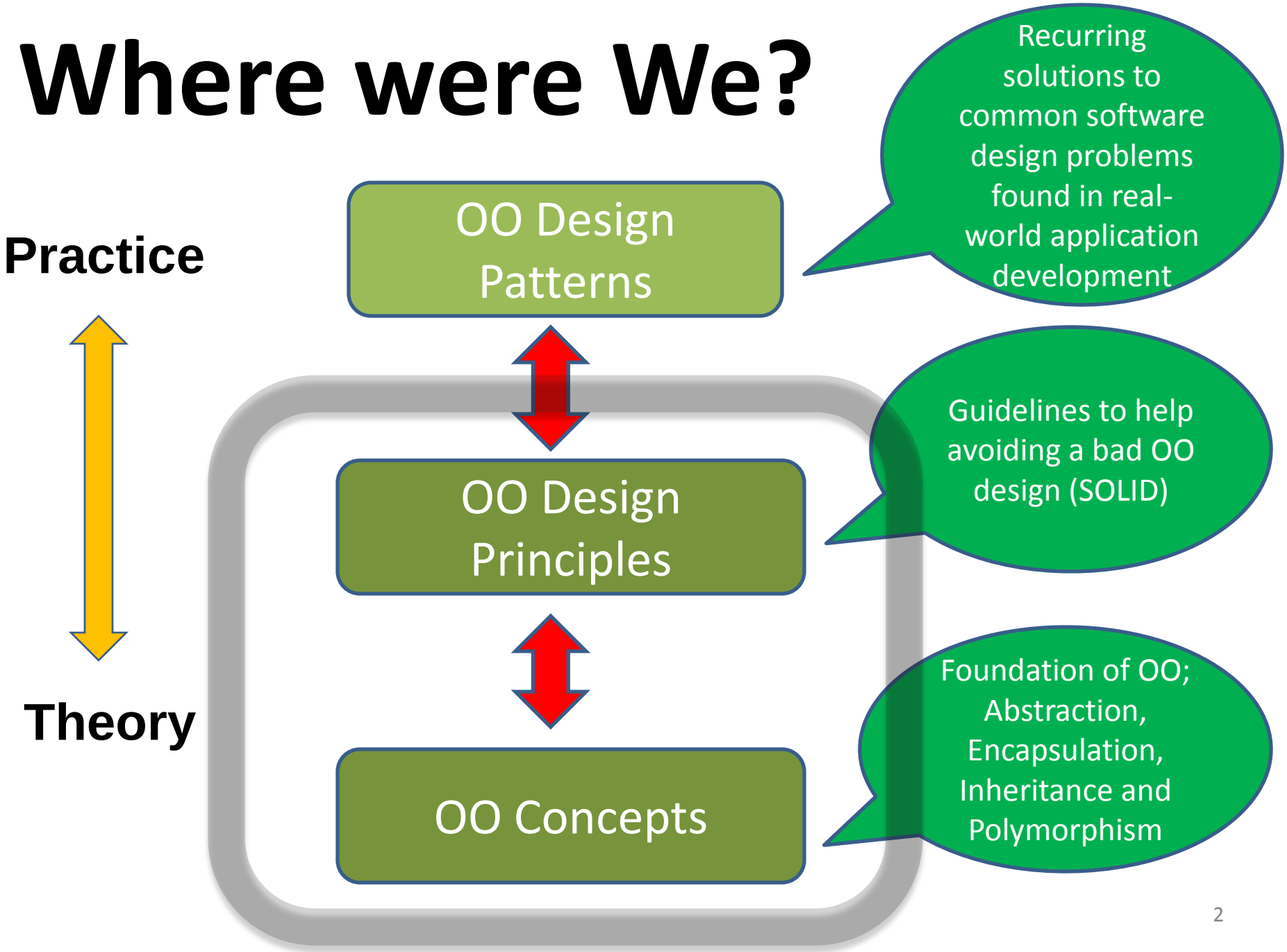
eBuilder TechTalk #7:
Object Oriented Design Patterns - 1 (Creational)

Speaker: Wimal Perera
When: 21/08 (Tuesday) from 9.30AM-11.00AM
Venue: Moonstone, 5th floor

Targeted audience: Developers
However this event is OPEN for anyone who has an interest in the topic.

Where were We?



Where were We?

OO Concepts

UML

OO Design Principles

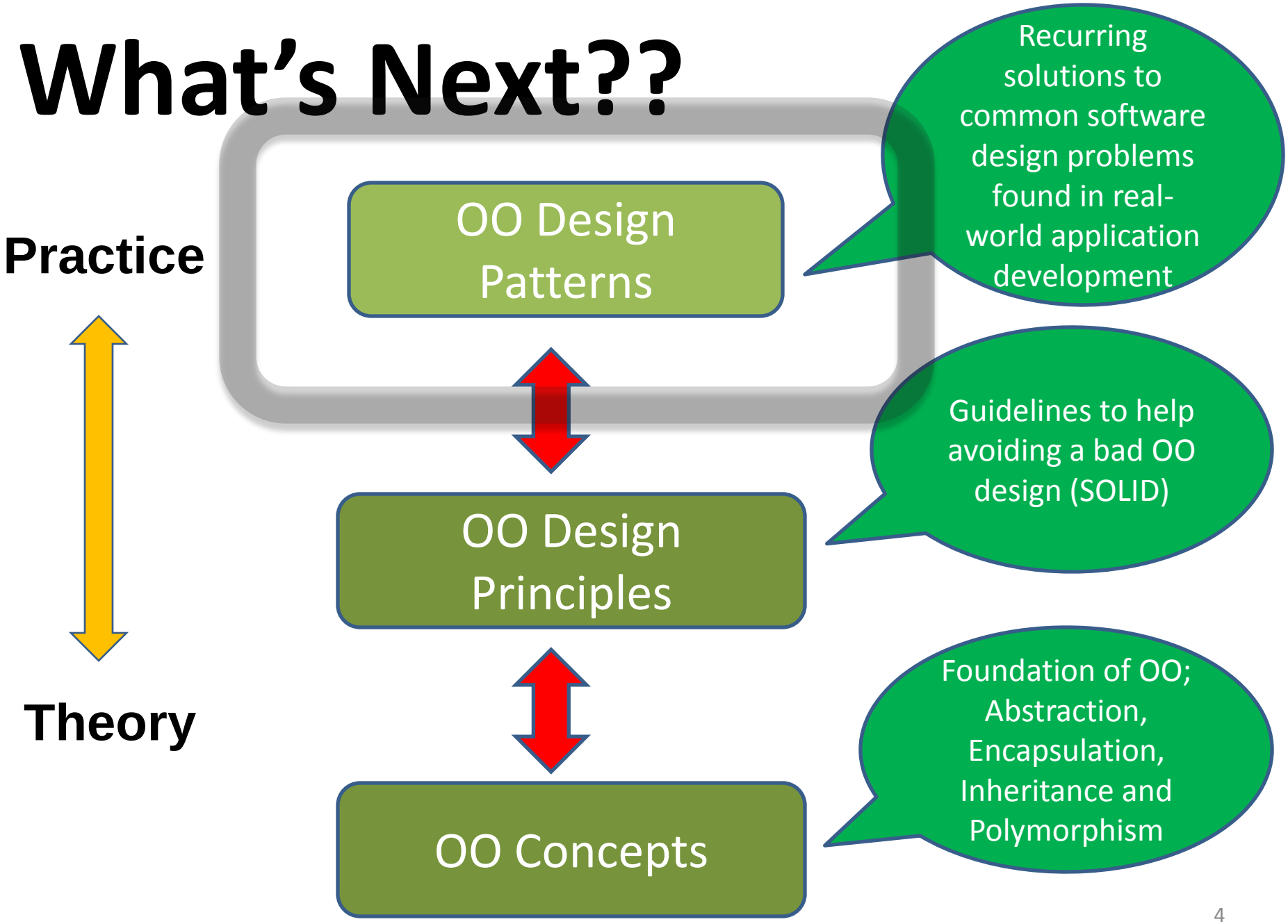
Agile/Scrum

Code Refactoring

Practical Application of OO

PUTTING IT ALL TOGETHER

What's Next??


Practice
OO Design Patterns
Recurring solutions to common software design problems found in real-world application development
Guidelines to help avoiding a bad OO design (SOLID)
OO Design Principles
Foundation of OO; Abstraction, Encapsulation, Inheritance and Polymorphism
OO Concepts
Theory

OO Design Patterns

Creational Patterns
(5)

Structural Patterns
(7)

Behavioural Patterns
(11)

5 + 7 + 11 = 23

Creational Patterns

CREATE objects for you, RATHER THAN having you instantiate objects directly

Provide your program, with the **CAPABILITY** to **DECIDE** which objects to be created based on the **GIVEN CASE**

1. Factory,
2. Abstract Factory, 3. Builder,
4. Singleton, 5. Prototype

Creational Patterns (e.g.)

We **NEVER INSTANTIATE** (i.e. use **"new"** keyword) an object from a **SINGLETON** class

Instead we ***"write code within the Singleton class to enable creation of only one object"***

Structural Patterns

Focus is on how
"CLASSES and OBJECTS are COMPOSED"

EMERGE NEW FEATURES by **ALTERING RELATIONSHIPS between OBJECTS"**

1. Adapter, 2. Bridge,
3. Composite, 4. Decorator,
5. Façade, 6. Flyweight,
7. Proxy

Structural Patterns (e.g.)

"110v printer" connects with a *"230v power supply"* by using 110v~230v **ADAPTER**

"Power Supply-Adapter-Printer" is **COMPOSED** into a single system

A NEW FEATURE; *"110v Power Supply"* is achieved via **ADAPTER**

Behavioural Patterns

Main Concern is about ***"COMMUNICATION BETWEEN OBJECTS"***

1. Template Method,
2. Iterator, 3. Visitor, 4. Observer,
5. Strategy, 6. Memento,
7. State, 8. Mediator,
9. Command, 10. Interpreter,
11. Chain of Responsibility

Behavioural Patterns (Contd …)

*2 Objects **COMMUNICATE** with each other by calling a method of one object within a method of the other*

NEW FEATURES can be implemented by altering how **METHOD INVOCATION** happen between objects

Behavioural Patterns (e.g.)

*Behavioural Patterns heavily rely on **COMBINING** polymorphism with Inheritance*

Template Method enables Implementing the **CONCEPT** of a **"Framework"**

Framework vs. Library

- A Library has a collection of reusable methods. We call these methods from our code.

  Examples in Java;

  System.currentTimeMillis()

  Math.abs(-5.3)

- The code written by us are called by a Framework.

  Examples in Java;

  Overriding doGet() method in javax.http.HttpServlet

  Overriding execute() method in org.apache.struts.action.Action

Behavioural Patterns (e.g.)

Template Method
Implements the
CONCEPT of
“Framework” by
COMBINING
Polymorphism &
Inheritance with
Open-Close Principle

Creational Patterns

With Best Compliments
from BirdSwing TT Services

1. Factory,
2. Abstract Factory,
3. Builder,
4. Singleton, 5. Prototype

The Story of 4 Musketeers

- Once upon a time in eBuilder, there lived 4 Musketeers.

1) SIDA
The **"Cleverest House-builder"** of All
&
"TT Champion" of All Time

The 4 Musketeers

2) <u>RASI</u>

A friend indeed,

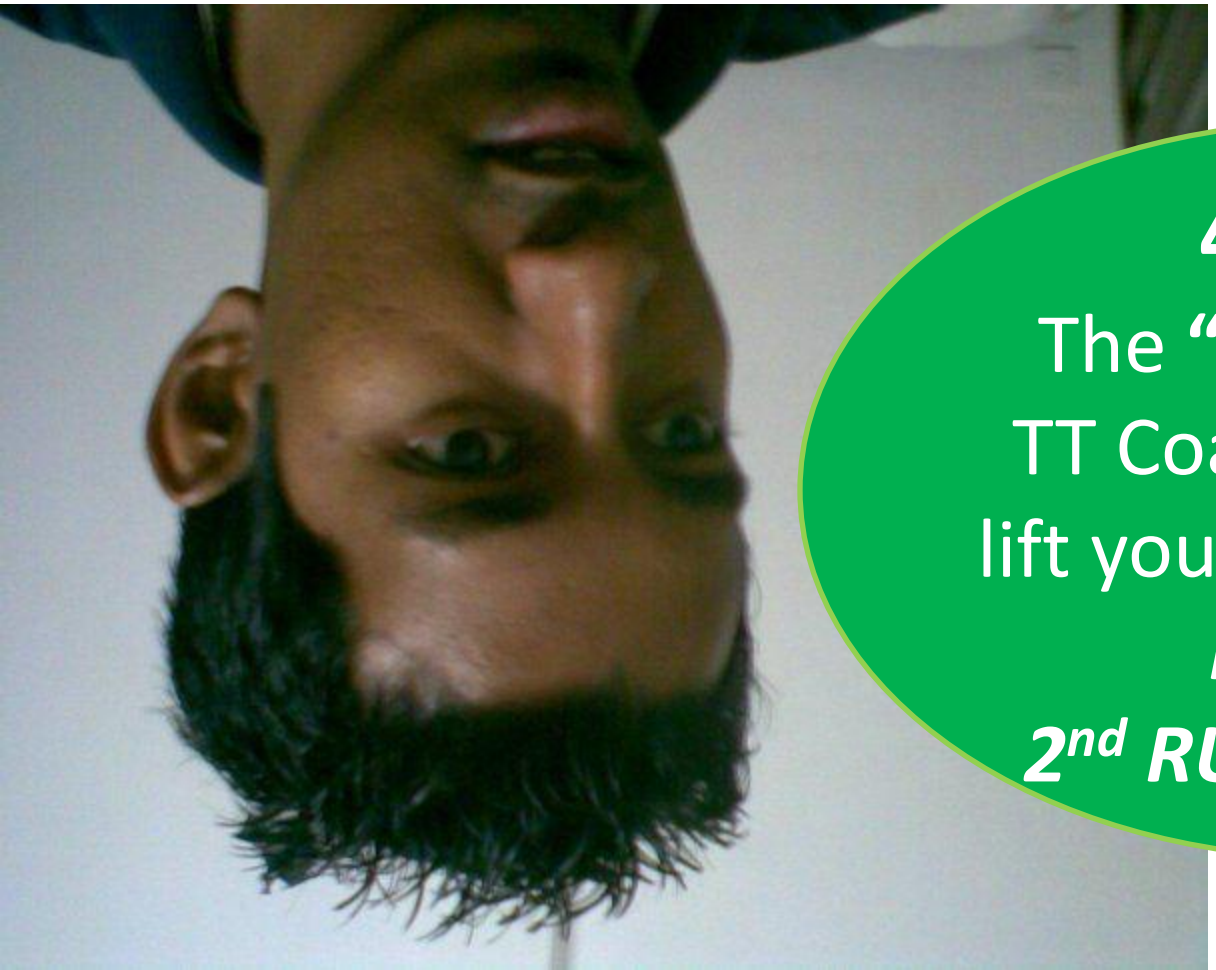
founder of

"TT Killer Spin"

The 4 Musketeers

3) <u>WIMA</u>
The humble developer, founder of ***"TT Kurulu"*** service

The 4 Musketeers



4) <u>ISSA</u>
The **"ALMIGHTY"**
TT Coach, who can lift you from ***"lowest rank to 2nd RUNNERS UP"***

The 4 Musketeers (Contd …)

- It all began when **SIDA** started to build a house on his own.
- Unfortunately **SIDA** couldn't find all cash needed to build the house and had to buy so many loans from so many places to build his house.
- Thus **SIDA** became "the largest DEBTOR having the largest BUNGLOW" by the time of completing his house.

The 4 Musketeers (Contd …)

- Since **SIDA** couldn't afford all his DEBTs from eBuilder's salary, he decided to start a **HOUSE BUILDING COMPANY** on his own as a part-time business.
- **SIDA's** company initially supported building houses similar to the bunglow built by **SIDA**: "SIDA's MAKABAS MODEL".
- **SIDA** decided to develop the initial website for his business on his own.

```
import java.util.Map;

public class SiDaConstructionsHomePage extends AbstractHTMLPage {

    public void render(Map<String, Object> arguments) {

        super.setPageHeader(
            "<title>Sida's MAKABAS House ...</title>");

        super.setPageDescription(
            "<p>Mama haduwa wagema umbalatath geyak hadala dennam !!!</p>"
        );

        super.setPageBody("<img src='images/sida_house.jpg'>");

        super.setPageFooter(
            "<p>We build houses at malabe, kotte and wattala</p>");
    }

}
```

The 4 Musketeers (Contd …)

- By the time **SIDA** was building his house, his fellow friend **RASI** also started building a house on its own by following **SIDA'**s strategies.
- Thus **RASI** also became "the largest DEBTOR having the largest BUNGLOW" and had no option other than joining **SIDA**'s "HOUSE BUILDING COMPANY" for settling his debts.

The 4 Musketeers (Contd …)

- But **RASI** was more interested in selling cars rather than building houses.
- The group of companies formed by **SIDA** and **RASI** had **“2 PRODUCT CATEGORIES”**; houses and cars.
- **SIDA** and **RASI** changed the company website to demonstrate both their **PRODUCT CATEGORIES** for the customers.

```
import java.util.Map;
public class SandRGroupHomePage extends AbstractHTMLPage {
    public void render(Map<String, Object> arguments) {
        if(arguments.get("category").equals("houses")) {
            super.setPageHeader("<title>Sida's Stunning House ...</title>");
            super.setPageDescription(
                "<p>Mama haduwa wagema umbalatath geyak hadala dennam !!!</p>"
            );
            super.setPageBody("<img src='images/sida_house.jpg'>");
        } else if(arguments.get("category").equals("cars")) {
            super.setPageHeader("<title>Rasiya's Civic Car ...</title>");
            super.setPageDescription(
                "<p>Sidagen ahala nam apahu gewal hadanna epo !!!</p>"
            );
            super.setPageBody("<img src='images/rasith_car.jpg'>");
        } else {
            throw new RuntimeException("Product Category not supported !!!!");
        }
        super.setPageFooter(
            "<p>Contact our sales centers at malabe, kotte and nawala</p>");
    }
}
```

The 4 Musketeers (Contd …)

- The business went on.
- **SIDA** and **RASI** were gradually settling their debts.
- However both of them realized that their website will not be maintainable if they try to demonstrate more and more different types of houses and cars in their website.

The 4 Musketeers (Contd …)

- **SIDA** and **RASI** hired **WIMA** to fix the source code of the current company website.
- With the bird swing of a single TT (Table Tennis) service, **WIMA** modified the existing source code.

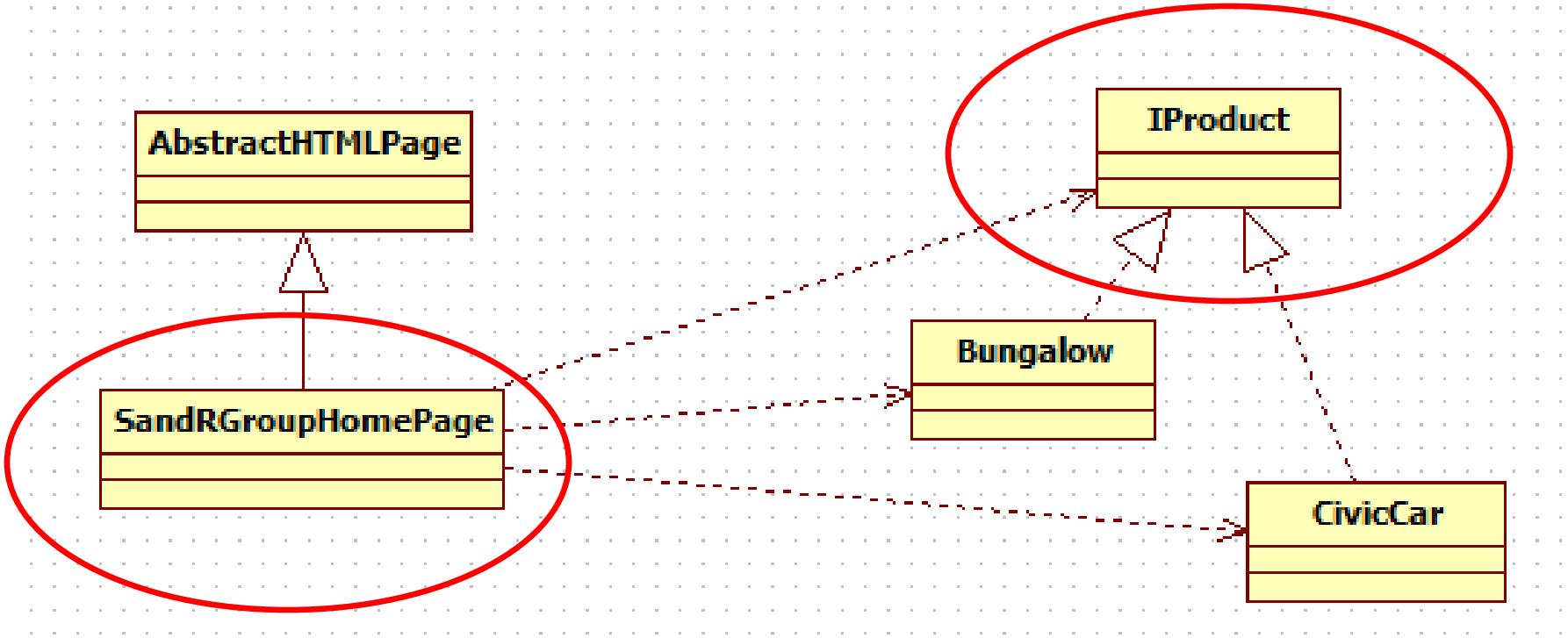
```
public interface IProduct {
    public String getHeading();
    public String getDescription();
    public String getProduct();
}
```

```java
public class Bunglow implements IProduct {
    public String getDescription() {
        return "Sidath Bunglows !!!";
    }
    public String getHeading() {
        return "Sida Model Bunglow ...";
    }
    public String getProduct() {
        // put logic to get all features of a bunglow here
        return "images/sida_house.jpg";
    }
}
```

```java
public class CivicCar implements IProduct {

    public String getDescription() {
        return "Honda civic is the best !!!!";
    }
    public String getHeading() {
        return "Honda Civic Car ...";
    }
    public String getProduct() {
        //put all logic to get product features of a honda civic car here ...
        return "images/rasith_car.jpg";
    }

}
```

```
public class SandRGroupHomePage extends AbstractHTMLPage {

    public void render(Map<String, Object> arguments) {
        String category = (String)arguments.get("category");
        IProduct product = this.getProductByCategory(category);
        super.setPageHeader("<title>" + product.getHeading() + "</title>");
        super.setPageDescription("<p>" + product.getDescription() + "</p>");
        super.setPageBody("<img src='" + product.getProduct() + "'/>");

        // they sell both of these product categories
        super.setPageFooter(
            "<p>Contact our sales centers at malabe, kotte and wattala</p>");
    }

    private IProduct getProductByCategory(String category) {
        if(category.equals("houses")) {
            return new Bunglow();
        } else if(category.equals("cars")) {
            return new CivicCar();
        } else {
            throw new RuntimeException("Unsupported Product Category ...");
        }

    }

}
```

Design with Factory Method



The 4 Musketeers (Contd …)

- **SIDA** and **RASI** asked “What have you done **WIMA?**”
- This is called the **“factory method”**
- It is **NOT** the **“factory design pattern”**
- But the code is in a satisfactory state for now

The 4 Musketeers (Contd …)

- As the business grew with more opportunities, **SIDA** and **RASI** decided to extend their business.
- **RASI** decided to sell **“Vitz”** cars in addition to **“Civic”** cars.
- **SIDA** decided to take on **“Office Buildings”** in addition to **“Bunglow”** constructions.

The 4 Musketeers (Contd …)

- **SIDA** and **RASI** consulted **WIMA** to alter the website based on the new business requirements.
- **WIMA** decided that now it is time to move to **“Factory Design Pattern”** from the current implementation of **“factory method”**.

Factory Design Pattern

CREATE objects *"WITHOUT specifying the exact class of the object"*

```
class OfficeBuilding implements IProduct {
    public String getDescription() {
        return "Office Buildings !!!";
    }
    public String getHeading() {
        return "Office Building Model ...";
    }
    public String getProduct() {
        // put logic to get all features of an office building here
        return "images/office_buildings.jpg";
    }
}
```

```
class VitzCar implements IProduct {

    public String getDescription() {
        return "Vitz is the best !!!!";
    }
    public String getHeading() {
        return "Vitz Car ...";
    }
    public String getProduct() {
        //put all logic to get product features of a vitz car here ...
        return "images/vitz_car.jpg";
    }

}
```

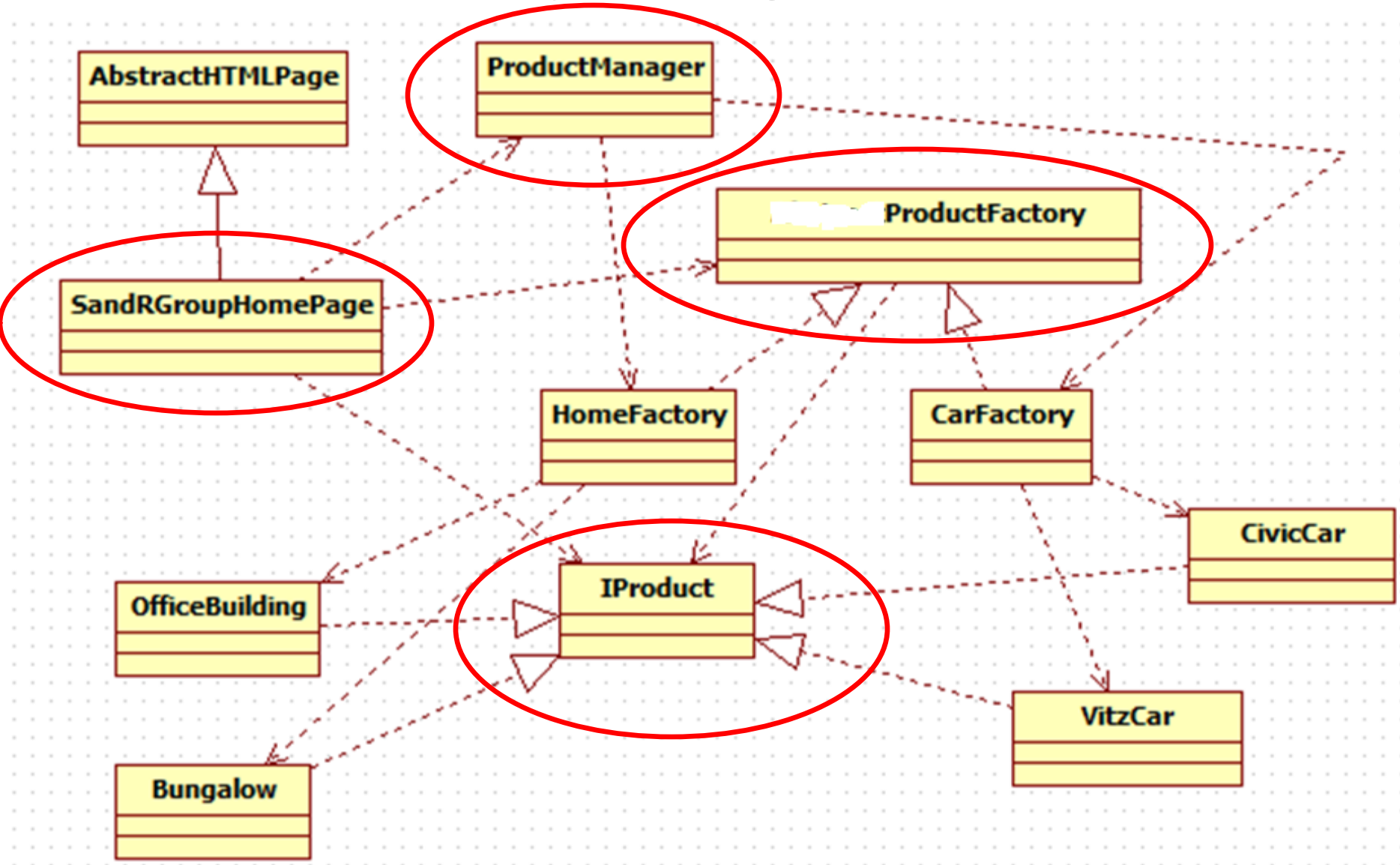
```
public interface ProductFactory {
    public IProduct createProductByModel(String model);
}
```

```
public class HomeFactory implements ProductFactory {

    public IProduct createProductByModel(String model) {
        if(model.equals("bunglow")) { return new Bunglow();
        } else if(model.equals("office")) { return new OfficeBuilding();
        } else { throw new IllegalArgumentException("Invalid Model ...");
        }
    }

}
```

```
public class CarFactory implements ProductFactory {

    public IProduct createProductByModel(String model) {
        if(model.equals("civic")) { return new CivicCar(); }
        else if(model.equals("vitz")) { return new VitzCar(); }
        else { throw new IllegalArgumentException("Invalid Model ...");
        }
    }

}
```

```
class ProductManager {
    public static ProductFactory getProductFactoryByCategory(String productCategory) {
        ProductFactory factory;
        // NOTE: You can use an xml/property file instead of if-else statements
        if(productCategory.equals("houses")) {
            factory = new HomeFactory();
        } else if(productCategory.equals("cars")) {
            factory = new CarFactory();
        } else {
            throw new RuntimeException("Unsupported Product Category ...");
        }
        return factory;
    }
}
public class SandRGroupHomePage extends AbstractHTMLPage {

    public void render(Map<String, Object> arguments) {
        String category = (String)arguments.get("category");
        String model = (String)arguments.get("model");

        ProductFactory productFactory = ProductManager.getProductFactoryByCategory(category);
        IProduct product = productFactory.createProductByModel(model);

        super.setPageHeader("<title>" + product.getHeading() + "</title>");
        super.setPageDescription("<p>" + product.getDescription() + "</p>");
        super.setPageBody("<img src='" + product.getProduct() + "'/>");

        // they build both of these types of houses
        super.setPageFooter(
            "<p>Contact our sales centers at malabe, kotte and wattala</p>");
    }
}
```

Factory Design Pattern



The 4 Musketeers (Contd …)

- **SIDA** decided to stop building houses at Wattala area, since it’s too far for him to go and not much orders are coming.
- **RASI** decided to close the Car Center at Kotte and put remaining cars to Wattala, since Kotte didn’t gave him much sales.

The 4 Musketeers (Contd …)

- **SIDA** and **RASI** again consulted **WIMA**, to change the website with these new business requirements.
- **WIMA** noticed that **“Products”** and **“SalesCenters”** share some **“COMMON THEME”** which is based on **“Product Category”**.
- **WIMA** decided that now it’s time to use **“Abstract Factory”**.

Abstract Factory

GROUPING of
"object FACTORIES"
that have a
"COMMON THEME"

Defines a
"WEB OF INTERFACES"
to represent this
"COMMON THEME"

```java
public interface ISalesCenter {
    public String getDescription();
    public String getLocation();
    public String getAddress();
}
```

```java
class HousingSalesCenter implements ISalesCenter {

    private String location;
    private String address;

    public HousingSalesCenter(String location, String address) {
        this.location = location;
        this.address = address;
    }

    public String getAddress() { return address; }
    public String getLocation() { return location; }

    public String getDescription() {
        // TODO Implement an attractive description
        // related to selling houses based on
        // the given location and address
        return null;
    }
}
```

```
class CarSalesCenter implements ISalesCenter {

    private String location;
    private String address;

    public CarSalesCenter(String location, String address) {
        this.location = location;
        this.address = address;
    }

    public String getAddress() { return address; }
    public String getLocation() { return location; }

    public String getDescription() {
        // TODO Implement an attractive description
        // related to selling cars based on
        // the given location and address
        return null;
    }
}
```

```
public interface AbstractProductFactory {
    public IProduct createProductByModel(String model);
    public ISalesCenter[] getAllSalesCenters();
}
```

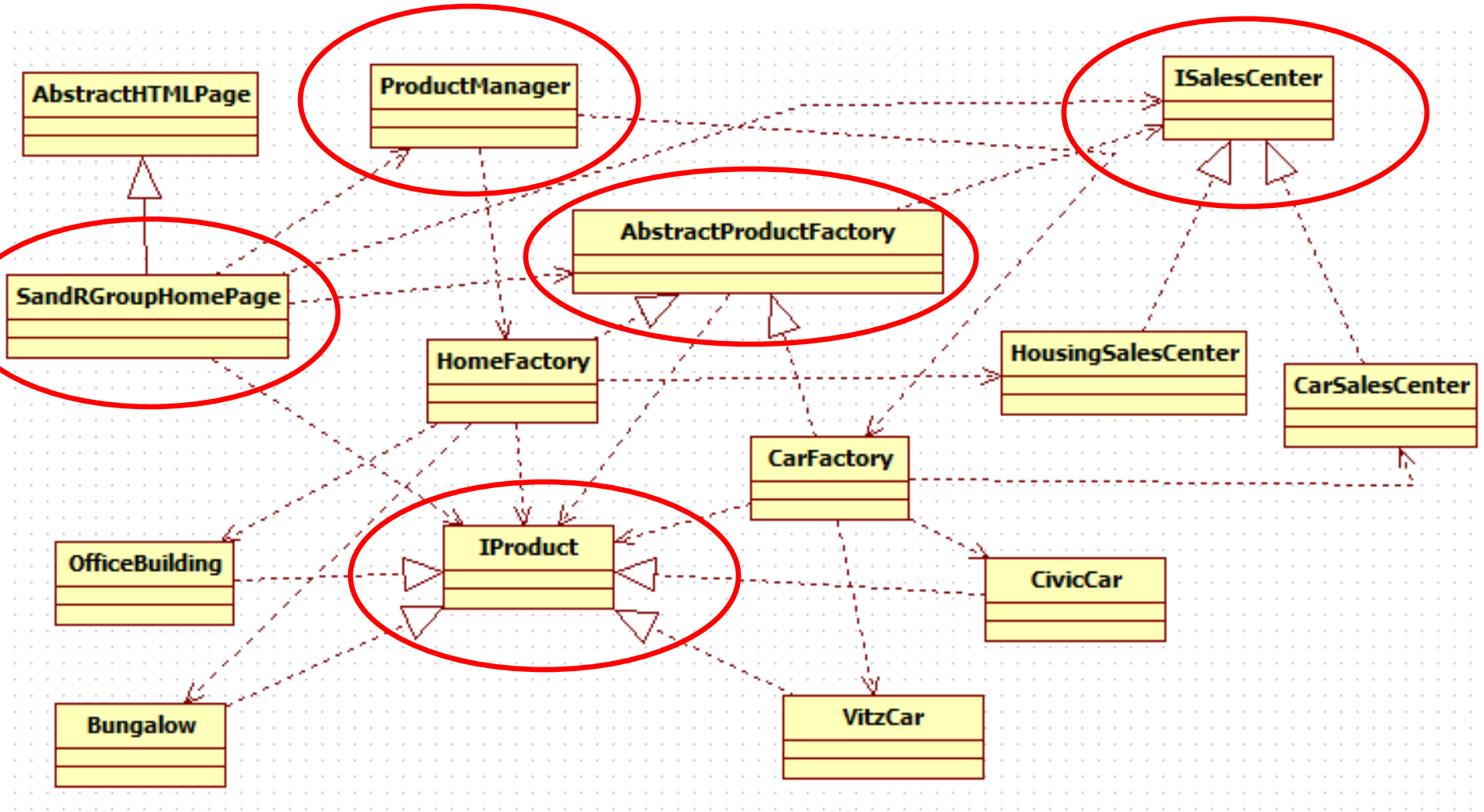
```
public class CarFactory implements AbstractProductFactory {

    public IProduct createProductByModel(String model) {
        if(model.equals("civic")) { return new CivicCar(); }
        else if(model.equals("vitz")) { return new VitzCar(); }
        else { throw new IllegalArgumentException("Invalid Model ...");
        }
    }

    public ISalesCenter[] getAllSalesCenters() {
        return new ISalesCenter[] {
            new CarSalesCenter("malabe", "malabe"),
            new CarSalesCenter("wattala", "wattala")
        };
    }

}
```

```
public class HomeFactory implements AbstractProductFactory {

    public IProduct createProductByModel(String model) {
        if(model.equals("bunglow")) { return new Bunglow();
        } else if(model.equals("office")) { return new OfficeBuilding();
        } else { throw new IllegalArgumentException("Invalid Model ...");
        }
    }

    public ISalesCenter[] getAllSalesCenters() {
        return new ISalesCenter[] {
            new HousingSalesCenter("malabe", "malabe"),
            new HousingSalesCenter("kotte", "kotte")
        };
    }

}
```

```
class ProductManager {
    public static AbstractProductFactory getProductFactoryByCategory(String productCategory) {
        AbstractProductFactory factory;
        // NOTE: You can use an xml/property file instead of if-else statements
        if(productCategory.equals("houses")) {
            factory = new HomeFactory();
        } else if(productCategory.equals("cars")) {
            factory = new CarFactory();
        } else {
            throw new RuntimeException("Unsupported Product Category ...");
        }
        return factory;
    }
}
```

```
public class SandRGroupHomePage extends AbstractHTMLPage {

    public void render(Map<String, Object> arguments) {
        String category = (String)arguments.get("category");
        String model = (String)arguments.get("model");

        AbstractProductFactory productFactory = ProductManager.getProductFactoryByCategory(category);
        IProduct product = productFactory.createProductByModel(model);

        super.setPageHeader("<title>" + product.getHeading() + "</title>");
        super.setPageDescription("<p>" + product.getDescription() + "</p>");
        super.setPageBody("<img src='" + product.getProduct() + "'/>");

        ISalesCenter[] salesCenters = productFactory.getAllSalesCenters();
        String footerInfo = "<p>Contact our sales centers at: ";
        for (ISalesCenter salesCenter : salesCenters) {
            footerInfo += salesCenter.getDescription();
        }
        footerInfo += "</p>";
        super.setPageFooter(footerInfo);
    }
}
```

Abstract Factory



The 4 Musketeers (Contd …)

- Suddenly **WIMA** noticed that some parts of code in **HousingSalesCenter** and **CarSalesCenter** are the same.

```
class HousingSalesCenter implements ISalesCenter {

    private String location;
    private String address;

    public HousingSalesCenter(String location, String address) {
        this.location = location;
        this.address = address;
    }

    public String getAddress() { return address; }
    public String getLocation() { return location; }

    public String getDescription() {
        // TODO Implement an attractive description
        // related to selling houses based on
        // the given location and address
        return null;
    }
}
```

```
class CarSalesCenter implements ISalesCenter {

    private String location;
    private String address;

    public CarSalesCenter(String location, String address) {
        this.location = location;
        this.address = address;
    }

    public String getAddress() { return address; }
    public String getLocation() { return location; }

    public String getDescription() {
        // TODO Implement an attractive description
        // related to selling cars based on
        // the given location and address
        return null;
    }
}
```

The 4 Musketeers (Contd …)

- **WIMA** thought why not find a way to reuse the same code, without replicating it.
- **WIMA** decided to refactor ISalesCenter "interface" into an "**abstract class**", SalesCenter.

```
public abstract class SalesCenter {
    private String location;
    private String address;
    public SalesCenter(String location, String address) {
        this.location = location;
        this.address = address;
    }
    public String getAddress() { return address; }
    public String getLocation() { return location; }

    public abstract String getDescription();
}
```

```
class HousingSalesCenter extends SalesCenter {

    public HousingSalesCenter(String location, String address) {
        super(location, address);
    }
    public String getDescription() {
        // TODO Implement an attractive description
        // related to selling houses based on
        // the given location and address
        return null;
    }
}
class CarSalesCenter extends SalesCenter {

    public CarSalesCenter(String location, String address) {
        super(location, address);
    }
    public String getDescription() {
        // TODO Implement an attractive description
        // related to selling cars based on
        // the given location and address
        return null;
    }
}
```

The 4 Musketeers (Contd …)

- **ISSA** (the almighty TT coach), came across **SIDA** and **RASI**'s company website.
- **ISSA** contacted **SIDA** and asked for a housing catalogue on all available housing models, so that he can select a house model based on them.

The 4 Musketeers (Contd …)

- **SIDA** was upset that he didn't have such a housing catalogue, other than the **Bunglow** and **Office Building**, he built.
- **SIDA** decided that it is time to build a housing catalogue with several housing models and introduce this housing catalogue to customers like **ISSA**.

The 4 Musketeers (Contd …)

- **SIDA** defined a house catalog as follows.

Model	Features			
	Foundation	**Garden**	**Floor**	**Roof**
Bungalow	Full Stones	With Pond	Floor Tiles	Asbestos
Kingfisher	Cement blocks and Stones	Plain Garden	Terrazzo	Clay Tiles
Nest	Full Stones	With Pond	Floor Tiles	Asbestos
Igloo	Full Stones	Plain Garden	Terrazzo	Asbestos
Office Building	Full Stones	No garden	Floor Tiles	Clay Tiles

- **SIDA** consulted **WIMA** to include all houses in this catalog in the company website.

The 4 Musketeers (Contd …)

1. **WIMA** saw that **CONSTRUCTION** of a house object is now **COMPLEX** than previously it was.
2. **WIMA** saw that certain features have **“repetitive patterns”** among different house models.

The 4 Musketeers (Contd …)

- Initially **WIMA** thought why not write 5 classes to represent 5 house models and initialize them via **HomeFactory** ?
- But **WIMA** realized that in this case, code inside these 5 classes will be repeated again and again.
- **WIMA** decided that it’s time to convert his **HomeFactory** into a **HomeBuilder**.

Builder

Construct objects by
"SEPERATING
between
CONSTRUCTION and
REPRESENTATION"

The 4 Musketeers (Contd …)

Model	Features			
	Foundation	Garden	Floor	Roof
Bungalow	Full Stones	With Pond	Floor Tiles	Asbestos
Kingfisher	Cement blocks and Stones	Plain Garden	Terrazzo	Clay Tiles
Nest	Full Stones	With Pond	Floor Tiles	Asbestos
Igloo	Full Stones	Plain Garden	Terrazzo	Asbestos
Office Building	Full Stones	No garden	Floor Tiles	Clay Tiles

- **WIMA** started by implementing all the features separately.

```
public interface Foundation {
    public String getFoundationInfo();
}
class FullStonesFoundation implements Foundation {
    public String getFoundationInfo() {
        return "Full Stones Foundation";
    }
}
class MixedFoundation implements Foundation {
    public String getFoundationInfo() {
        return "Full Stones with Cement Blocks Foundation";
    }
}
```

```
interface Garden {
    public String getGardenInfo();
}
class PondGarden implements Garden {
    public String getGardenInfo() {
        return "Garden with Pond";
    }
}
class PlainGarden implements Garden {
    public String getGardenInfo() {
        return "Plain Garden";
    }
}
class NoGarden implements Garden {
    public String getGardenInfo() {
        return "Empty Garden";
    }
}
```

```
interface Floor {
    public String getFloorInfo();
}
class TerrazzoFloor implements Floor {
    public String getFloorInfo() {
        return "Terrazzo Floor";
    }
}
class TileFloor implements Floor {
    public String getFloorInfo() {
        return "Floor Tiles";
    }
}
```

```
interface Roof {
    public String getRoofInfo();
}
class AsbestosRoof implements Roof {
    public String getRoofInfo() {
        return "Asbestos Roof";
    }
}
class ClayTileRoof implements Roof {
    public String getRoofInfo() {
        return "Roof with Clay Tiles";
    }
}
```

The 4 Musketeers (Contd …)

- **WIMA** created;
  1. a class called **House** to hold the **REPRESENTATION** of a House
  2. an interface **HouseBuilder** which is used as guidelines behind **CONSTRUCTION** of a House
  3. a class called **HouseDirector** for building the House based on selected **CONSTRUCTION**

```
public class House {
    private Floor floor;
    private Foundation foundation;
    private Garden garden;
    private Roof roof;
    public void setFoundation(Foundation foundation) { this.foundation = foundation; }
    public Foundation getFoundation() { return this.foundation; }
    public void setGarden(Garden garden) { this.garden = garden; }
    public Garden getGarden() { return this.garden; }
    public void setFloor(Floor floor) { this.floor = floor; }
    public Floor getFloor() { return this.floor; }
    public void setRoof(Roof roof) { this.roof = roof; }
    public Roof getRoof() { return this.roof; }
}
```

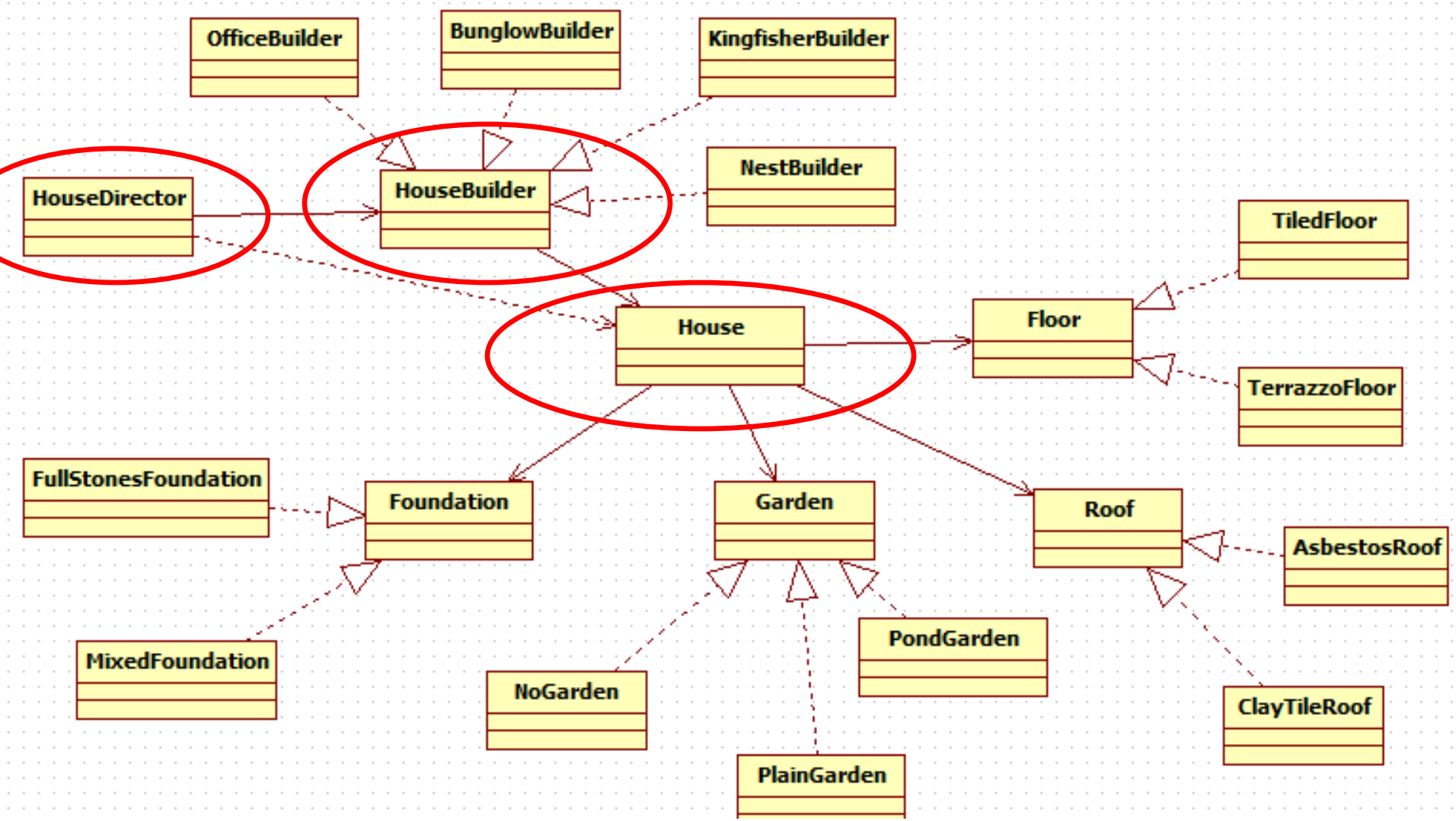
```
interface HouseBuilder {
    public void buildFoundation();
    public void buildGarden();
    public void buildFloor();
    public void buildRoof();
    public House getHouse();
}

class HouseDirector {
    private HouseBuilder houseBuilder;
    public HouseDirector(HouseBuilder builder) { this.houseBuilder = builder; }
    public void constructHouse() {
        this.houseBuilder.buildFoundation();
        this.houseBuilder.buildGarden();
        this.houseBuilder.buildFloor();
        this.houseBuilder.buildRoof();
    }
    public House getHouse() {
        return this.houseBuilder.getHouse();
    }
}
```

```
class BungalowBuilder implements HouseBuilder {
    private House house;
    public BungalowBuilder() { this.house = new House(); }
    public void buildFoundation() {
        this.house.setFoundation(new FullStonesFoundation()); }
    public void buildGarden() { this.house.setGarden(new PondGarden()); }
    public void buildFloor() { this.house.setFloor(new TileFloor()); }
    public void buildRoof() { this.house.setRoof(new AsbestosRoof()); }
    public House getHouse() { return this.house; }
}

class KingfisherBuilder implements HouseBuilder {
    private House house;
    public KingfisherBuilder() { this.house = new House(); }
    public void buildFoundation() { this.house.setFoundation(new MixedFoundation()); }
    public void buildGarden() { this.house.setGarden(new PlainGarden()); }
    public void buildFloor() { this.house.setFloor(new TerrazzoFloor()); }
    public void buildRoof() { this.house.setRoof(new ClayTileRoof()); }
    public House getHouse() { return this.house; }
}
```

```
class NestBuilder implements HouseBuilder {
    private House house;
    public NestBuilder() { this.house = new House(); }
    public void buildFoundation() {
        this.house.setFoundation(new FullStonesFoundation()); }
    public void buildGarden() { this.house.setGarden(new PondGarden()); }
    public void buildFloor() { this.house.setFloor(new TileFloor()); }
    public void buildRoof() { this.house.setRoof(new AsbestosRoof()); }
    public House getHouse() { return this.house; }
}

class IglooBuilder implements HouseBuilder {
    private House house;
    public IglooBuilder() { this.house = new House(); }
    public void buildFoundation() {
        this.house.setFoundation(new FullStonesFoundation()); }
    public void buildGarden() { this.house.setGarden(new PlainGarden()); }
    public void buildFloor() { this.house.setFloor(new TerrazzoFloor()); }
    public void buildRoof() { this.house.setRoof(new AsbestosRoof()); }
    public House getHouse() { return this.house; }
}
```

```
class OfficeBuilder implements HouseBuilder {
    private House house;
    public OfficeBuilder() { this.house = new House(); }
    public void buildFoundation() {
        this.house.setFoundation(new FullStonesFoundation()); }
    public void buildGarden() { this.house.setGarden(new NoGarden()); }
    public void buildFloor() { this.house.setFloor(new TileFloor()); }
    public void buildRoof() { this.house.setRoof(new ClayTileRoof()); }
    public House getHouse() { return this.house; }
}
```

Builder Pattern



Using **Builder** to build a House …

```
class SampleUse {
    public House getHouseByModel(String model) {

        HouseDirector director = null;
        if(model.equals("bunglow")) {
            director = new HouseDirector(new BungalowBuilder());
        } else if(model.equals("kingfisher")) {
            director = new HouseDirector(new KingfisherBuilder());
        } else if(model.equals("nest")) {
            director = new HouseDirector(new NestBuilder());
        } else if(model.equals("igloo")) {
            director = new HouseDirector(new IglooBuilder());
        } else {
            director = new HouseDirector(new OfficeBuilder());
        }

        director.constructHouse();
        return director.getHouse();
    }
}
```

The 4 Musketeers (Contd …)

- Back to **SIDA** and **RASI**'s company website;
  1. A House is built from scratch, component by component. Therefore **CONSTRUCTION** of a house is **COMPLEX**.
  2. A Car is sold as a whole object. No manufacturing is done such as assembling components inside a car.
- Houses need **Builder**. But for Cars **Factory** is sufficient.
- How to combine **HouseBuilder** with the already existing **AbstractProductFactory**?

The 4 Musketeers (Contd …)

- **WIMA** was wondering how to embed **“Builder”** logic while maintaining **“Abstract Factory”**.
- After the swing of several TT “Kurulu” services, **WIMA** did the following fix.

```
public class House implements IProduct {

    private String heading;
    public House(String heading) { this.heading = heading; }
    public String getDescription() {
        return  this.foundation.getFoundationInfo() +
                this.garden.getGardenInfo() +
                this.floor.getFloorInfo() +
                this.roof.getRoofInfo();
    }
    public String getHeading() {
        return this.heading;
    }
    public String getProduct() {
        return  this.foundation.toString() +
                this.garden.toString() +
                this.floor.toString() +
                this.roof.toString();
    }

    private Floor floor;
    private Foundation foundation;
    private Garden garden;
    private Roof roof;
    public void setFoundation(Foundation foundation) {
        this.foundation = foundation; }
    public Foundation getFoundation() { return this.foundatio
```

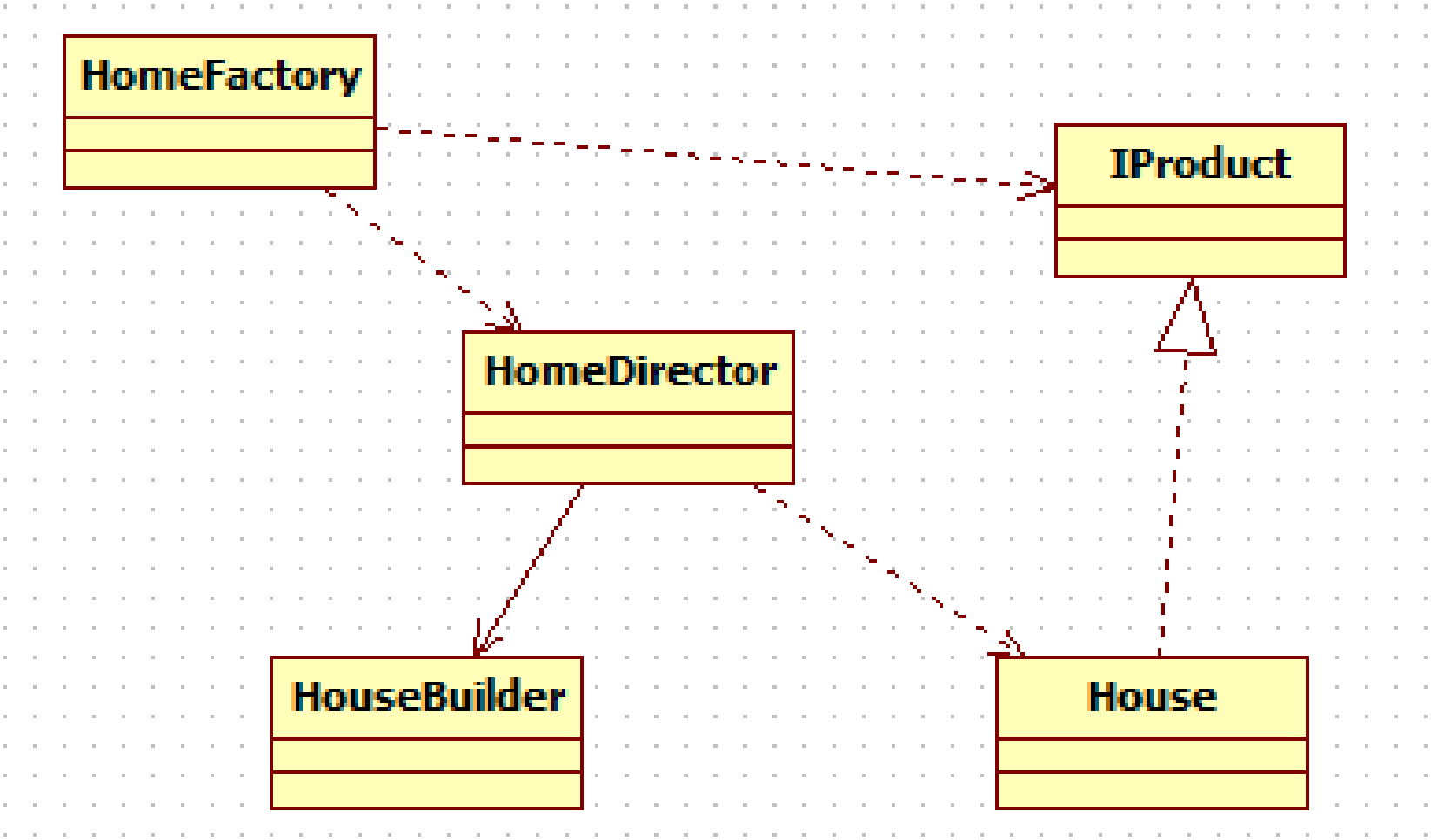
```
class BungalowBuilder implements HouseBuilder {
    private House house;
    public BungalowBuilder() { this.house = new House("Bungalow Model"); }
    public void buildFoundation() {
        this.house.setFoundation(new FullStonesFoundation()); }
    public void buildGarden() { this.house.setGarden(new PondGarden()); }
    public void buildFloor() { this.house.setFloor(new TileFloor()); }
    public void buildRoof() { this.house.setRoof(new AsbestosRoof()); }
    public House getHouse() { return this.house; }
}

class KingfisherBuilder implements HouseBuilder {
    private House house;
    public KingfisherBuilder() { this.house = new House("Kingfisher Model"); }
    public void buildFoundation() { this.house.setFoundation(new MixedFoundation()); }
    public void buildGarden() { this.house.setGarden(new PlainGarden()); }
    public void buildFloor() { this.house.setFloor(new TerrazzoFloor()); }
    public void buildRoof() { this.house.setRoof(new ClayTileRoof()); }
    public House getHouse() { return this.house; }
}
```

```
class NestBuilder implements HouseBuilder {
    private House house;
    public NestBuilder() { this.house = new House("Nest Model"); }
    public void buildFoundation() {
        this.house.setFoundation(new FullStonesFoundation()); }
    public void buildGarden() { this.house.setGarden(new PondGarden()); }
    public void buildFloor() { this.house.setFloor(new TileFloor()); }
    public void buildRoof() { this.house.setRoof(new AsbestosRoof()); }
    public House getHouse() { return this.house; }
}

class IglooBuilder implements HouseBuilder {
    private House house;
    public IglooBuilder() { this.house = new House("Igloo Model"); }
    public void buildFoundation() {
        this.house.setFoundation(new FullStonesFoundation()); }
    public void buildGarden() { this.house.setGarden(new PlainGarden()); }
    public void buildFloor() { this.house.setFloor(new TerrazzoFloor()); }
    public void buildRoof() { this.house.setRoof(new AsbestosRoof()); }
    public House getHouse() { return this.house; }
}
```

```
class OfficeBuilder implements HouseBuilder {
    private House house;
    public OfficeBuilder() { this.house = new House("Office Model"); }
    public void buildFoundation() {
        this.house.setFoundation(new FullStonesFoundation()); }
    public void buildGarden() { this.house.setGarden(new NoGarden()); }
    public void buildFloor() { this.house.setFloor(new TileFloor()); }
    public void buildRoof() { this.house.setRoof(new ClayTileRoof()); }
    public House getHouse() { return this.house; }
}
```

```
public class HomeFactory implements AbstractProductFactory {

    public IProduct createProductByModel(String model) {

        HouseDirector director = null;
        if(model.equals("bunglow")) {
            director = new HouseDirector(new BungalowBuilder());
        } else if(model.equals("kingfisher")) {
            director = new HouseDirector(new KingfisherBuilder());
        } else if(model.equals("nest")) {
            director = new HouseDirector(new NestBuilder());
        } else if(model.equals("igloo")) {
            director = new HouseDirector(new IglooBuilder());
        } else if(model.equals("office")) {
            director = new HouseDirector(new OfficeBuilder());
        } else {
            throw new IllegalArgumentException("Invalid Model ...");
        }

        director.constructHouse();
        return director.getHouse();
    }

    public SalesCenter[] getAllSalesCenters() {
        return new SalesCenter[] {
            new HousingSalesCenter("malabe", "malabe"),
            new HousingSalesCenter("kotte", "kotte")
        };
    }
```

Factory with Builder



The 4 Musketeers (Contd …)

- After having a look at catalog, **ISSA** was interested about the **FEATURES** inside all **5 STANDARD HOUSE MODELS** in the catalog. But **NOT** interested in any of the 5 STANDARD HOUSE MODELS.
- **ISSA** inquired from **SIDA** “Can I preview a **CUSTOM** house model of my own built from these **FEATURES**, other than these standard models?”

The 4 Musketeers (Contd …)

- **SIDA** consulted **WIMA** to change the website to support **CUSTOMIZABLE** house models.
- After having several looks on his 2nd Runner Up Trophy, **WIMA** did the following fix on top of the existing code base.

```
class FoundationManager {
    public static Foundation getFoundationByType(String foundationType) {
        if(foundationType.equals("mixed")) {
            return new MixedFoundation();
        } else if(foundationType.equals("stones")) {
            return new FullStonesFoundation();
        } else {
            throw new IllegalArgumentException("Invalid Foundation Type ...");
        }
    }
}
class GardenManager {
    public static Garden getGardenByType(String gardenType) {
        if(gardenType.equals("pond")) {
            return new PondGarden();
        } else if(gardenType.equals("plain")) {
            return new PlainGarden();
        } else if(gardenType.equals("empty")) {
            return new NoGarden();
        } else {
            throw new IllegalArgumentException("Invalid Foundation Type ...");
        }
    }
}
```

```
class FloorManager {
    public static Floor getFloorByType(String floorType) {
        if(floorType.equals("tile")) {
            return new TileFloor();
        } else if(floorType.equals("terrazzo")) {
            return new TerrazzoFloor();
        } else {
            throw new IllegalArgumentException("Invalid Floor Type ...");
        }
    }
}
class RoofManager {
    public static Roof getRoofByType(String roofType) {
        if(roofType.equals("asbestos")) {
            return new AsbestosRoof();
        } else if(roofType.equals("claytile")) {
            return new ClayTileRoof();
        } else {
            throw new IllegalArgumentException("Invalid Roof Type ...");
        }
    }
}
```

```
class CustomHouseBuilder {
    private House house;
    public CustomHouseBuilder() { this.house = new House("Custom House"); }
    public CustomHouseBuilder buildFoundation(String foundationType) {
        this.house.setFoundation(
            FoundationManager.getFoundationByType(foundationType)
        );
        return this;
    }
    public CustomHouseBuilder buildGarden(String gardenType) {
        this.house.setGarden(
                GardenManager.getGardenByType(gardenType)
            );
        return this;
    }
    public CustomHouseBuilder buildFloor(String floorType) {
        this.house.setFloor(
                FloorManager.getFloorByType(floorType)
            );
        return this;
    }
    public CustomHouseBuilder buildRoof(String roofType) {
        this.house.setRoof(
                RoofManager.getRoofByType(roofType)
            );
        return this;
    }
    public House getHouse() {
        return this.house;
    }
}
```

```
class CustomHouseDirector {
    private CustomHouseBuilder houseBuilder;
    public CustomHouseDirector(CustomHouseBuilder customHouseBuilder,
            String customModelInfo) {
        this.houseBuilder = customHouseBuilder;
        //e.g. Custom House Model => stones:pond:tile:asbestos
        String foundationType = this.getFoundationType(customModelInfo);
        String gardenType = this.getGardenType(customModelInfo);
        String floorType = this.getFloorType(customModelInfo);
        String roofType = this.getRoofType(customModelInfo);

        this.houseBuilder
            .buildFoundation(foundationType)
            .buildGarden(gardenType)
            .buildFloor(floorType)
            .buildRoof(roofType);
    }
    public House getHouse() {
        return this.houseBuilder.getHouse();
    }

    private String getFoundationType(String customModelInfo) {
        // TODO: Implement some String logic here
        return null;
    }
    private String getGardenType(String customModelInfo) {
        // TODO: Implement some String logic here
        return null;
    }
    private String getFloorType(String customModelInfo) {
        // TODO: Implement some String logic here
```

```
public class HomeFactory implements AbstractProductFactory {

    public IProduct createProductByModel(String model) {

        boolean isACustomHouseModel = false;
        HouseDirector director = null;
        if(model.equals("bunglow")) {
            director = new HouseDirector(new BungalowBuilder());
        } else if(model.equals("kingfisher")) {
            director = new HouseDirector(new KingfisherBuilder());
        } else if(model.equals("nest")) {
            director = new HouseDirector(new NestBuilder());
        } else if(model.equals("igloo")) {
            director = new HouseDirector(new IglooBuilder());
        } else if(model.equals("office")) {
            director = new HouseDirector(new OfficeBuilder());
        } else {
            isACustomHouseModel = true;
        }

        if(!isACustomHouseModel) {
            director.constructHouse();
            return director.getHouse();
        } else {
            //e.g. Custom House Model => stones:pond:tile:asbestos
            CustomHouseDirector customDirector =
                new CustomHouseDirector(new CustomHouseBuilder(), model);
            return customDirector.getHouse();
        }
    }
```

The 4 Musketeers (Contd …)

- **ISSA** managed to come up with a **CUSTOMIZED** house model he wanted.
- The business went well. **SIDA** and **RASI** settled all their debts and became billionaires.
- **ISSA** was satisfied about the house he's got.
- **WIMA** got well paid for all the work he did.

The 4 Musketeers (Contd …)

- And they all lived happily ever after.

Moral of the Story …

- When to use design patterns and when not?
- What is the difference between factory method and factory design pattern?
- What is the difference between factory design pattern and abstract factory design pattern?
- Is it necessary to go for a factory design pattern when you have more than one product?

Moral of the Story (Contd …)

- Why factory is more suitable than builder based on the given problem?
- When to use abstract classes and when to use interfaces within your factories?
- Why builder is more suitable than factory based on the given problem?
- How to evolve your code with design patterns, according to changing business requirements?
- How to come up with a hybrid design by integrating multiple design patterns?

Q & A

Singleton

RESTRICTS *"object creation"* from a class to **"ONLY ONE OBJECT"**.

Singleton - Example

```
import java.net.InetAddress;
import java.util.ArrayList;
import java.util.List;

public class LoadBalancer {
    private List<InetAddress> serverList = new ArrayList<InetAddress>();
    private int counter = 0;

    private void loadServerConfiguation() {
        // TODO: Add ips of all servers to serverList here
    }

    public LoadBalancer() { this.loadServerConfiguation(); }

    public InetAddress getNextServer() {
        // We use round robin here
        counter++;
        return serverList.get((serverList.size() % counter) - 1);
    }

    public void printAllServerInfo() {
        // Print the list of servers here ...
    }

}
```

Why need a Singleton???

- We have an instance variable; **“counter”** which we use to implement round robin.
- We have another instance variable **serverList**
- **counter** and **serverList** MUST have SAME VALUE in any LoadBalancer object reference used in any code block of our application
- If there aren’t any instance variables no need of Singleton, can have a class with static methods.

Step 1 – Private Constructor

```
import java.net.InetAddress;
import java.util.ArrayList;
import java.util.List;

public class LoadBalancer {
    private List<InetAddress> serverList = new ArrayList<InetAddress>();
    private int counter = 0;

    private void loadServerConfiguation() {
        // TODO: Add ips of all servers to serverList here
    }

    private LoadBalancer() {
        this.loadServerConfiguation();
    }

    public InetAddress getNextServer() {
        // We use round robin here
        counter++;
        return serverList.get((serverList.size() % counter) - 1);
    }
}
```

Step 2 – Static Attribute

```
import java.net.InetAddress;
import java.util.ArrayList;
import java.util.List;

public class LoadBalancer {
    private List<InetAddress> serverList = new ArrayList<InetAddress>();
    private int counter = 0;

    private static LoadBalancer instance;

    private void loadServerConfiguation() {
        // TODO: Add ips of all servers to serverList here
    }

    private LoadBalancer() {
        this.loadServerConfiguation();
    }

    public InetAddress getNextServer() {
        // We use round robin here
        counter++;
        return serverList.get((serverList.size() % counter) - 1);
    }
}
```

Step 3 - Static Method

```
import java.net.InetAddress;
import java.util.ArrayList;
import java.util.List;

public class LoadBalancer {
    private List<InetAddress> serverList = new ArrayList<InetAddress>();
    private int counter = 0;

    private static LoadBalancer instance;

    public static LoadBalancer getInstance() {
        if(instance == null) {
            instance = new LoadBalancer();
        }
        return instance;
    }

    private void loadServerConfiguation() {
        // TODO: Add ips of all servers to serverList here
    }

    private LoadBalancer() {
        this.loadServerConfiguation();
    }

    public InetAddress getNextServer() {
        // We use round robin here
        counter++;
        return serverList.get((serverList.size() % counter) - 1);
```

Step 4 – Sync, **ONLY** if required

```
public class LoadBalancer {
    private List<InetAddress> serverList = new ArrayList<InetAddress>();
    private int counter = 0;

    private static Object lock1 = new Object();
    private static Object lock2 = new Object();
    private static LoadBalancer instance;

    private LoadBalancer() { this.loadServerConfiguation(); }

    public static LoadBalancer getInstance() {
        synchronized (lock1) {
            if(instance == null) {
                instance = new LoadBalancer();
            }
        }
        return instance;
    }

    public InetAddress getNextServer() {
        // We use round robin here
        synchronized (lock2) {
            counter++;
        }
        return serverList.get((serverList.size() % counter) - 1);
    }

    public void printAllServerInfo() {
        // Print the list of servers here ...
    }
```

Why use **lock1** and **lock2**?

- Why not use "**static synchronized methods**" OR "**synchronized(this)**" ?????
- LoadBalancer class need not to be fully locked, since printAllServerInfo() method don't need synchronization, even in case of multiple threads.

Using Singleton

```
class SomeClientClass {
    public void someMethod() {
        LoadBalancer balancer = LoadBalancer.getInstance();
        InetAddress server = balancer.getNextServer();
        //... more code here ...
    }
}

class SomeClientClass2 {
    public void someMethod2() {
        LoadBalancer balancer = LoadBalancer.getInstance();
        balancer.printAllServerInfo();
    }
}
```

What if **severList** changes even after initialization????

- Assume you have a public method to **setServerList()**
- 2 Options;
  - Lock **setServerList()**, **getNextServer()** and **printAllServerInfo()** using **lock2**
  - **OR** use “static synchronized methods” or “synchronized(this)”
- **Which is more suitable????**

```
public class LoadBalancer {
    private List<InetAddress> serverList = new ArrayList<InetAddress>();
    private int counter = 0;
    private static Object lock1 = new Object();
    private static Object lock2 = new Object();
    private static LoadBalancer instance;

    private LoadBalancer() { this.loadServerConfiguation(); }
    public static LoadBalancer getInstance() {
        synchronized (lock1) {
            if(instance == null) { instance = new LoadBalancer(); }
        }
        return instance;
    }
    public void setServerList(List<InetAddress> newServerList) {
        synchronized (lock2) {
            this.serverList = newServerList;
        }
    }
    public InetAddress getNextServer() {
        // We use round robin here
        InetAddress nextServer;
        synchronized (lock2) {
            counter++;
            nextServer = serverList.get((serverList.size() % counter) - 1);
        }
        return nextServer;
    }
    public void printAllServerInfo() {
        synchronized (lock2) {
            // Print the list of servers here ...
        }
```

Prototype

CREATE objects by **CLONING** an existing object

Why Clone???

- JVM's Perspective
  - Higher performance than creating objects via constructors (i.e. with **new** keyword)
- Java Developer's Perspective
  - Can create objects having **COMPLEX** structures by using already such **EXISTING** objects; **WITHOUT** any knowledge about the construction
    - E.g. Creating a COMPLEX woman by using another COMPLEX woman, without knowing the COMPLEXITY of woman kind (previous Slide)

Example – Graphics App

```
class ColorSettings {
    private int rValue;
    private int gValue;
    private int bValue;
    private int alphaValue;

    public ColorSettings(int rValue, int gValue, int bValue, int alphaValue) {
        this.setRValue(rValue);
        this.setGValue(gValue);
        this.setBValue(bValue);
        this.setAlphaValue(alphaValue);
    }

    public void setRValue(int rValue) { this.rValue = rValue; }
    public int getRValue() { return rValue; }
    public void setGValue(int gValue) { this.gValue = gValue; }
    public int getGValue() { return gValue; }
    public void setBValue(int bValue) { this.bValue = bValue; }
    public int getBValue() { return bValue; }
    public void setAlphaValue(int alphaValue) { this.alphaValue = alphaValue; }
    public int getAlphaValue() { return alphaValue; }
}
```

Example – Graphics App

```
class Color {
    private String name;
    private String description;
    private ColorSettings settings;

    public Color(String name,
            String description, ColorSettings settings) {
        this.name = name;
        this.description = description;
        this.settings = settings;
    }
    public int getPixelValue( /* some params here */) {
        //do something by using settings, calculate pixel value
        // e.g.
        settings.getAlphaValue();
        return -1;
    }
    public void alterDarkness(int alpha /* some more params here */) {
        //e.g.
        settings.setAlphaValue(alpha);
        // do darkening
    }
    public void alterLightness(/* some params here */) { /* do lighting */ }

}
```

Without Cloning …

```
class SomeClass
{
    public void someMethod() {
        int rValue = 255;
        int gValue = 0;
        int bValue = 0;
        int alphaValue = 1;
        ColorSettings settings =
            new ColorSettings(rValue, gValue, bValue, alphaValue);

        String colorName = "RED";
        String colorDescription = "Hot as a burning Hot Plate !!!";
        Color redColor = new Color(colorName, colorDescription, settings);
        redColor.alterDarkness(1);
    }
}
```

With Cloning !!!

```
class Color implements Cloneable {
    private String name;
    private String description;
    private ColorSettings settings;

    public Color clone() throws CloneNotSupportedException {
        return (Color)super.clone();
    }

    public Color(String name,
            String description, ColorSettings settings) {
        this.name = name;
        this.description = description;
        this.settings = settings;
    }
    public int getPixelValue( /* some params here */) {
        //do something by using settings, calculate pixel value
        // e.g.
        settings.getAlphaValue();
        return -1;
    }
    public void alterDarkness(int alpha /* some more params here */) {
        //e.g.
        settings.setAlphaValue(alpha);
        // do darkening
```

With Cloning !!!

```
public class ColorManager {

    private static Map<String, Color> standardColorsMap
        = new HashMap<String, Color>();

    static {
        // Initialize all colors here ...
        ColorSettings redSettings = new ColorSettings(255, 0, 0, 1);
        Color red = new Color("RED", "Hot as fire !!!", redSettings);
        standardColorsMap.put("RED", red);
        ColorSettings greenSettings = new ColorSettings(0, 255, 0, 1);
        Color green = new Color("GREEN", "Like a tree leaf !!!", greenSettings);
        standardColorsMap.put("GREEN", green);
        ColorSettings blueSettings = new ColorSettings(0, 0, 255, 1);
        Color blue = new Color("BLUE", "Like deep sea !!!", blueSettings);
        standardColorsMap.put("BLUE", blue);
    }

    public static Color getColor(String colourName) {
        try {
            return standardColorsMap.get(colourName).clone();
        } catch(CloneNotSupportedException clex) {
            return null;
        }
    }
}
```

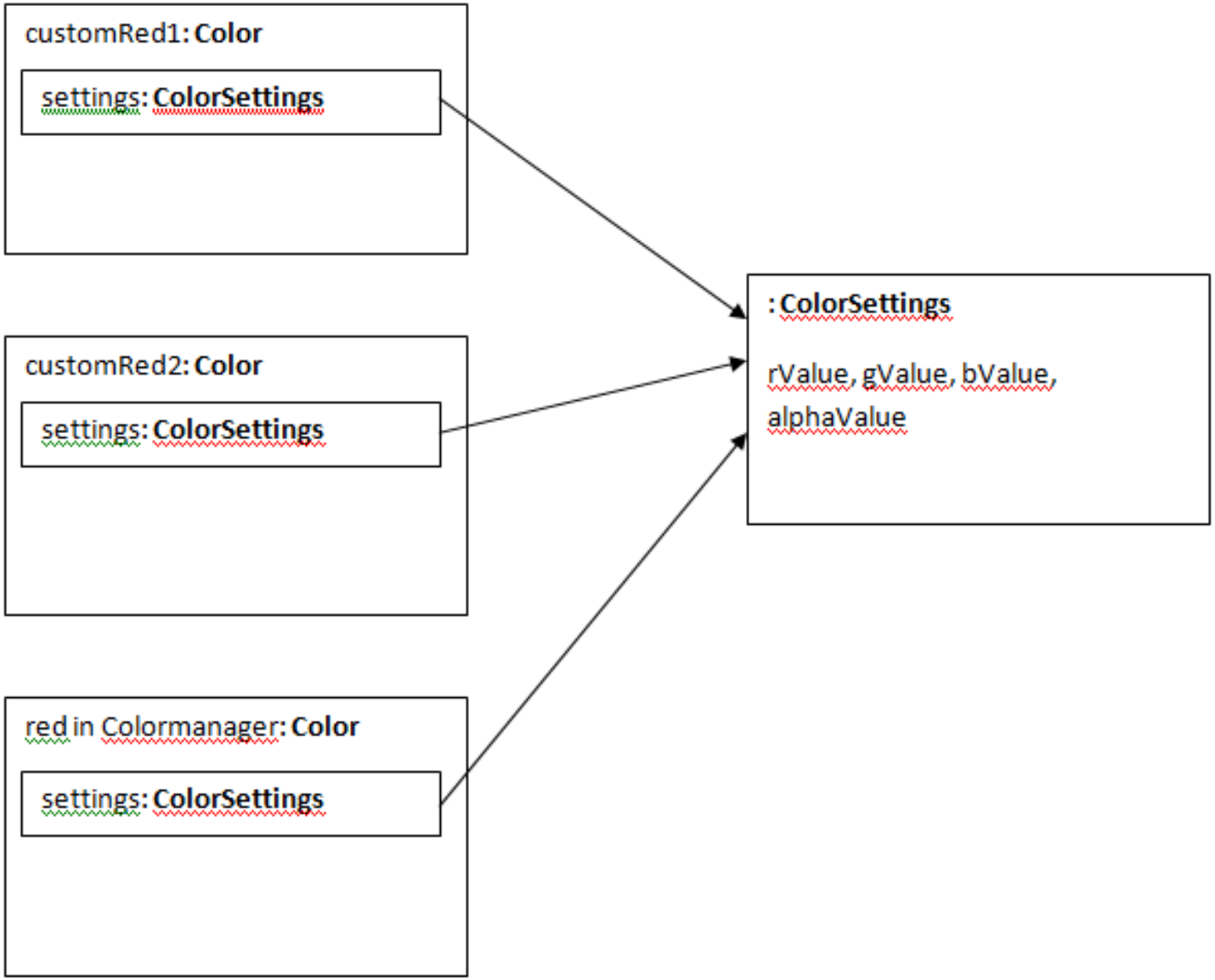
With Cloning !!!

```
class SomeClass
{
    public void someMethod() {

        Color customRed1 = ColorManager.getColor("RED");
        customRed1.alterDarkness(1);

        Color customRed2 = ColorManager.getColor("RED");
        customRed2.alterDarkness(7);
    }
}
```

Is It Correct ????

```
class Color implements Cloneable {
    private String name;
    private String description;
    private ColorSettings settings;

    public Color clone() throws CloneNotSupportedException {
        return (Color)super.clone();
    }

    public Color(String name,
            String description, ColorSettings settings) {
        this.name = name;
        this.description = description;
        this.settings = settings;
    }
    public int getPixelValue( /* some params here */) {
        //do something by using settings, calculate pixel value
        // e.g.
        settings.getAlphaValue();
        return -1;
    }
    public void alterDarkness(int alpha /* some more params here */) {
        //e.g.
        settings.setAlphaValue(alpha);
        // do darkening
```

Problem of Shallow Copies



Problems with Shallow Copies …

- Reason
  - ColorSettings is a **“REFERENCE TYPE”** (i.e. not a primitive data type)

  AND
  - is **“NOT IMMUTABLE”**

Deep Copies

```
class ColorSettings implements Cloneable {
    private int rValue;
    private int gValue;
    private int bValue;
    private int alphaValue;

    public ColorSettings(int rValue, int gValue, int bValue, int alphaValue) {
        this.setRValue(rValue);
        this.setGValue(gValue);
        this.setBValue(bValue);
        this.setAlphaValue(alphaValue);
    }

    public ColorSettings clone() throws CloneNotSupportedException {
        return (ColorSettings)super.clone();
    }

    public void setRValue(int rValue) { this.rValue = rValue; }
    public int getRValue() { return rValue; }
    public void setGValue(int gValue) { this.gValue = gValue; }
    public int getGValue() { return gValue; }
```

Deep Copies

```
class Color implements Cloneable {
    private String name;
    private String description;
    private ColorSettings settings;

    public Color clone() throws CloneNotSupportedException {
        Color newColor = (Color)super.clone();
        newColor.settings = (ColorSettings)this.settings.clone();
        return newColor;
    }

    public Color(String name,
            String description, ColorSettings settings) {
        this.name = name;
        this.description = description;
        this.settings = settings;
    }
    public int getPixelValue( /* some params here */) {
        //do something by using settings, calculate pixel value
        // e.g.
        settings.getAlphaValue();
```

Prototype - Quiz

- Why ColorManager need **NOT** necessarily be a **Singleton** class?
  - In LoadBalancer value of **"counter" CHANGED** everytime, but in ColorManager standard colors are **FIXED** after being initialized
- Why is factory, abstract factory, builder; **NOT** suitable for this problem?
  - Factories **CREATE** objects from a **SELECTED CLASS** based on the **GIVEN CASE**
  - Prototypes are **PRE-CREATED** objects from the **SAME CLASS**(i.e. Color), used as **TEMPLATES** for creating **CUSTOMIZED** objects (i.e. customRed1)

Q & A