

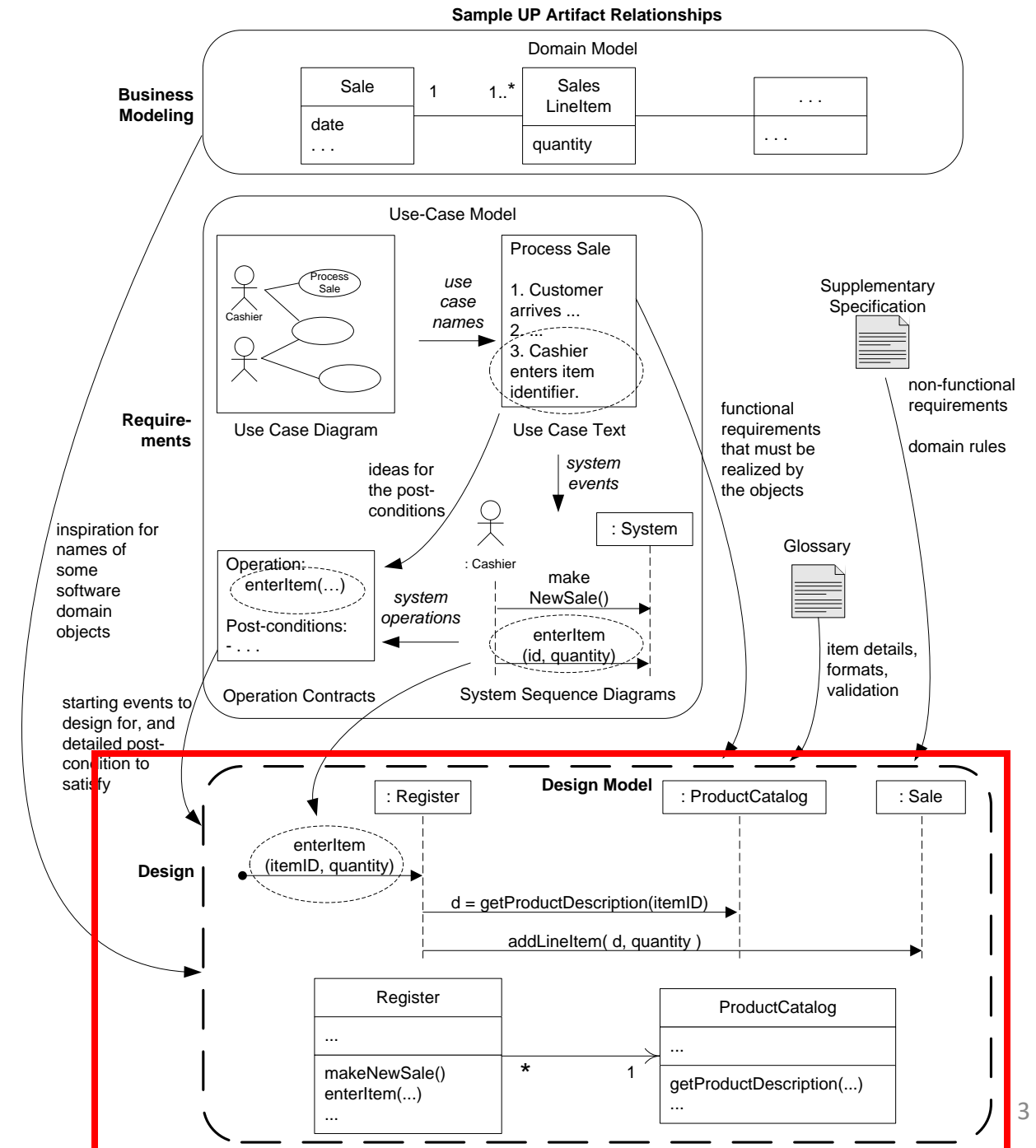
Protected Variations and Polymorphism




Topics

- GRASP
 - Protected Variations
 - Polymorphism
- Examples
 - Password Generation Algorithms
 - Using External Services
 - Painting Objects

Artifacts Overview



GRASP - General Responsibility Assignment Software Patterns (or Principles)



Recall previous presentations

- GRASP is a methodical **approach to OO Design**
 - Based on principles/patterns for **responsibilities assignment**
 - Helps to understand the fundamentals of object design
 - Allows to apply design reasoning in a methodical, rational, and understandable way
- In UML, the design of Interaction Diagrams (e.g. class and sequence diagrams) is a means to consider and represent responsibilities
 - When designing, you decide which responsibilities to assign to each object

GRASP

- Pure Fabrication
- Controller
- Information Expert
- Creator
- High Cohesion
- Low Coupling
- Polymorphism *
- Indirection
- Protected Variation *

* Patterns addressed in this class

Protected Variations

Motivation for the Problem

Motivating the Need For Protected Variations

- Consider the two example scenarios presented in the following slides
 - **Scenario 1: Password Generation Algorithms**
 - **Scenario 2: Obtaining Geographic Areas**
- Generalize both problems/scenarios to a more universal one
- Analyze the proposed solution for the generic problem and also for the example scenarios

Scenario 1 – Password Generation

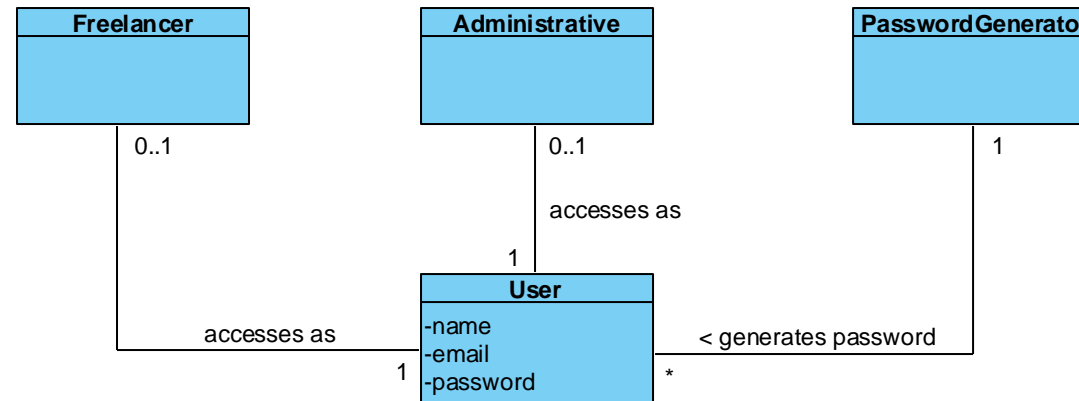
For a given application, the **user's initial passwords** must:

- Be **generated by the system**;
- Using an **external password generator algorithm** (i.e. designed by a third party); and
- Configured only **when deploying the system**.

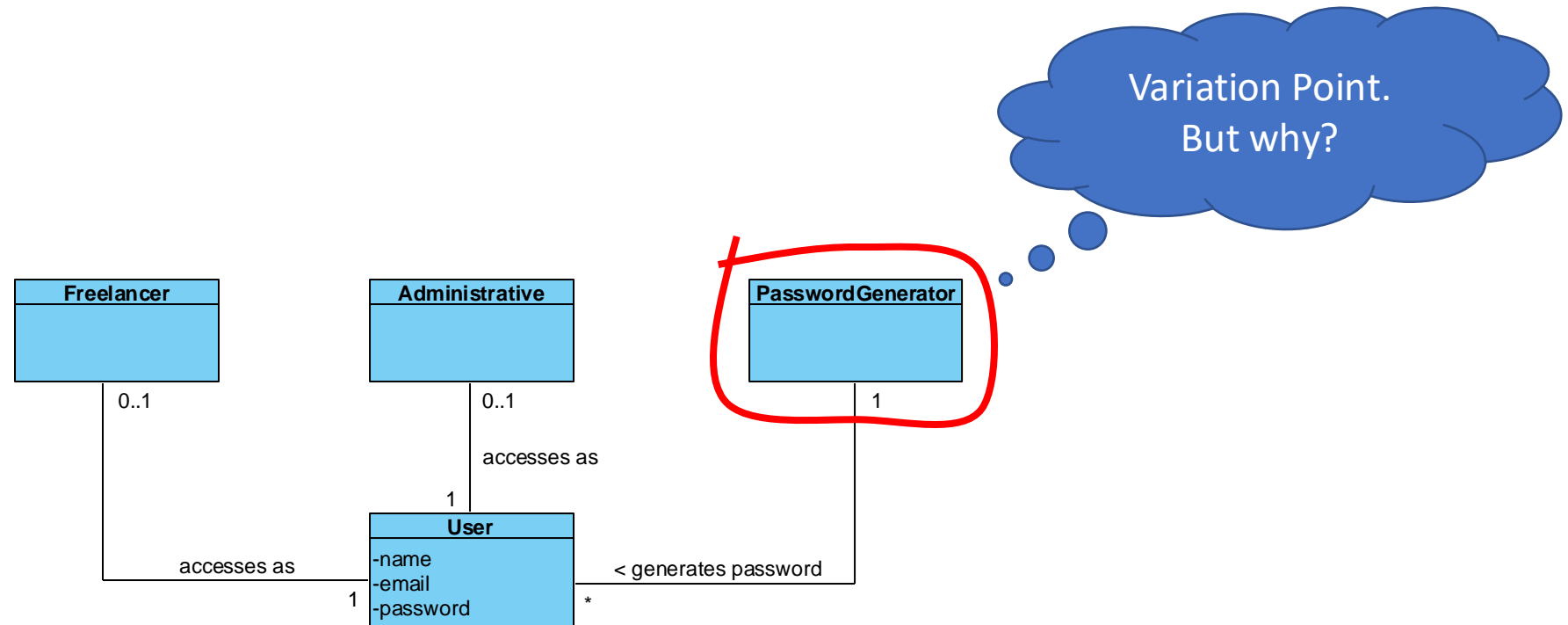


**HOW TO
SUPPORT/DESIGN
THIS?**

Scenario 1 – (Partial) Domain Model (1/2)

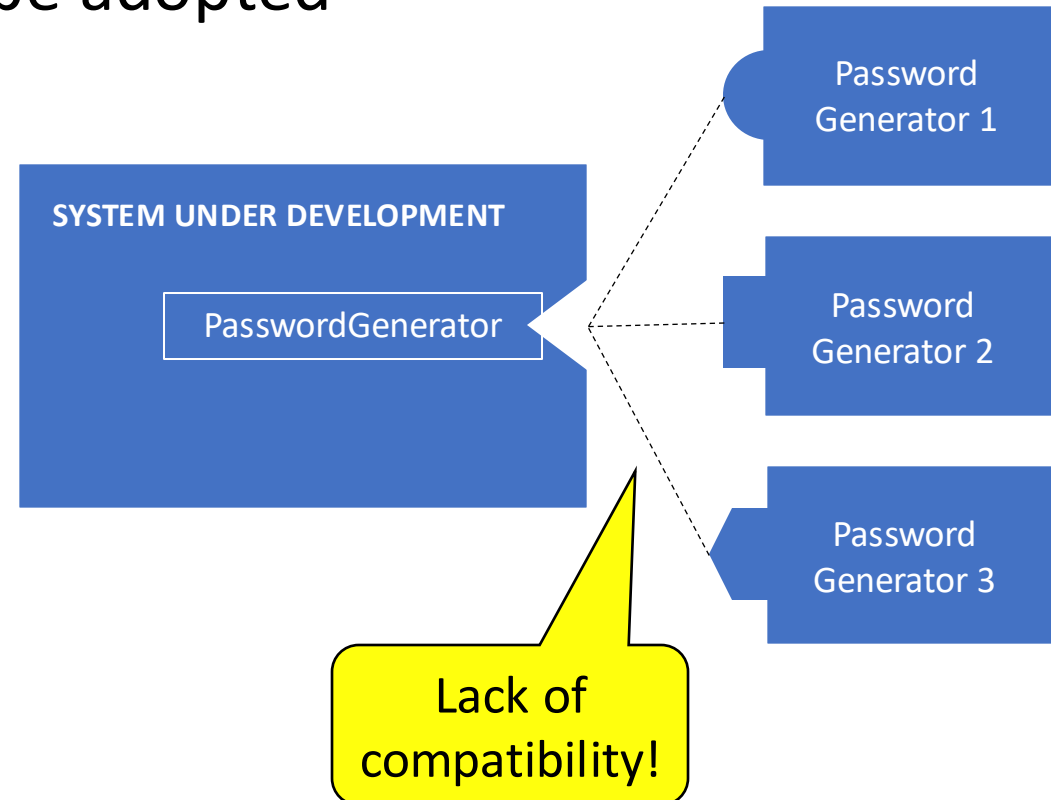


Scenario 1 – (Partial) Domain Model (2/2)



Scenario 1 – Variation Point

- A.k.a. Instability Point
- Several algorithms may exist and be adopted
- Possibly different regarding the:
 - Form of invocation
 - Input data
 - Output data
 - Process / flow
 - ...



Scenario 1 – Coding (with current knowledge)

```
private String generatePassword(String name,  
                                String email) {  
    String pwd;  
  
    if (this.pwdGenerator instanceof XXX) {  
        //...  
    }  
    else if (this.pwdGenerator instanceof YYY) {  
        //...  
    }  
    else if (this.pwdGenerator instanceof ZZZ) {  
        //...  
    }  
  
    return pwd;  
}
```

Variation
Point

- To which class does this code belong?
 - Controller? Why?
 - Another class (e.g. PasswordGenerator)?
- Does this code do what is needed?
- Is it “nice” or “pretty”?
- What happens if more rules are needed?

Too many engineering problems to
maintain the software!

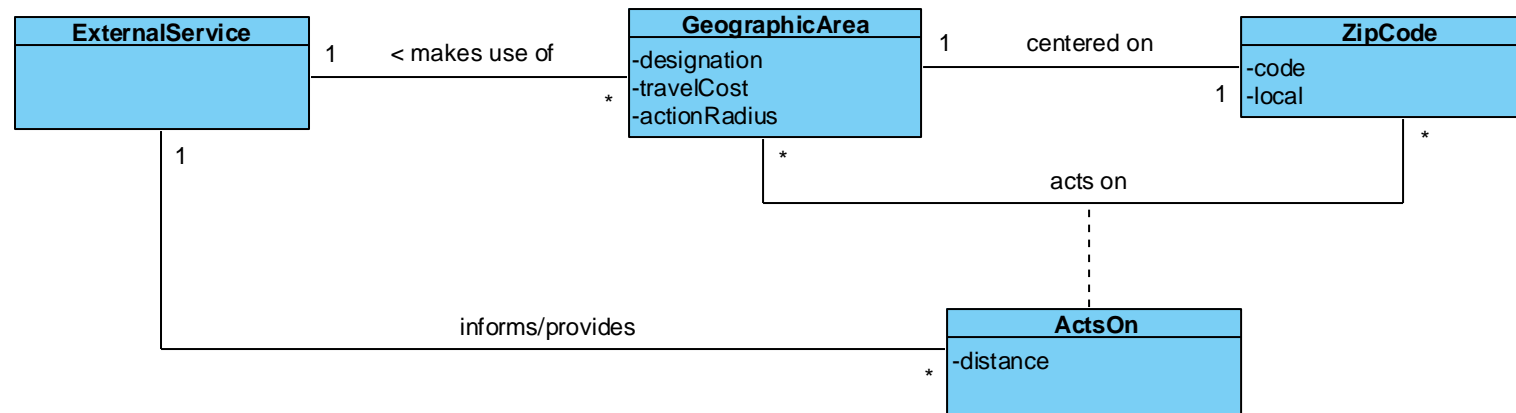
Scenario 2 – Obtaining Geographic Areas

- For an existing project, each **geographic area** is centered on a single zip code (e.g. “4249-015”) and operates on all postal addresses whose zip code is within its radius of action (e.g. 5 km).
- To obtain the zip codes within the radius of action of another zip code, **an external service is used**
- The system must **support different external services** and the one to be used is **set by configuration at the time of deployment**
- If a zip code is covered by more than one geographic area, the one with the shortest distance is chosen

A red starburst graphic with multiple points, containing the text 'HOW TO SUPPORT/DESIGN THIS?' in white capital letters.

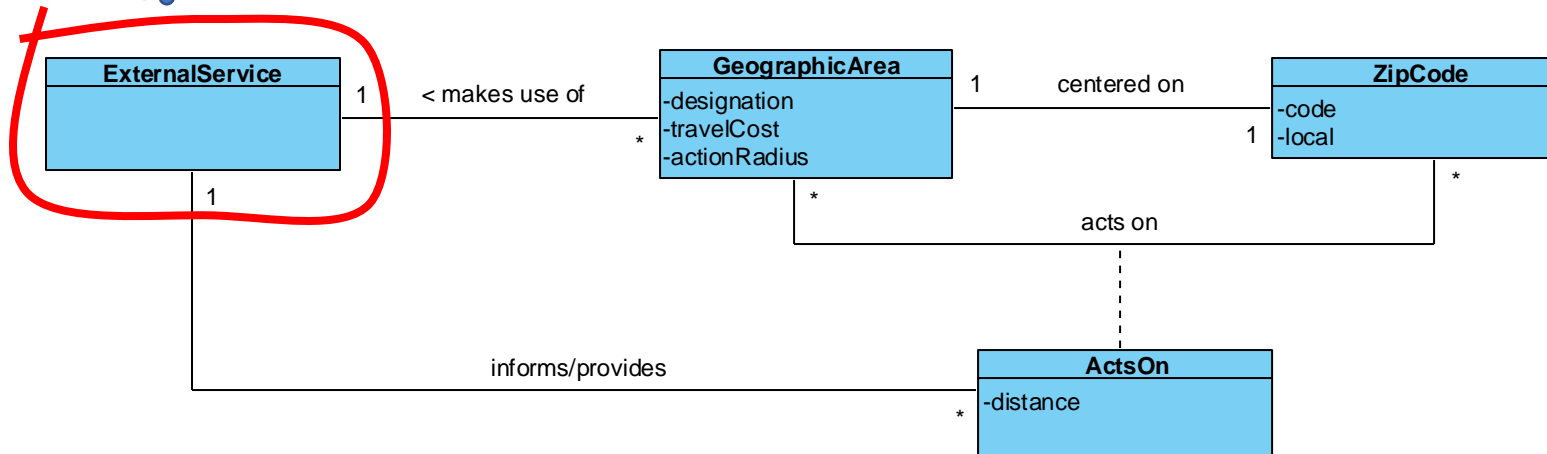
HOW TO
SUPPORT/DESIGN
THIS?

Scenario 2 – (Partial) Domain Model (1/2)



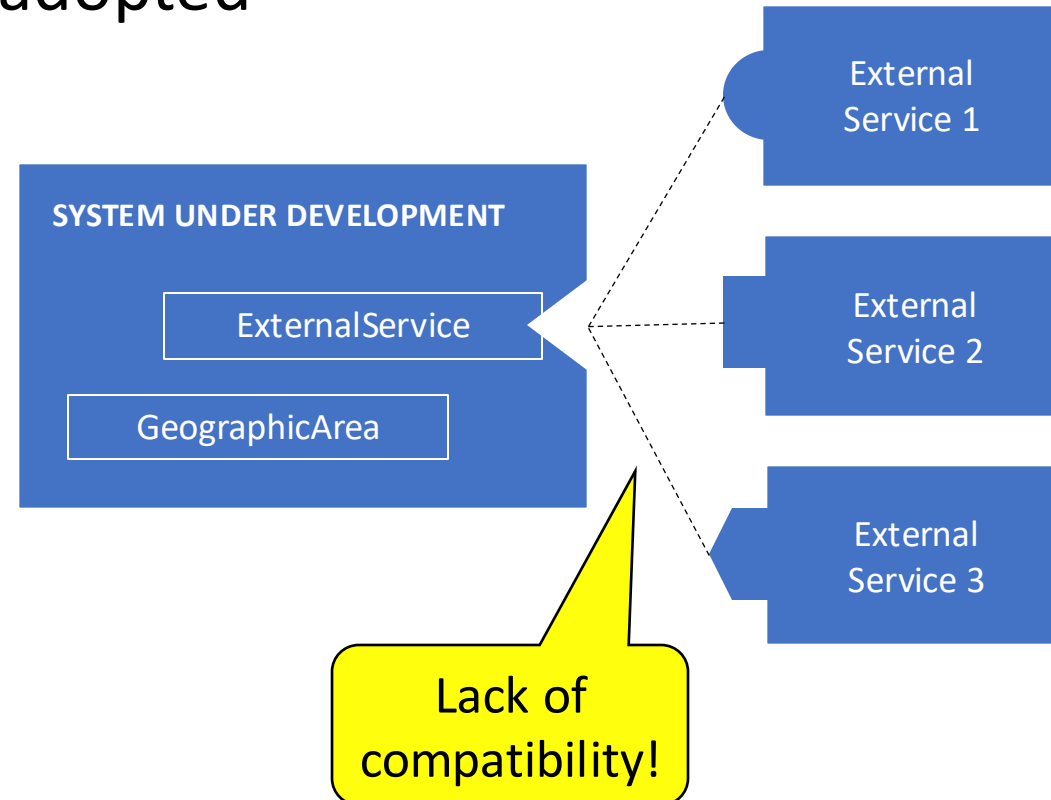
Scenario 2 – (Partial) Domain Model (2/2)

Variation Point.
But why?



Scenario 2 – Variation Point

- A.k.a. Instability Point
- Several services may exist and be adopted
- Possibly different regarding the:
 - Form of invocation
 - Input data
 - Output data
 - Process / flow
 - ...



Scenario 2 – Coding (with current knowledge)

```
private List<ActsOn> computeActsOn (ZipCode base,
                                     float radius) {
    List<ActsOn> listOfActsOn;

    if (this.externalService instanceof XXX) {
        //...
    }
    else if (this.externalService instanceof YYY) {
        //...
    }
    else if (this.externalService instanceof ZZZ) {
        //...
    }

    return listOfActsOn;
}
```

Variation
Point

- To which class does this code belong?
 - GeographicArea? Why?
 - Another class?
- Does this code do what is needed?
- Is it “nice” or “pretty”?
- What happens if more rules are needed?

Too many engineering problems to
maintain the software!

Generalizing the Underlying Problem (1/2)

- Possibility of doing the same thing in different ways
 - Known *a priori* by the development team
 - Not yet known to the team
 - Developed by other teams (third parties)
 - in the past; or
 - in the future
- Variation
 - Over time
 - By deployment/installation



Variation Point

Generalizing the Underlying Problem (2/2)

- Possibility of doing the same thing in different ways
 - Known *a priori* by the development team
 - Not yet known to the team
 - Developed by other teams (third parties)
 - in the past; or
 - in the future
- Variation
 - Over time
 - By deployment/installation



Variation Point



**HOW TO HANDLE
VARIATION POINTS?**

GRASP

Protected Variations

Protected Variations

- **Problem**

- How to design objects, components and systems so that variations in these elements do not have an undesirable impact on other elements of the system?

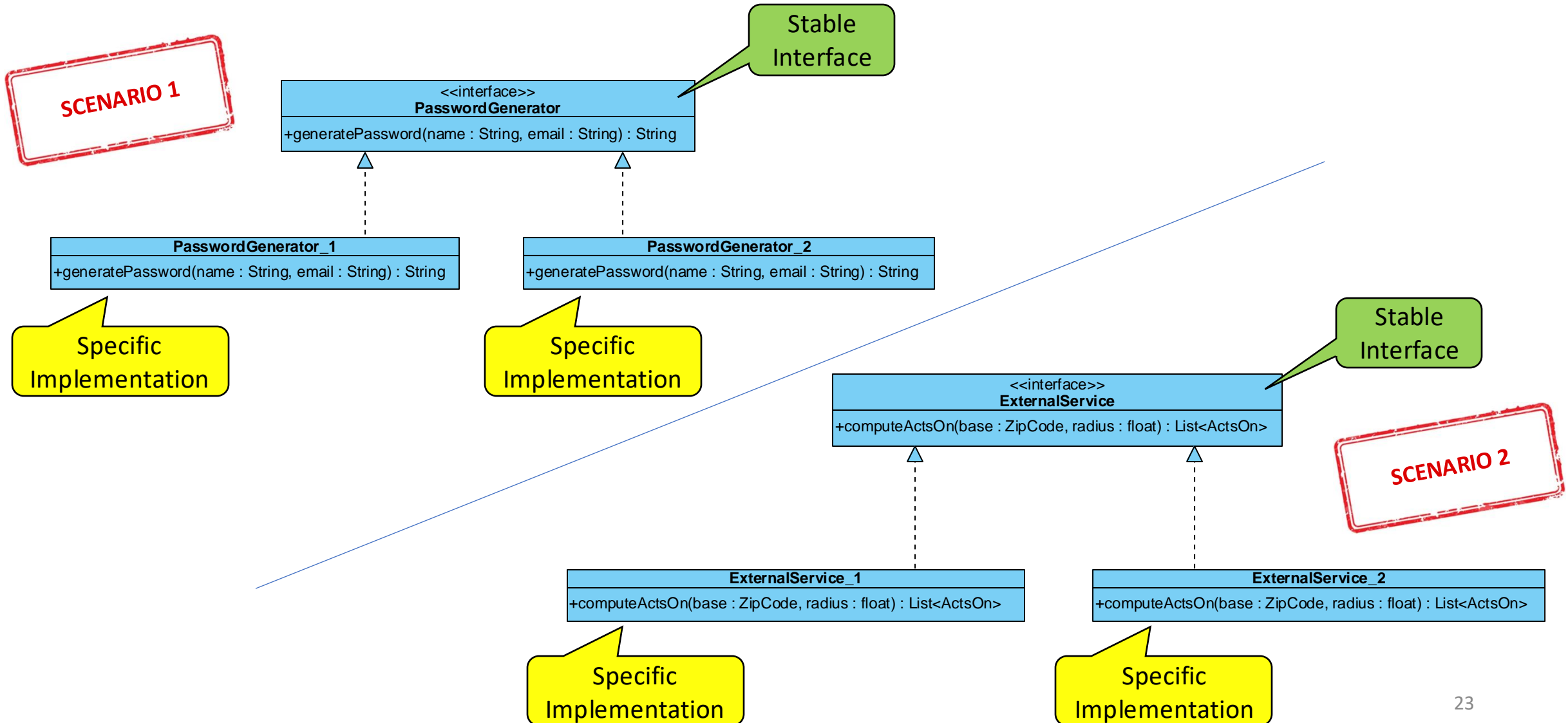
- **Solution**

- Identify predictable points of variation
- Assign responsibilities to create a **stable interface** around these points

Proposed Solution for the Example Scenarios (1/2)

- The **point of variation** (or instability) is the **existence of different interfaces (API)** for:
 - Scenario 1: Password Generator Algorithms
 - Scenario 2: Obtaining Geographic Areas
- Solution
 - Internal objects collaborate with a **stable interface**
 - Scenario 1: `String generatePassword(String name, String email)`
 - Scenario 2: `List<ActsOn> computeActsOn(ZipCode base, float radius)`
 - Specific implementations of the interface hide the variants of different algorithms/services

Proposed Solution for the Example Scenarios (2/2)



Protected Variations – Another Example

- **Scenario 3: Application that paints several distinct objects according to their characteristics**
 - Painting a **Car** involves painting the wheels, the bodywork, avoiding windows, etc.
 - Painting a **Table** implies painting the table legs and the tabletop.
- Point of Instability
 - Each object has its own painting particularities and consequently, a distinct way of painting
 - There are **different painting algorithms**
- How to create a stable interface?
 - Internal objects collaborate with a **stable interface** named **Paintable** that declares the **paint()** method
 - Each interface implementation hides a specific painting algorithm under the **paint()** method

GRASP

Polymorphism

Polymorphism (1/2)

- **Problem**

- How to handle alternatives based on types (classes)?
- How to create pluggable software components?

- **Solution**

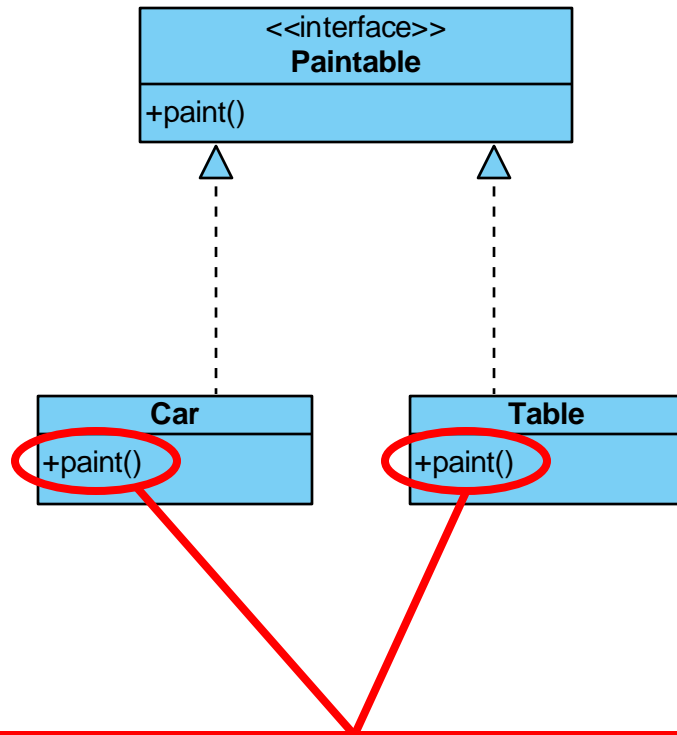
- When alternatives or related behavior vary depending on the type, responsibilities should be assigned to polymorphic operations on such types.

- **Polymorphism** argues that polymorphic operations should be used rather than decisions based on types

Polymorphism (2/2)

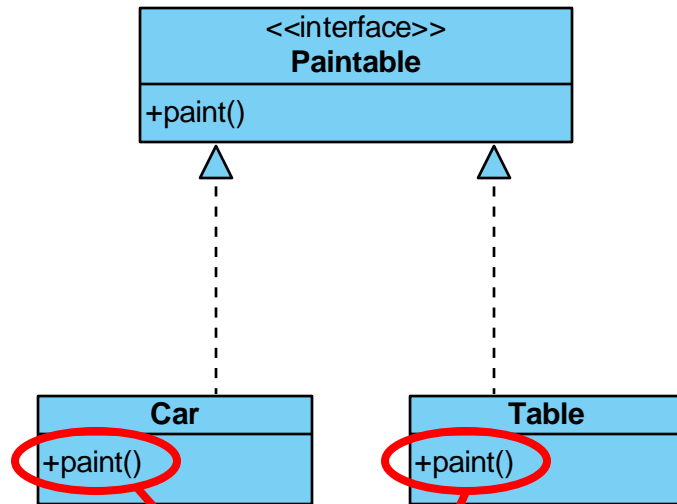
- Target (or cause)
 - Applications with logical and/or behavioral variations typically handled with multiway branch statements (e.g. if-then-else, switch-case)
- Consequences (of not using polymorphism)
 - Makes it difficult to understand and evolve the program
 - A new variation implies modifying the “logic” in several parts of the code
- Polymorphic methods
 - These are methods with the same name and signature/header on different objects, but with different behaviors
 - E.g.: The `paint()` method in `Car` and `Table` classes

Polymorphism Example – Design (1/2)

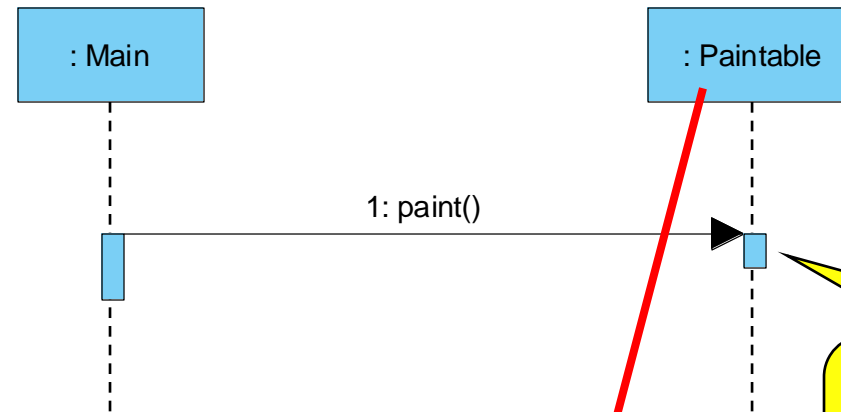


The implementation of the `paint()` method differs from one class to another.

Polymorphism Example – Design (2/2)



The implementation of the `paint()` method differs from one class to another.



It doesn't matter what it does or how it does it. It depends on the `Paintable` implementation.

This is an instance of any class that implements the `Paintable` interface (e.g. `Car`, `Table`).

One cannot detail what happens in `Paintable` when the `paint()` method is invoked, as it depends on the instance type.

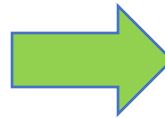
Polymorphism Example – Code

```
public static void main(String[] args) {
    //...
    paintObject(object);
    //...
}

private void paintObject(Object o) {
    if (o instanceof Car) {
        paintCar((Car) o);
    } else if (o instanceof Table) {
        paintTable((Table) o);
    }
}

private void paintCar(Car c) {
    //... Behavior A
}

private void paintTable(Table t) {
    //... Behavior B
}
```



```
public static void main(String[] args) {
    Table t = new Table();
    paintObject(t);

    Car c = new Car();
    paintObject(c);
}

private void paintObject(Paintable p) {
    p.paint();
}
```

```
interface Paintable {
    public void paint();
}
```

```
class Car implements Paintable {
    public void paint() {
        //... Behavior A
    }
}
```

```
class Table implements Paintable {
    public void paint() {
        //... Behavior B
    }
}
```

Polymorphism Application

- It is a fundamental principle when specifying how a system should be organized to handle variations on similar behaviors
 - E.g.: Adding a new class (e.g. `Computer`) that implements the `Paintable` interface, has less impact on the application design than implementing the various algorithms in methods of a single class
- Benefits
 - Required extensions for new variants are easily added
 - New implementations can be introduced without affecting clients

Summary

- We've discussed how to protect a system from variations of implementation from different external services
 - Making use of Polymorphism
 - Thus, achieving Protected Variations
- In the next presentations, you'll see how to implement this using the Adapter Pattern
- By using Java Reflection, you can even provide new behavior for existing code, such as adding a new password generator external service

References & Bibliography

- Larman, Craig; Applying UML and Patterns; Prentice Hall (3rd ed.); ISBN 978-0131489066
- Booch, G. 1994. Object-Oriented Analysis and Design. Redwood City, CA.: Benjamin/Cummings.