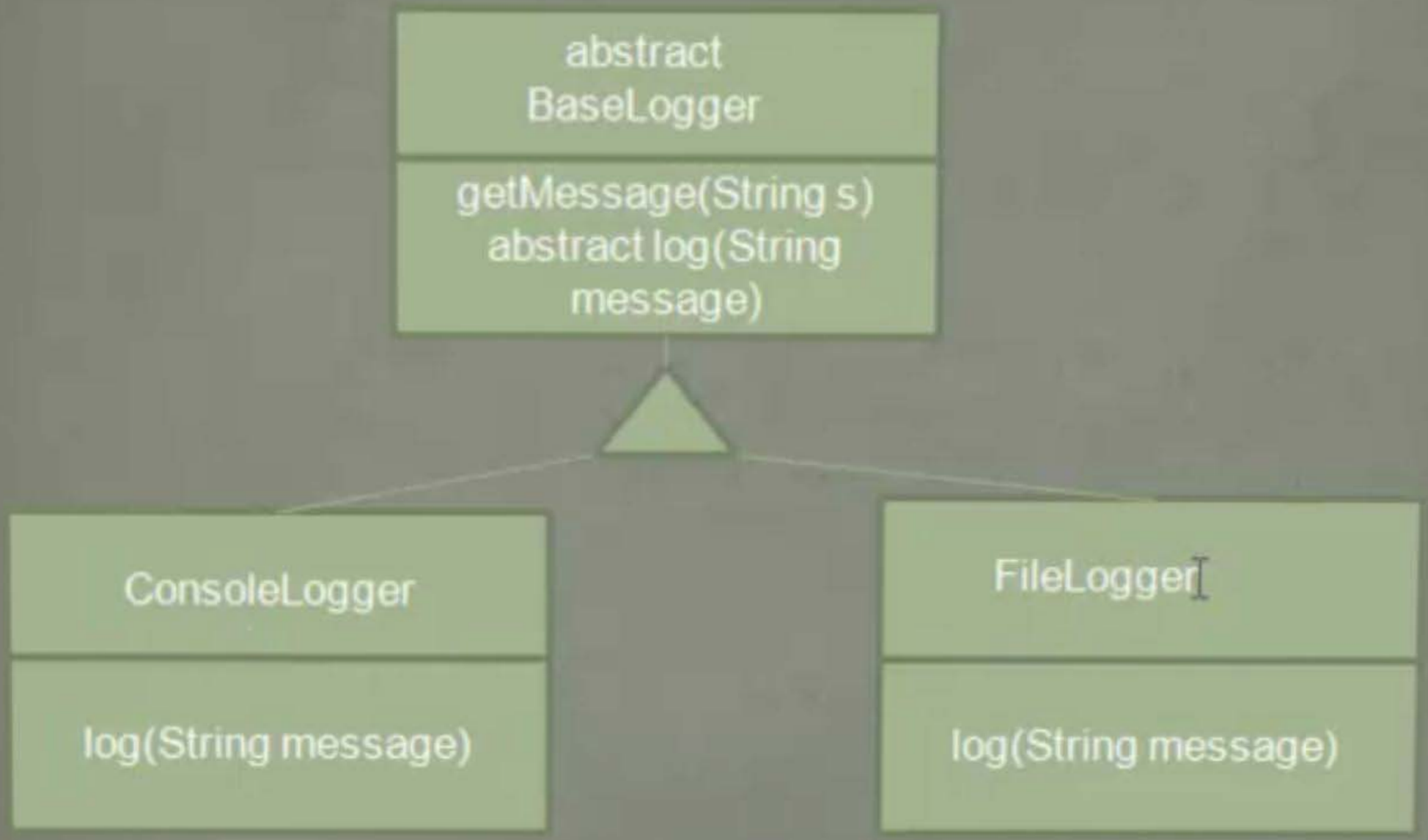
- **Benefits**
  - **Delegate** object creation
  - **Hooks** for subclasses
  - Base class can provide a **default implementation**

Problem definition

- Peter is a software engineer and team member of a team making a new web application using J2EE
- Once the development is complete the application is send to production environment when actual users will interact with the system
- When the development team develops the system , all the exceptions should appear in console while when the end users use the system the exceptions should be logged in a file

it is difficult for the client to remember all the constructors and parameters passed in those constructor for invoking new operator, therefore this approach is not good.

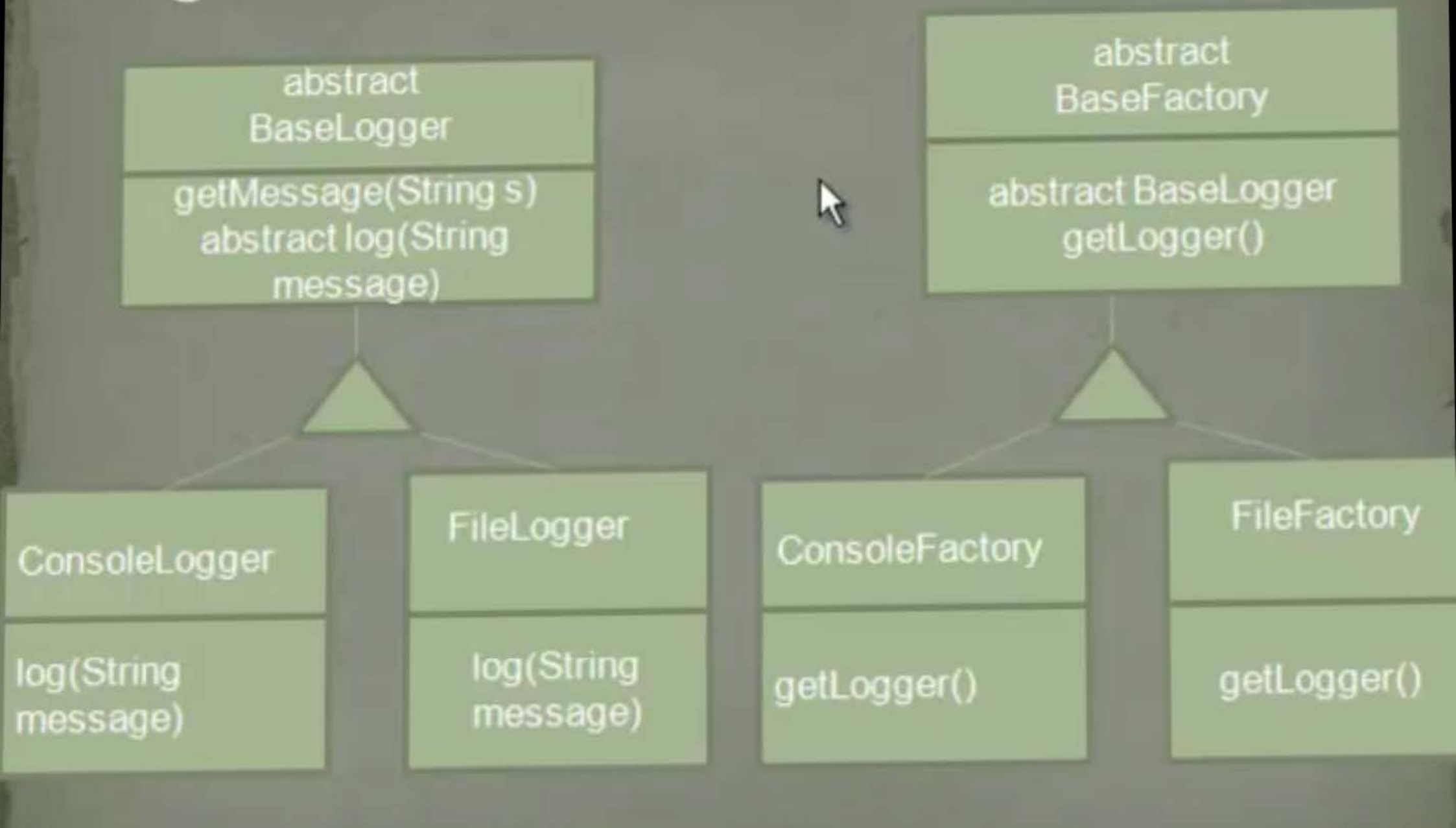
Design Solution 1



Analysis of design 1

- BaseLogger b = new ConsoleLogger();
- BaseLogger g = new FileLogger();
- b.log("Hello");
- g.log("Hello");

Design Solution 2



```
package com.syne.FactoryMethodPattern.product;

public abstract class BaseLogger {
	public String getMsg(String msg)
	{
		return msg;
	}
	public abstract void log(String message);
}
```

```
package com.syne.FactoryMethodPattern.product;

import java.io.PrintStream;

public class ConsoleLogger extends BaseLogger{
	private PrintStream outer;
   public ConsoleLogger(PrintStream stream)//System.out
   {
	outer = stream;
   }
	@Override
	public void log(String message) {
		outer.println("The output is ::"+message);

	}

}
```

```
package com.syne.FactoryMethodPattern.product;

import java.io.PrintStream;

public class ErrorLogger extends BaseLogger{
	private PrintStream outer;
   public ErrorLogger(PrintStream stream)//System.err
   {
	outer = stream;
   }
	@Override
	public void log(String message) {
		outer.println("The error is ::"+message);

	}

}
```

```
package com.syne.FactoryMethodPattern.factory;

import com.syne.FactoryMethodPattern.product.BaseLogger;

public interface BaseFactory<T extends BaseLogger> {
  public  T getLogger();
}
```

```
package com.syne.FactoryMethodPattern.factory;

import com.syne.FactoryMethodPattern.product.ConsoleLogger;

public class ConsoleFactory implements BaseFactory<ConsoleLogger>{

	public ConsoleLogger getLogger() {
		// TODO Auto-generated method stub
		return new ConsoleLogger(System.out);
	}

}
```

```
package com.syne.FactoryMethodPattern.factory;

import com.syne.FactoryMethodPattern.product.ErrorLogger;

public class ErrorFactory implements BaseFactory<ErrorLogger>{

	public ErrorLogger getLogger() {
		// TODO Auto-generated method stub
		return new ErrorLogger(System.err);
	}

}
```

```
package com.syne.FactoryMethodPattern;

import com.syne.FactoryMethodPattern.factory.BaseFactory;
import com.syne.FactoryMethodPattern.factory.ConsoleFactory;
import com.syne.FactoryMethodPattern.factory.ErrorFactory;

public class Intermediate {
	public BaseFactory getFactory(String option)
	{
		if(option.equalsIgnoreCase("out"))
		{
			return new ConsoleFactory();
		}
		else if(option.equalsIgnoreCase("error"))
		{
			return new ErrorFactory();
		}
		return null;
	}
}
```

```
package com.syne.FactoryMethodPattern;

public class App
{
    public static void main( String[] args )
    {
       Intermediate i = new Intermediate();
       i.getFactory("out").getLogger().log("Hi This is an output");
      //  factory.getLogger().log("Hi this is an output");
       i.getFactory("error").getLogger().log("Hi this an error");
    }
}
```

Conclusion

- **Factory Method pattern is great**
  - **Delegate object creation** at runtime
  - Don't know what type of class you need to create

- Prototype-
Create object on prototype

Problem Statement

Brad

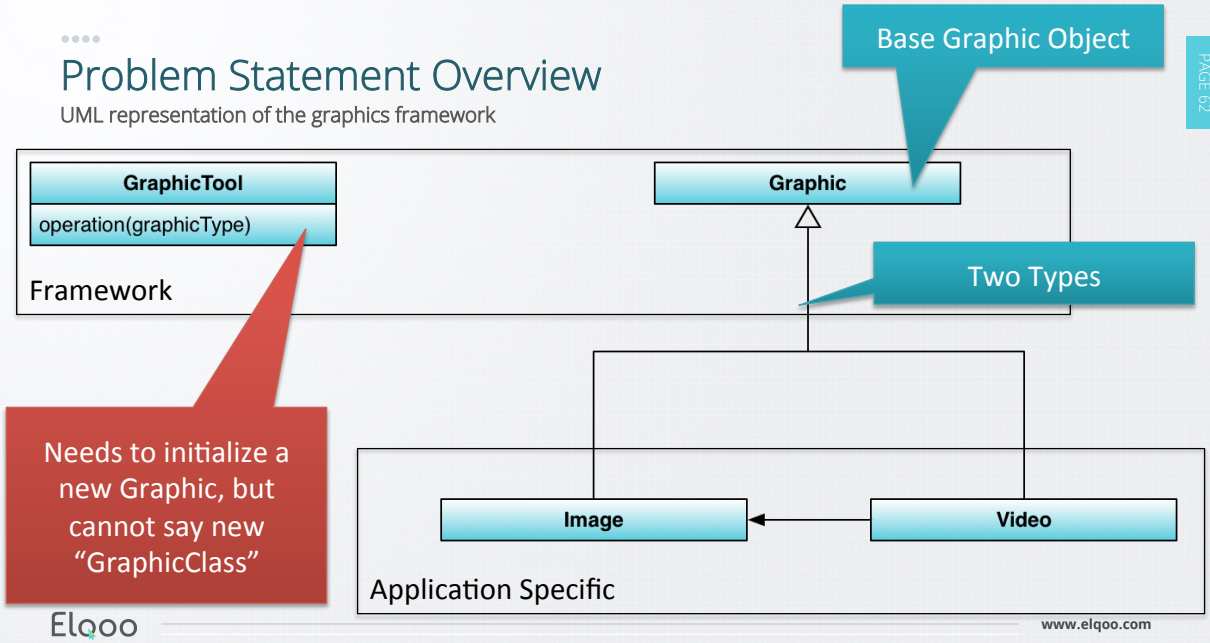
- Hello Brad
- Hi Suzy
- Our Graphic framework isn't as usable as we thought.
- Strange, I thought it was well designed

Suzy

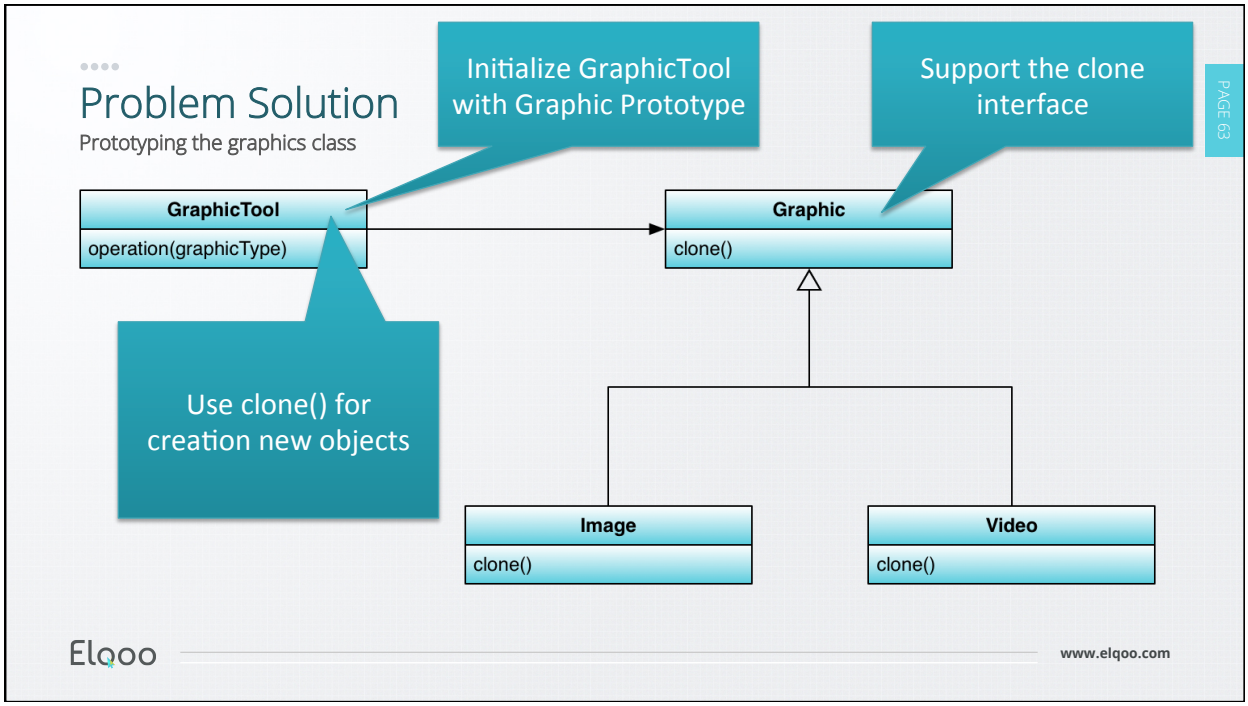
Elqoo www.elqoo.com

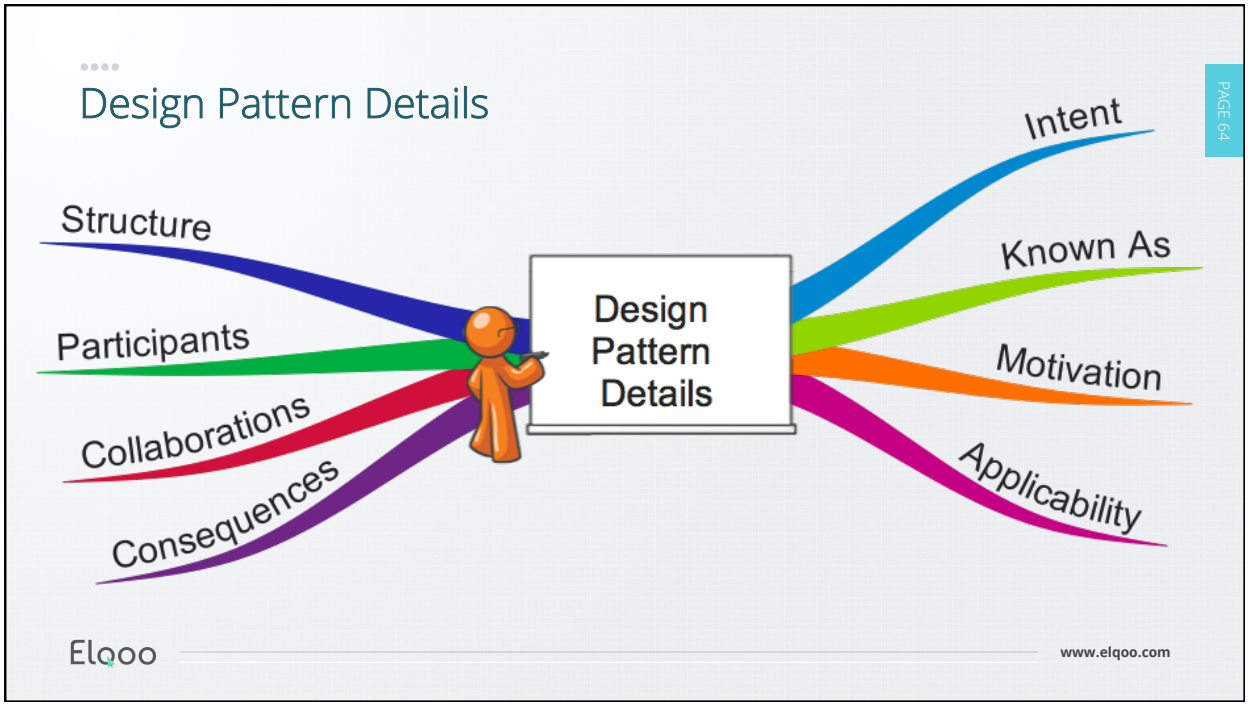
Problem Statement Overview

UML representation of the graphics framework



Elqoo www.elqoo.com


Problem Solution
Prototyping the graphics class
PAGE 63
Initialize GraphicTool with Graphic Prototype
Support the clone interface
GraphicTool
operation(graphicType)
Graphic
clone()
Use clone() for creation new objects
Image
clone()
Video
clone()
Elqoo
www.elqoo.com


Design Pattern Details
PAGE 64
Structure
Participants
Collaborations
Consequences
Design Pattern Details
Intent
Known As
Motivation
Applicability
Elqoo
www.elqoo.com

Prototype Pattern

Intent and known as

Intent

Specify the kinds of objects to create using a **prototypical instance**, and create new objects by **copying this prototype**.

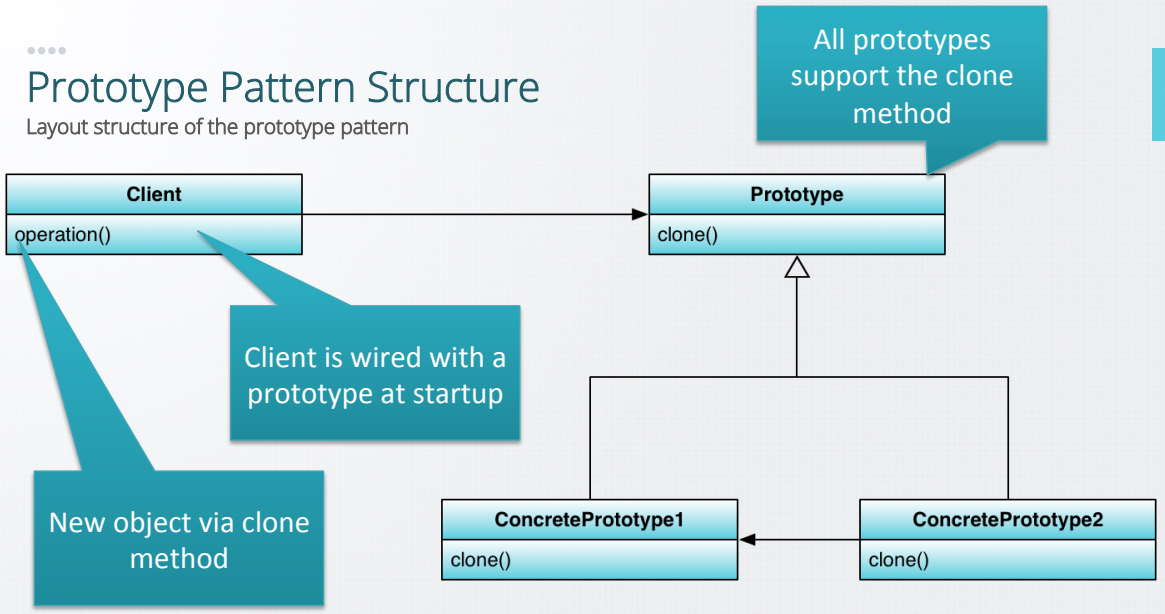
Apply Prototype Pattern

Applicability

- **When**
  - **Classes** to instantiate are specific at **run-time**
  - Avoid building class hierarchies (abstract factory pattern)
  - A class can have **limited instances** of state
    - Cloning is more efficient

Prototype Pattern Structure

Layout structure of the prototype pattern



Prototype Pattern Consequences

Considerations for the prototype pattern

- **Benefits**
  - **Add/remove** products at **runtime**
  - Reduce sub classing (avoid abstract factory)
  - **Configure** application **dynamically**
- **Drawbacks**
  - Requires to create prototypes before other object creation

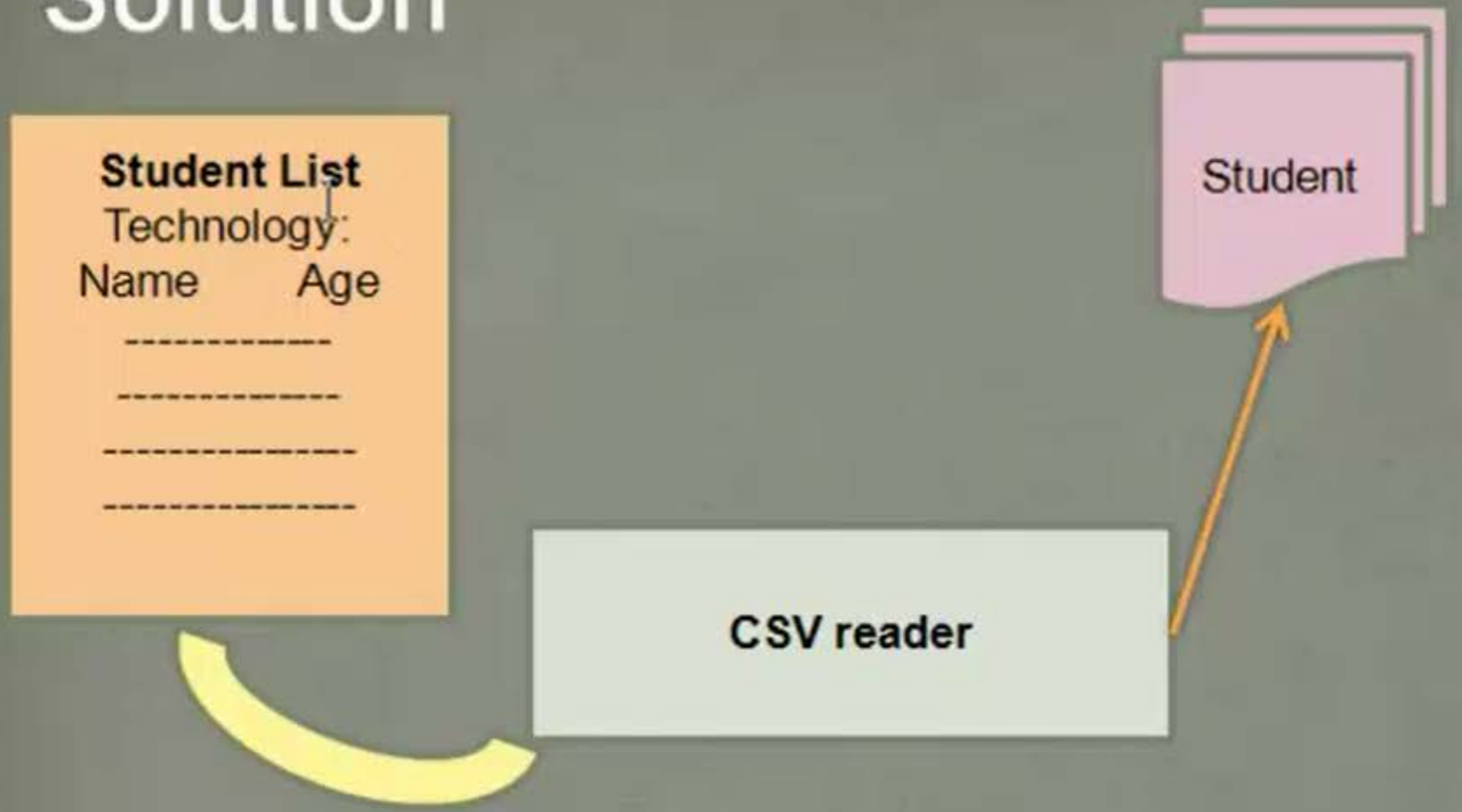
Conclusion

- **Prototype pattern is great**
  - **Configure object creation** by cloning
  - **Dynamically change** object **creation**

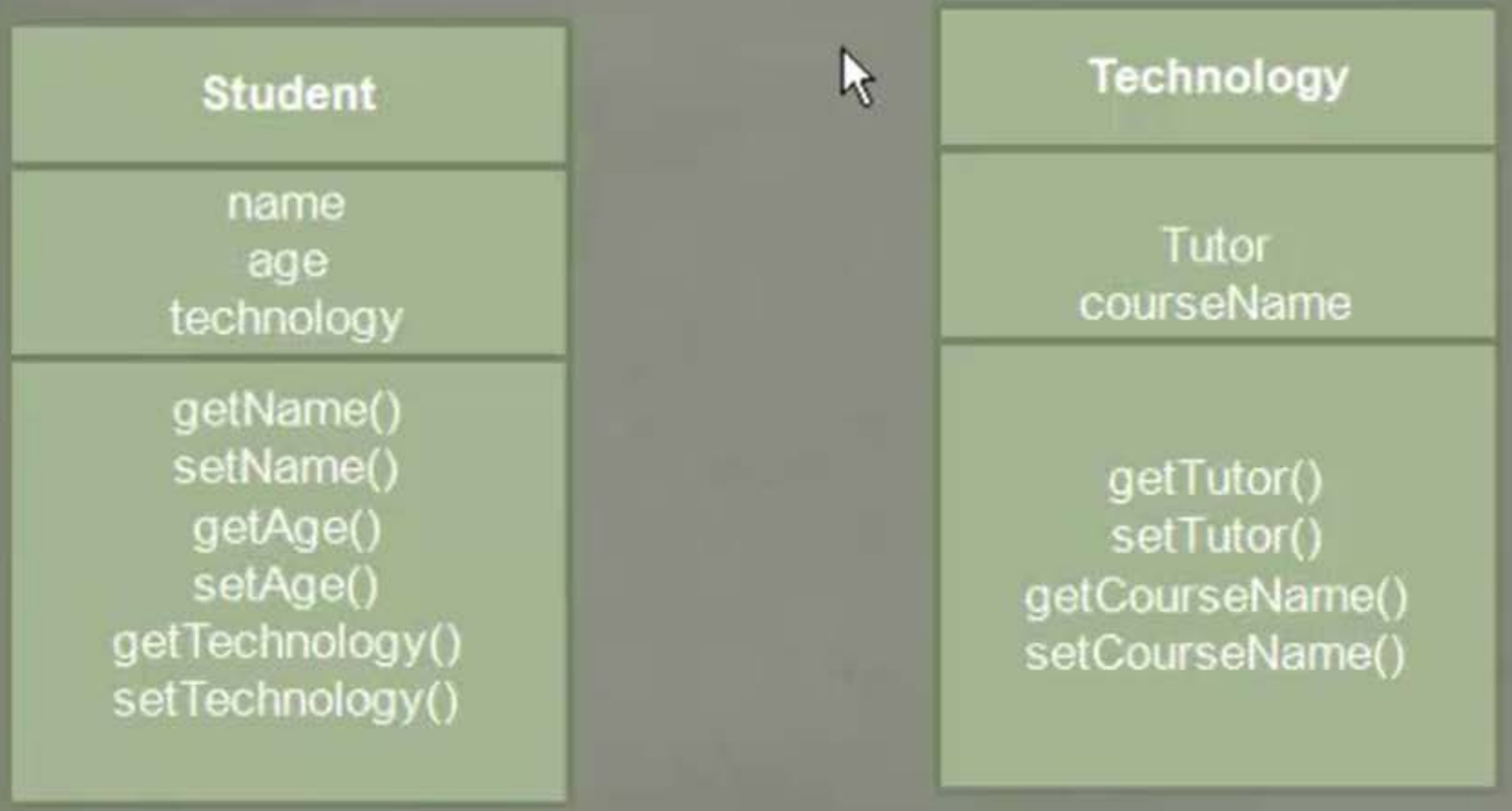
Problem

- Yurk is a famous training centre for various software technologies like Java, Net, Oracle etc
- Since the number of students are quite large, the students data are maintained in the form of csv
- The csv's are maintained technology wise i.e. java students list are maintained in separate csv and so on
- Yurk wants to maintain this information in database that to with good performance

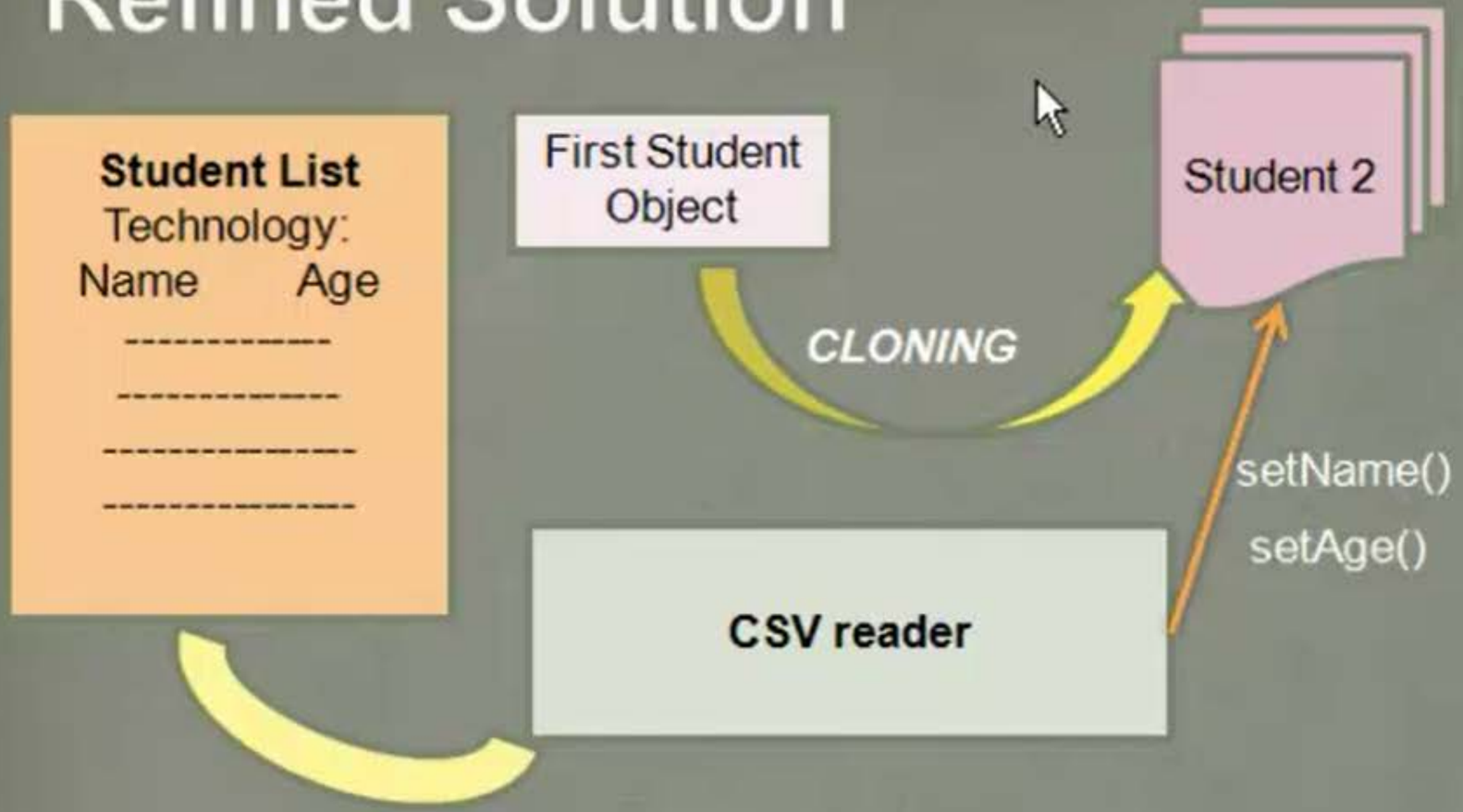
Solution



Student and technology class

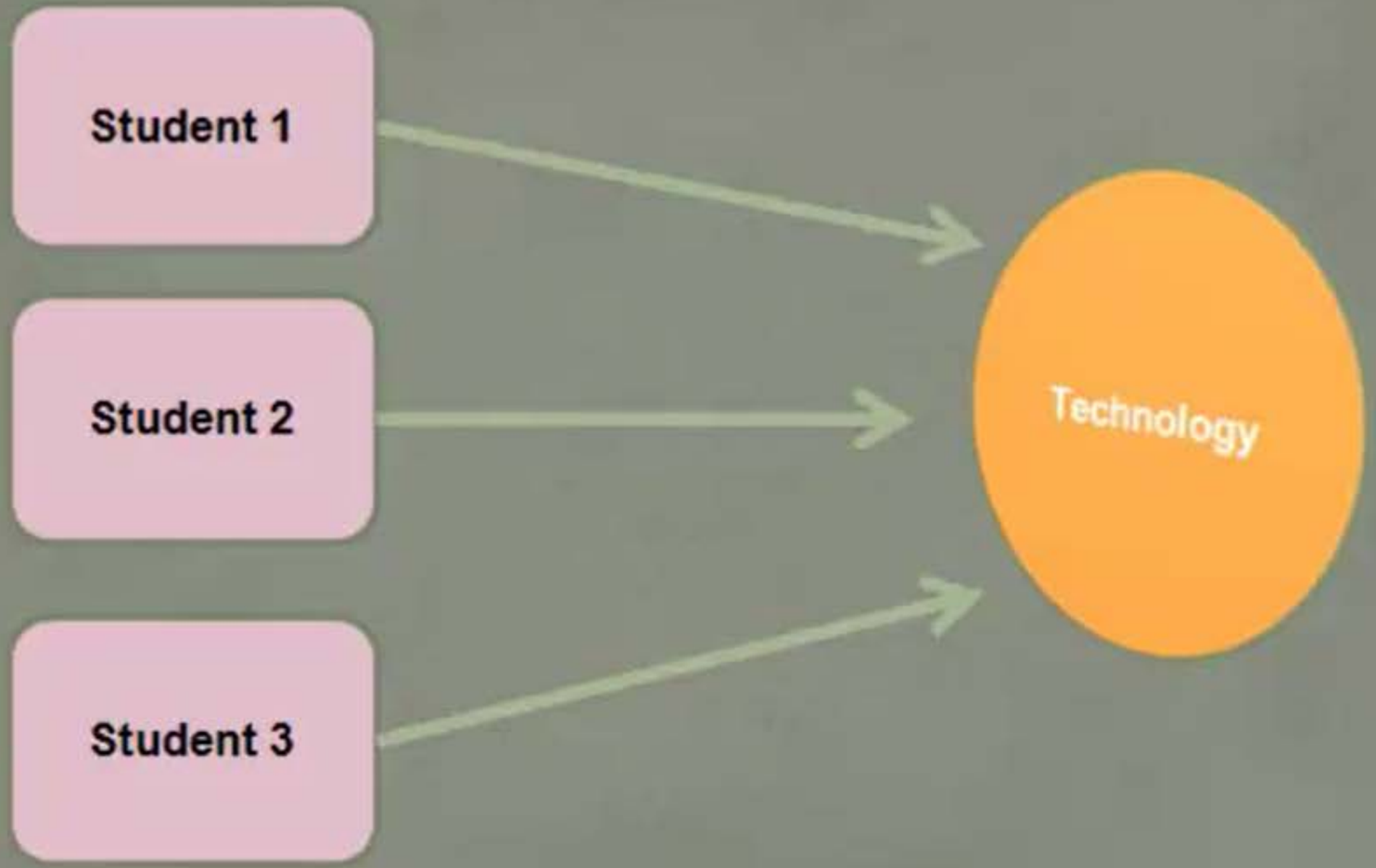


Refined Solution

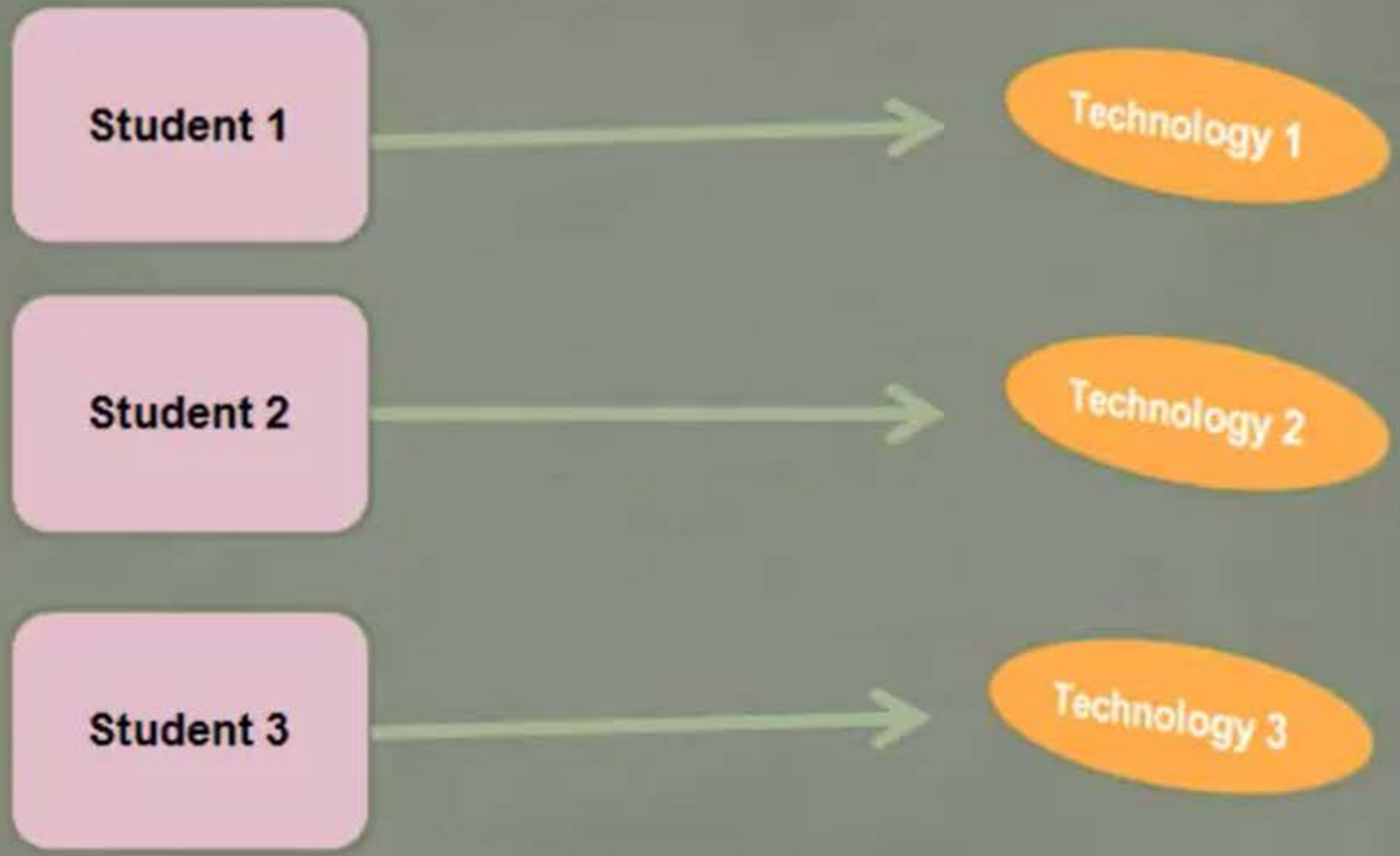


Now suppose if we want to copy all the students objects from CSV to DB, now all the technology and batch of students are same, only name & age should be changed. For this, we can clone the first created object of Student and then clone all the other objects from that and make the changes of age and name in them. Here in this case, it will be a shallow cloning. Instead of creating the whole object and copying all the attributes, we will just clone and change the required attributes.

Soft copy/Shallow copy



Hard copy/Deep copy



Java Cloning: Copy Constructors vs. Cloning

Let's run through the pros and cons of Object.clone() and see how it stacks up against copy constructors when it comes to copying objects.

by Naresh Joshi · Jul. 03, 17 · Java Zone

In my previous article, Shallow and Deep Java Cloning, I discussed Java cloning in detail and answered questions about how we can use cloning to copy objects in Java, the two different types of cloning (Shallow and Deep), and how we can implement both of them. If you haven't read it, please go ahead.

In order to implement cloning, we need to configure our classes and to follow following steps:

- Implement the Cloneable interface in our class or its superclass or interface.
- Define a clone() method that should handle CloneNotSupportedException (either throw or log).
- And, in most cases from our clone() method, we call the clone() method of the superclass.

```
class Person implements Cloneable { // Step 1
    private String name;
    private int income;
    private City city; // deep copy
    private Country country; // shallow copy

    // No @Override, means we are not overriding clone
    public Person clone() throws CloneNotSupportedException { // Step 2
        Person clonedObj = (Person) super.clone(); // Step 3
        clonedObj.city = this.city.clone(); // Making deep copy of city
        return clonedObj;
    }
```

And super.clone() will call its super.clone() and the chain will continue until the call reaches the clone() method of the Object class, which will create a field by field mem copy of our object and return it back.

Like everything, Cloning also comes with its advantages and disadvantages. However, Java cloning is more famous for its design issues but still, it is the most common and popular cloning strategy present today.

Advantages of Object.clone()

Object.clone(), as mentioned, has many design issues, but it is still the most popular and easiest way of copying objects. Some advantages of using clone() are:

- Cloning requires much fewer lines of code — just an abstract class with a 4- or 5-line long clone() method, but we will need to override it if we need deep cloning.
- It is the easiest way of copying objects, especially if we are applying it to an already developed or an old project. We just need to define a parent class, implement Cloneable in it, provide the definition of the clone() method, and we are ready. Every child of our parent will get the cloning feature.
- We should use clone to copy arrays because that's generally the fastest way to do it.
- As of release 1.5, calling clone on an array returns an array whose compile-time

type is the same as that of the array being cloned, which clearly means calling a clone on arrays does not require type casting.

Disadvantages of Object.clone()

Below are some cons that cause many developers not to use Object.clone():

- Using the Object.clone() method requires us to add lots of syntax to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() and casting it on our object.
- The Cloneable interface lacks the clone() method. Actually, Cloneable is a marker interface and doesn't have any methods in it, and we still need to implement it just to tell the JVM that we can perform clone() on our object.
- Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
- We don't have any control over object construction because Object.clone() doesn't invoke any constructor.
- If we are writing a clone method in a child class, e.g. Person, then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
- Object.clone() supports only shallow copying, so the reference fields of our newly cloned object will still hold objects whose fields of our original object was holding. In order to overcome this, we need to implement clone() in every class whose reference our class is holding and then call their clone separately in our clone() method like in the example below.
- We can not manipulate final fields in Object.clone() because final fields can only be changed through constructors. In our case, if we want every Person object to be unique by id, we will get the duplicate object if we use Object.clone() because Object.clone() will not call the constructor, and final id field can't be modified from Person.clone().

```
class City implements Cloneable { private final int id; private String name; public City clon
e() throws CloneNotSupportedException { return (City) super.clone(); } } class Person impleme
nts Cloneable { public Person clone() throws CloneNotSupportedException { Person clonedObj =
(Person) super.clone(); clonedObj.name = new String(this.name); clonedObj.city = this.city.cl
```

```
one(); return clonedObj; } }
```

Because of the above design issues with Object.clone(), developers always prefer other ways to copy objects like using:

- BeanUtils.cloneBean(object): creates a shallow clone similar to Object.clone().
- SerializationUtils.clone(object): creates a deep clone. (i.e. the whole properties graph is cloned, not only the first level), but all classes must implement Serializable.
- Java Deep Cloning Library: offers deep cloning without the need to implement Serializable.

All these options require the use of some external library, plus these libraries will also be using Serialization or Copy Constructors or Reflection internally to copy our object. So if you don't want to go with the above options or want to write your own code to copy the object, then you can use:

1. **Serialization**
2. **Copy constructors**

Serialization

I discussed in 5 Different ways to create objects in Java with Example how we can create a new object using serialization. Similarly, we can also use serialization to copy an object by first Serializing it and again deserializing it like below, or we can also use other APIs like JAXB, which supports serialization.

```
public Person copy(Person original) { try (ObjectOutputStream out = new ObjectOutputStream(ne
w FileOutputStream("data.obj")); ObjectInputStream in = new ObjectInputStream(new FileInputSt
ream("data.obj"))) { out.writeObject(original); return (Person) in.readObject(); } catch (Exc
eption e) { throw new RuntimeException(e); } }
```

But serialization is not solving any problems because we will still not be able to modify the final fields, we still don't have any control on object construction, and we still need to implement Serializable, which is similar to Cloneable. Plus, the serialization process is slower than Object.clone().

Copy Constructors

This method of copying objects is the most popular among the developer community. It overcomes every design issue of Object.clone() and provides better control over object construction.

```
public Person(Person original) { this.id = original.id + 1; this.name = new String(original.name); this.city = new City(original.city); }
```

Advantages of Copy Constructors Over Object.clone()

Copy constructors are better than Object.clone() because they:

- Don't force us to implement any interface or throw any exception, but we can surely do it if it is required.
- Don't require any casts.
- Don't require us to depend on an unknown object creation mechanism.
- Don't require parent classes to follow any contract or implement anything.
- Allow us to modify final fields.
- Allow us to have complete control over object creation, meaning we can write our initialization logic in it.

By using the copy constructors strategy, we can also create conversion constructors, which can allow us to convert one object to another object — e.g. The ArrayList(Collection<? extends E> c) constructor generates an ArrayList from any Collection object and copies all items from the Collection object to a newly created ArrayList object.