Software Engineering Lab

Implementing relationships among classes of analysis pattern languages using aspects

Rosana T. V. Braga

Rodrigo H. R. Marchesini

University of São Paulo

Institute of Mathematics and Computer Science

São Carlos – SP – Brazil

Summary

- Introduction
  - Context
  - Motivation
  - Objectives
- Proposed approach for implementing associations using aspects
- Case study
- Concluding remarks

Artigo publicado no workshop do ECOOP 2009: RAOOL: Workshop on Relations and Associations in Object-Oriented Languages

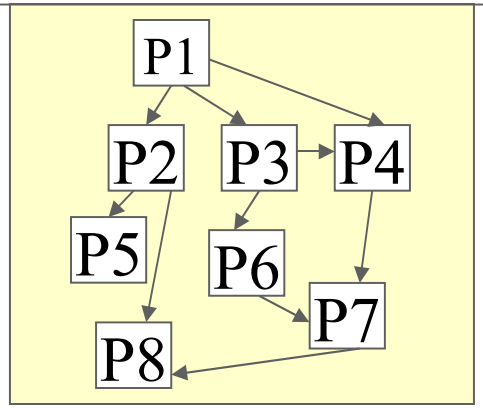
Introduction

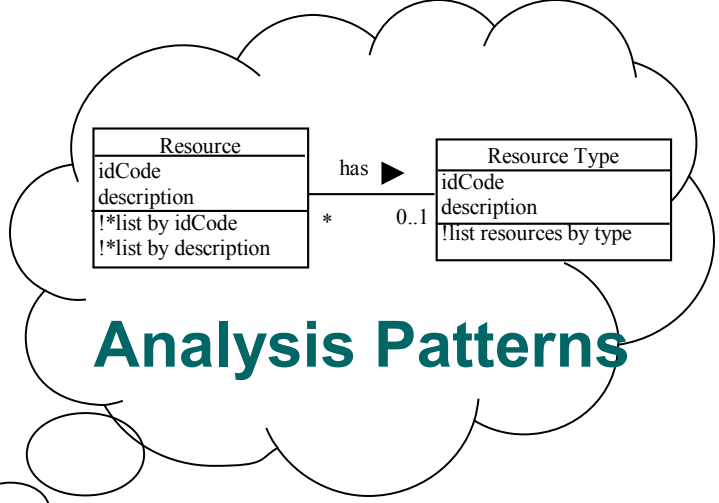
- Software Patterns:
  - Describe solutions to common problems found during software development
  - Patterns: reuse in higher abstraction levels:
    - architectural patterns
    - analysis patterns
    - design patterns
    - organizational patterns
    - coding patterns, etc.
- Analysis Patterns:
  - Propose analysis models that solve problems found during analysis phase (UML)

Introduction

- Pattern Language:
  - Set of inter-related patterns that lead to the complete model of a particular application
    - A structured collection of patterns that support each other to transform requirements into an architecture [Coplien 98].
  - The application of one pattern sets the context for the next pattern to be applied
  - It can be thought of as a way of dividing a general problem and its solution into a number of related problems and the corresponding solutions

Pattern Language

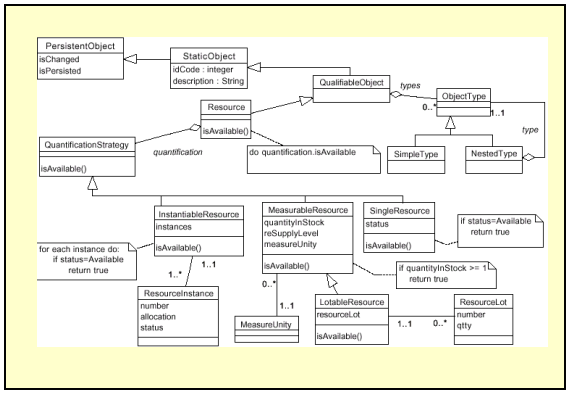




Analysis Patterns

Complete design and implementation of analysis patterns and their relationships

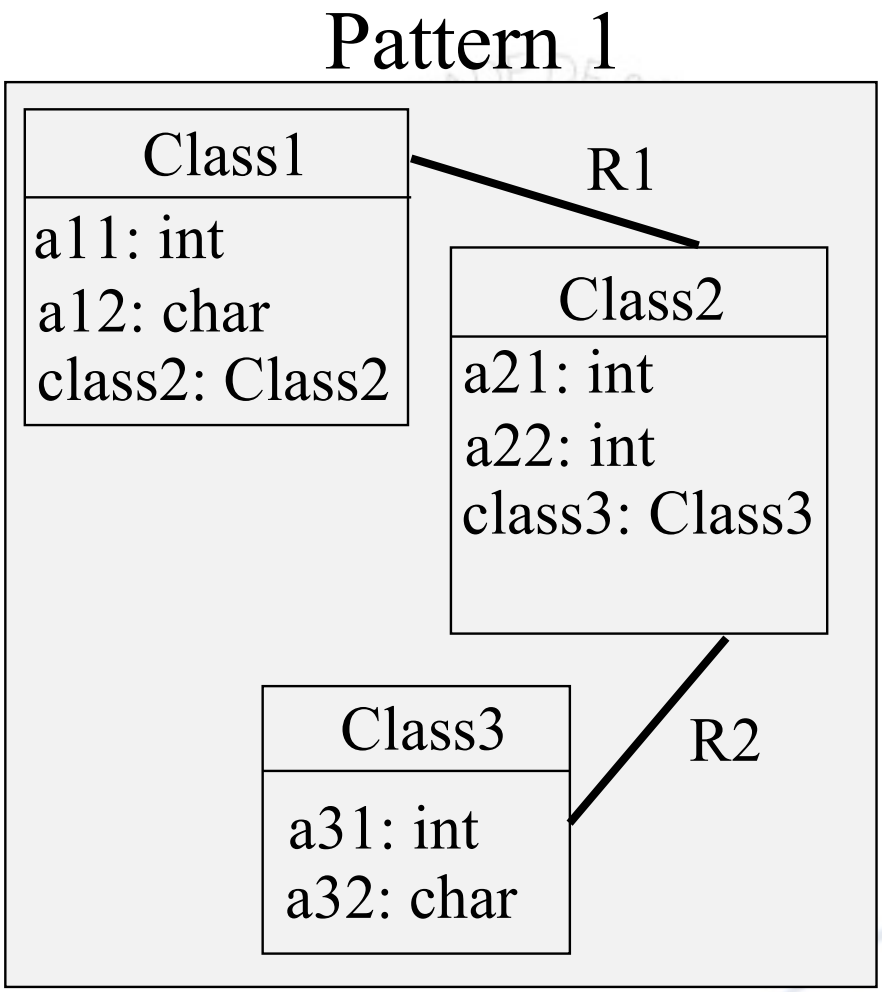
Framework



Motivation

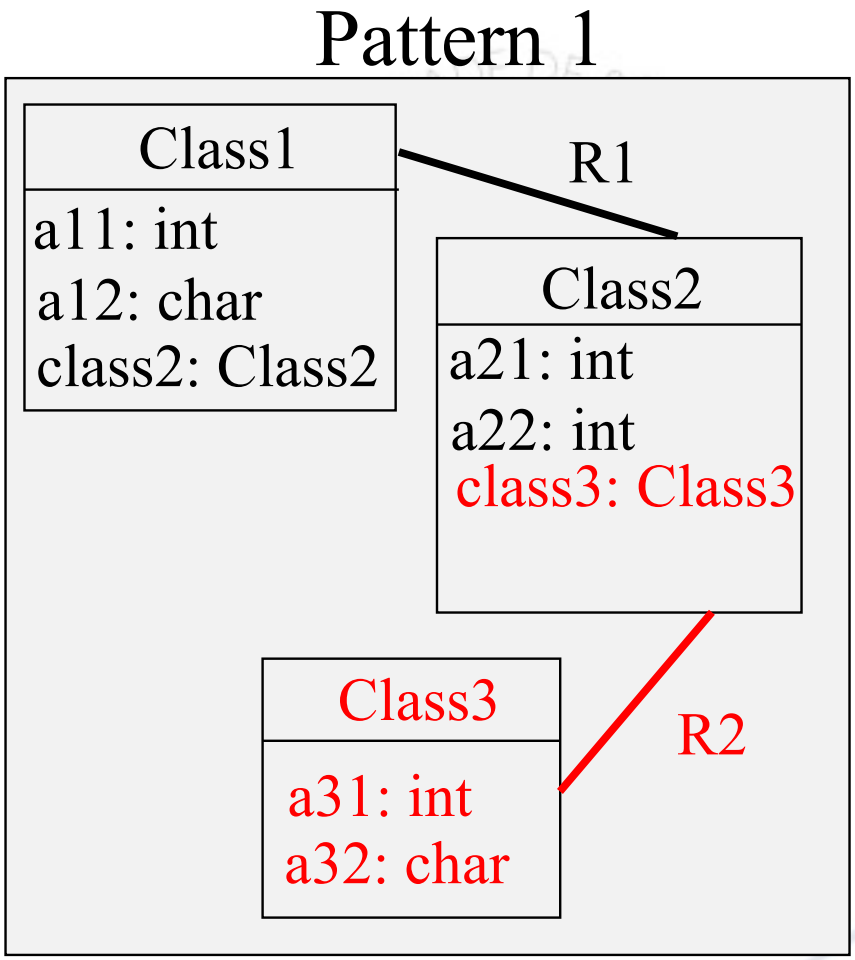
- Patterns solutions are expressed as a set of classes and their relationships
- A pattern can have variants
- Patterns are used together (combined or integrated with each other)
- Relationships among classes can be added or removed depending on the combination of patterns, or on pattern variants

Motivation – First Scenario - Variants



Default solution

Motivation – First Scenario - Variants

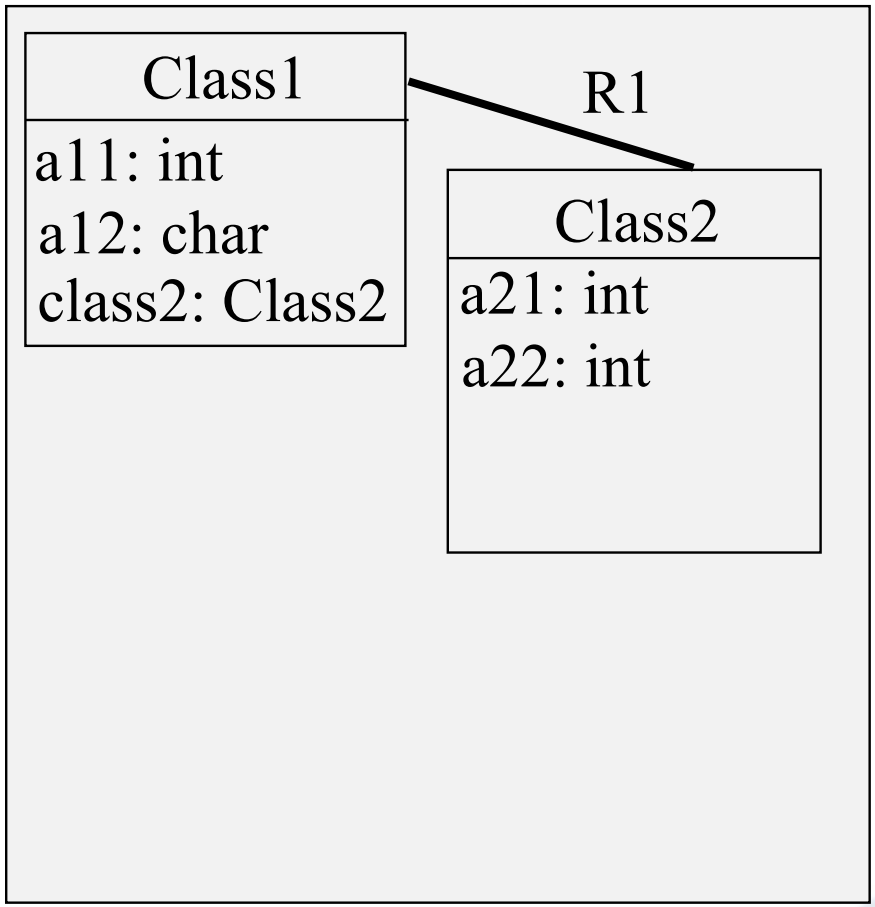


Default solution

What happens if there is a pattern variant in which Class3 does not exist?

Motivation – First Scenario - Variants

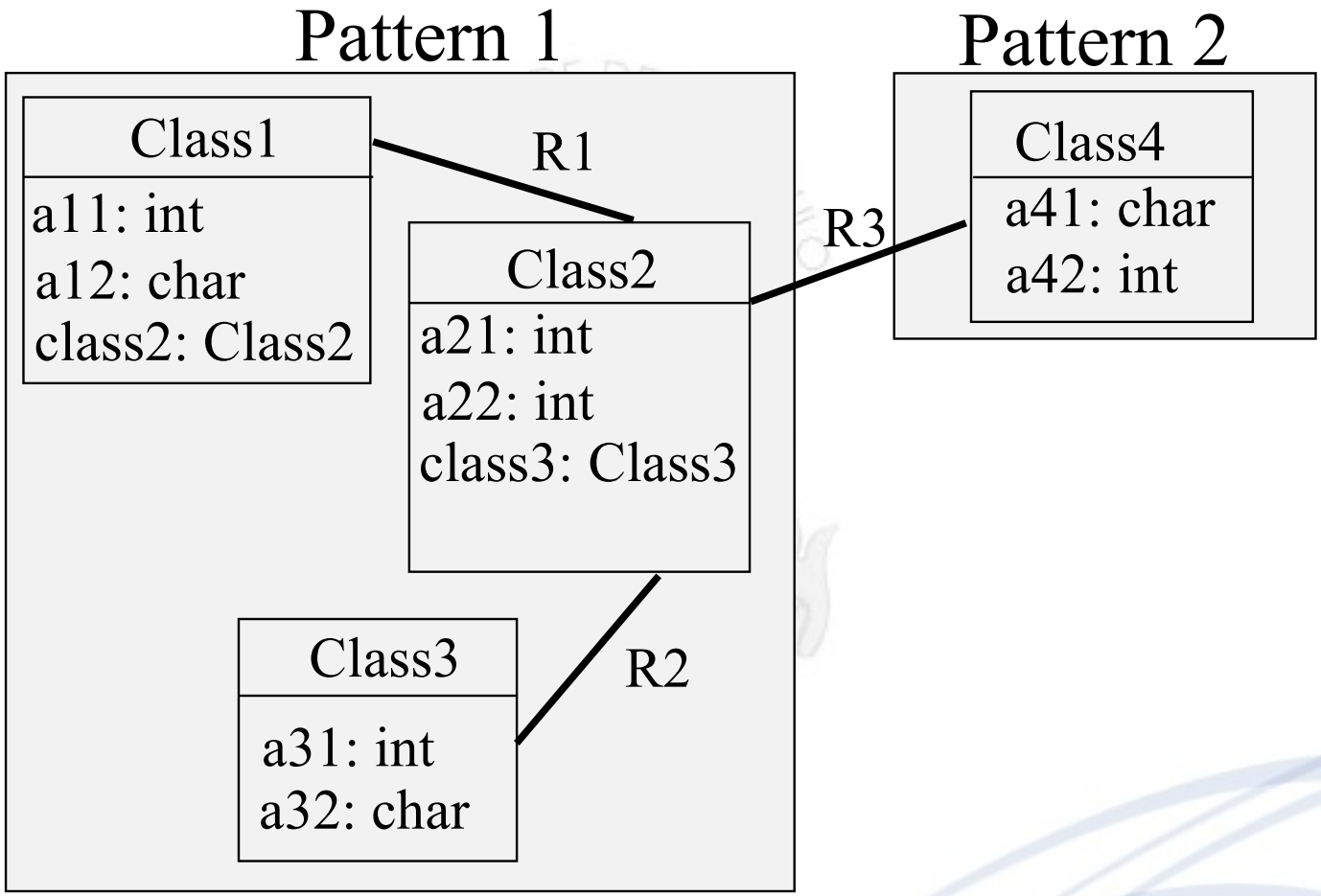
Pattern 1



Variant 1 – Class3 is removed as well as its relationship to Class2

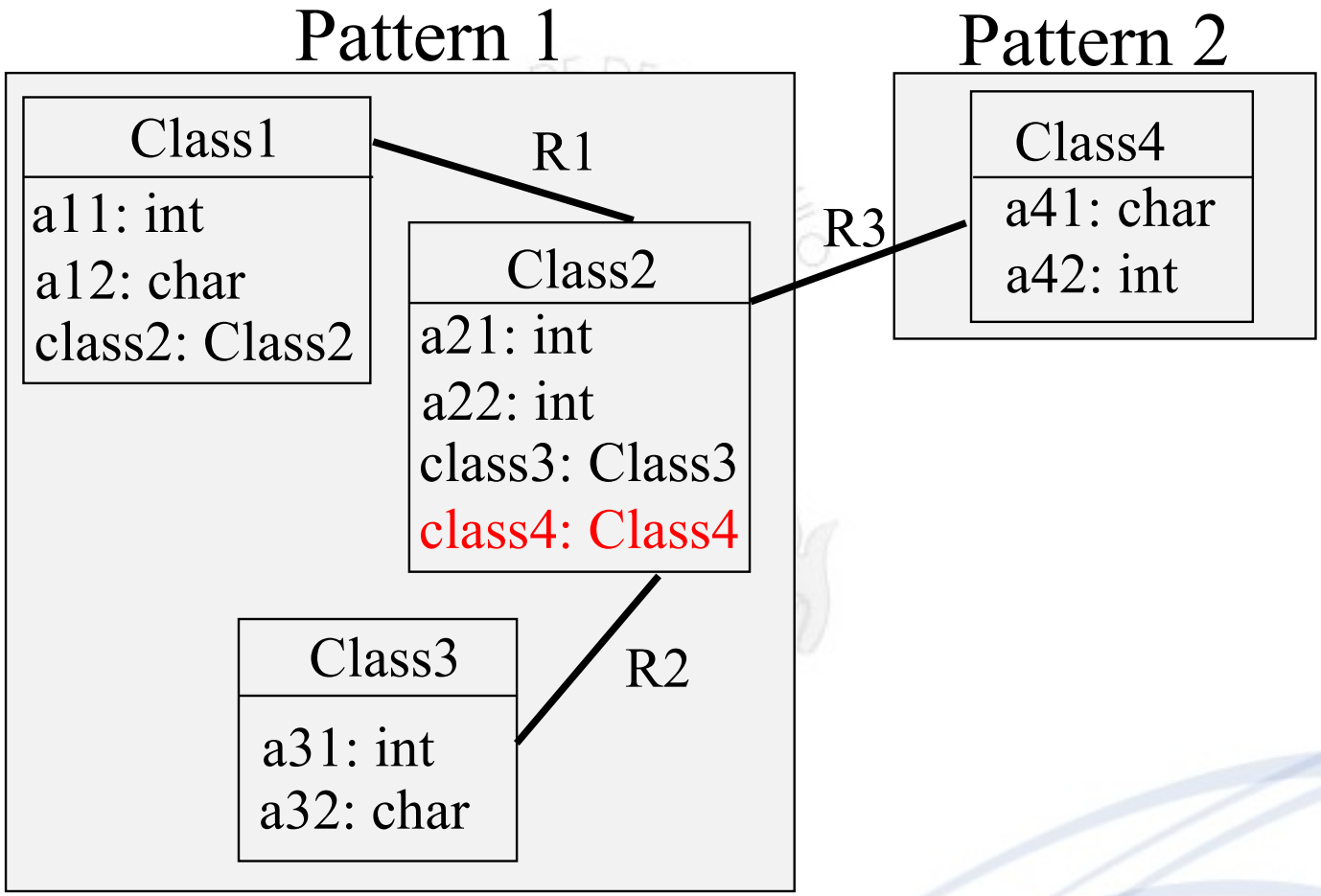
This is difficult to do because the relationship is hard-coded in Class2

Motivation – Second Scenario – Combination of patterns



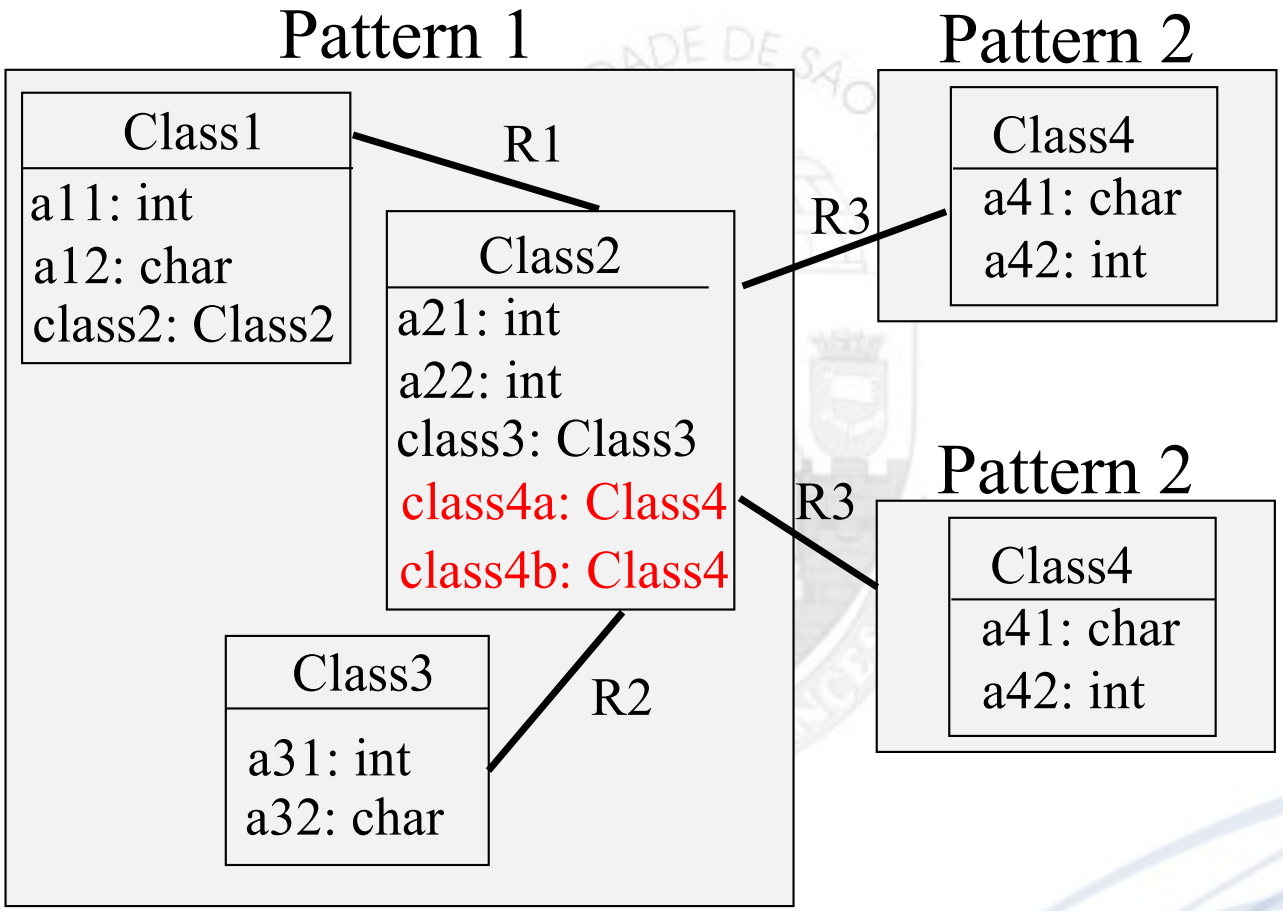
The user can apply only pattern 1
Or pattern 1 combined with pattern 2

Motivation – Second Scenario – Combination of patterns



The relationship (R3) implies an extra attribute in Class2

Motivation – Third Scenario – Same pattern applied several times



The relationship (R3) is instantiated twice

Objective of this work

- An approach to gradual implementation of patterns using aspect-oriented programming (AOP)
  - each pattern can be considered individually, and aspects are used to include the crosscutting behaviors, particularly those associated with relationships among classes

Proposed Approach

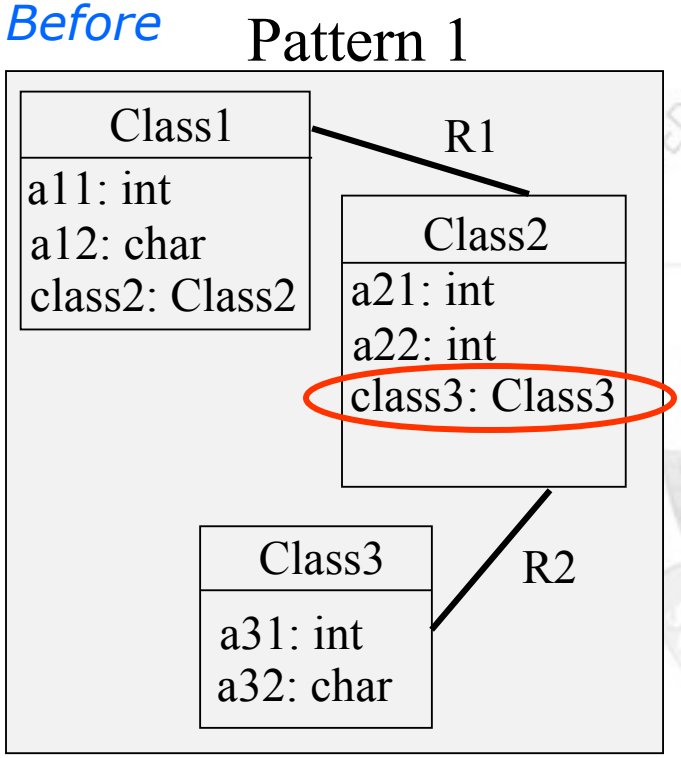
- In each step one pattern is designed and implemented (the analysis pattern is the basis for the design)
- Initially, all relationships are considered permanent, so they are hard-coded in the classes
- Relationship aspects [Pearce and Noble] are used to implement relationships when some situations occur during the design of a pattern (detailed in the next slides)
- To obtain a particular application, combine the classes with the aspects needed to implement the desired patterns and their relationships.

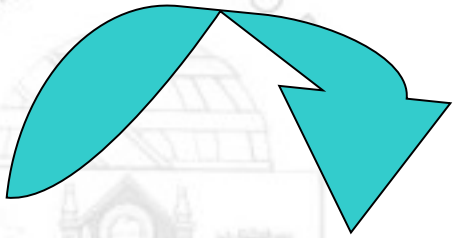
Proposed Approach

- Situations that lead to the refactoring of previous patterns to include relationship aspects:
  - The application of a pattern or pattern variant implies removing an existing relationship between classes
  - The application of a pattern or pattern variant requires the replacement of one relationship by another
  - The same pattern can be applied several times, each of which originating different concrete classes
  - Etc.

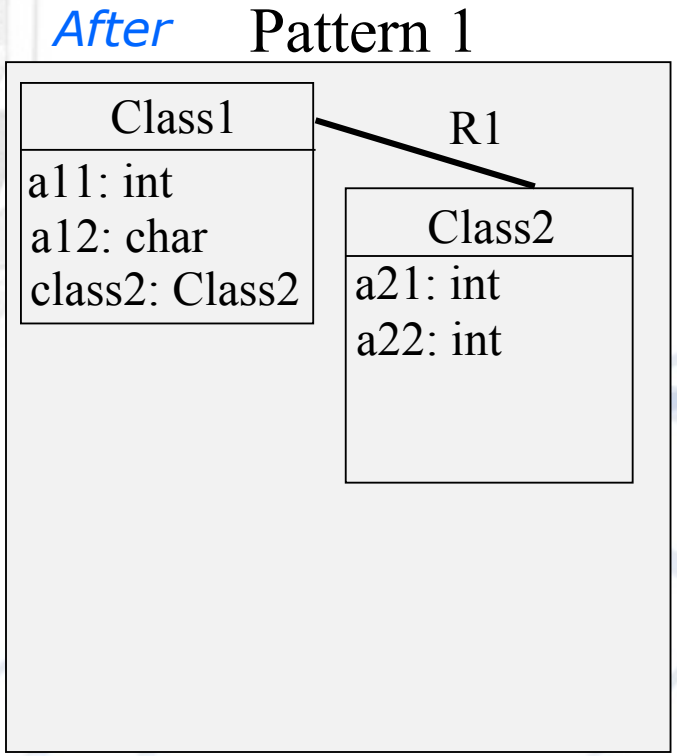
Example 1

Before

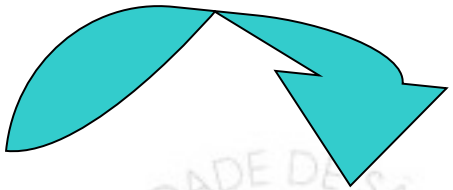




After



Example 1



Before

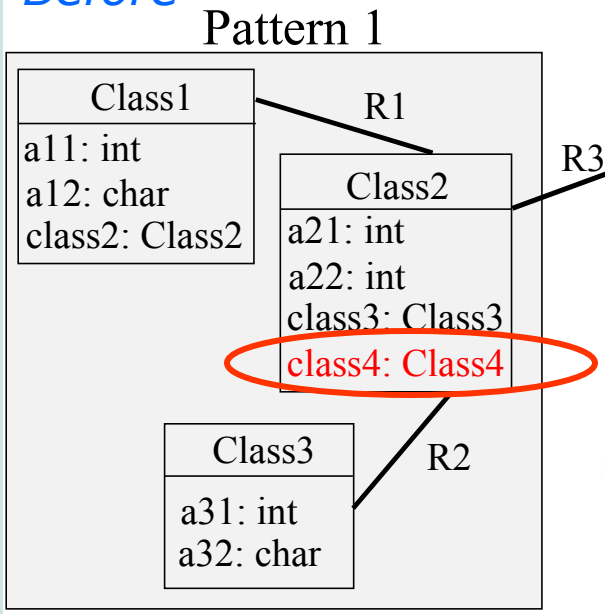
```
public Class2
{
  . . .
  int a21;
  int a22;
  . . .
  private Class3 class3;  // represents R2
  public Class3 getClass3()
    { return class3; }
  public void setClass3 (Class3 c3)
    { class3 = c3; }
}
```

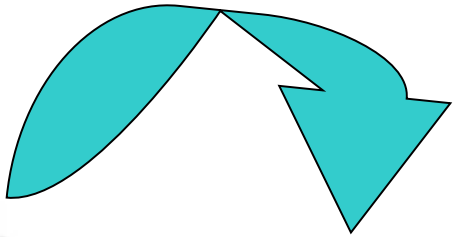
After

```
public Class2
{
  . . .
  int a21;
  int a22;
  . . .
}

public aspect R2
{
  // introduce attributte class3 in Class2
  private Class3 Class2.class3;
  // introduce setters and getters in Class2
  public Class3 Class2.getClass3()
    { return class3; }
  public void Class2.setClass3 (Class3 c3)
    { class3 = c3; }
}
```

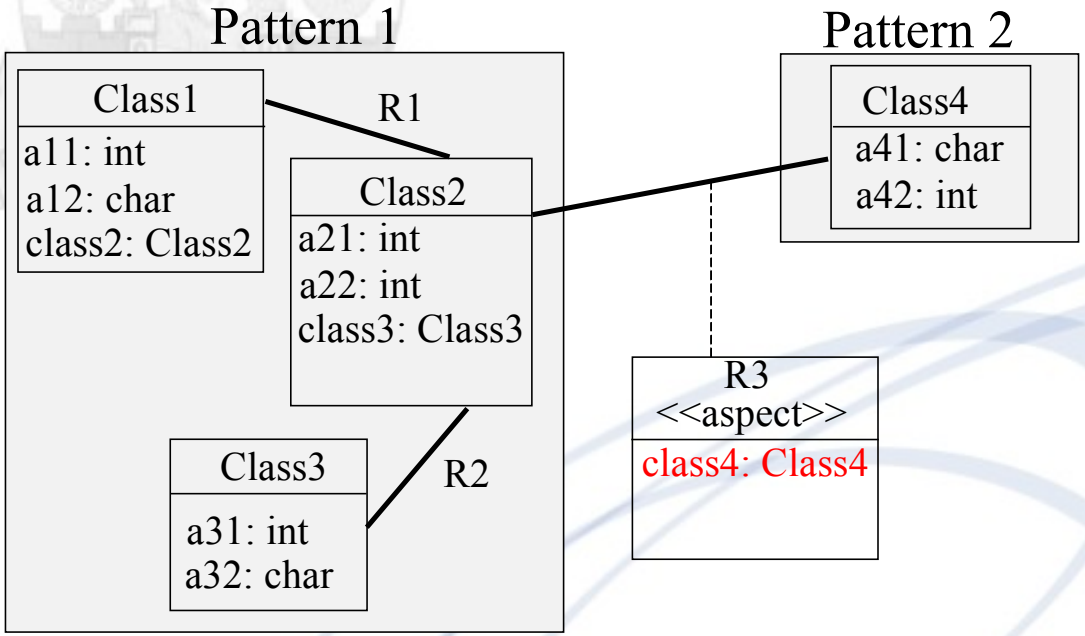
Example 2

Before





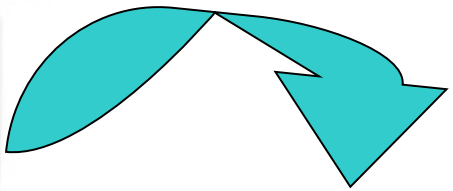
After



Example 2

Before

```
public Class2
{
   . . .
   int a21;
   int a22;
   . . .
   private Class4 class4;  // represents R3
   public Class4 getClass4()
     { return class4; }
   public void setClass4 (Class4 c4)
     { class4 = c4; }
}
```



After

```
public aspect R3
{
   // introduce attributte class4 in Class2
   private Class4 Class2.class4;
   // introduce setters and getters in Class2
   public Class4 Class2.getClass4()
     { return class4; }
   public void Class2.setClass4 (Class4 c4)
     { class4 = c4; }
}
```

Problem with this solution

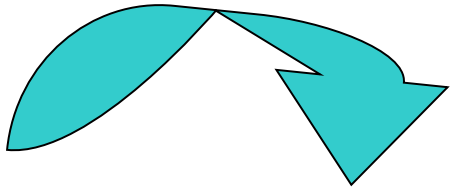
- Patterns have to be instantiated every time they are used
  - Several classes are abstract and need subclassing
- We don't know beforehand the name of the concrete subclass
  - This name would be used to create the relationship aspect

Valid solution

- Use relationship aspects and Relationship Aspects Library (RAL) [Pearce and Noble, 2005/2006]
  - Main idea: the relationship itself is separated from the participating objects.
  - Code belonging to the relationship is isolated and easily changed when necessary
  - Implementation (based on RAL)
    - abstract aspects are created to represent the relationships
    - By using Java Generics, it is not required to make the types of the target classes explicit in the base (or abstract) aspect
    - placeholders will be replaced by the

Before

```
public aspect R3
{
  // introduce attributte class4 in Class2
  private Class4 Class2.class4;
  // introduce setters and getters in Class2
  public Class4 Class2.getClass4()
    { return class4; }
  public void Class2.setClass4 (Class4 c4)
    { class4 = c4; }
}
```

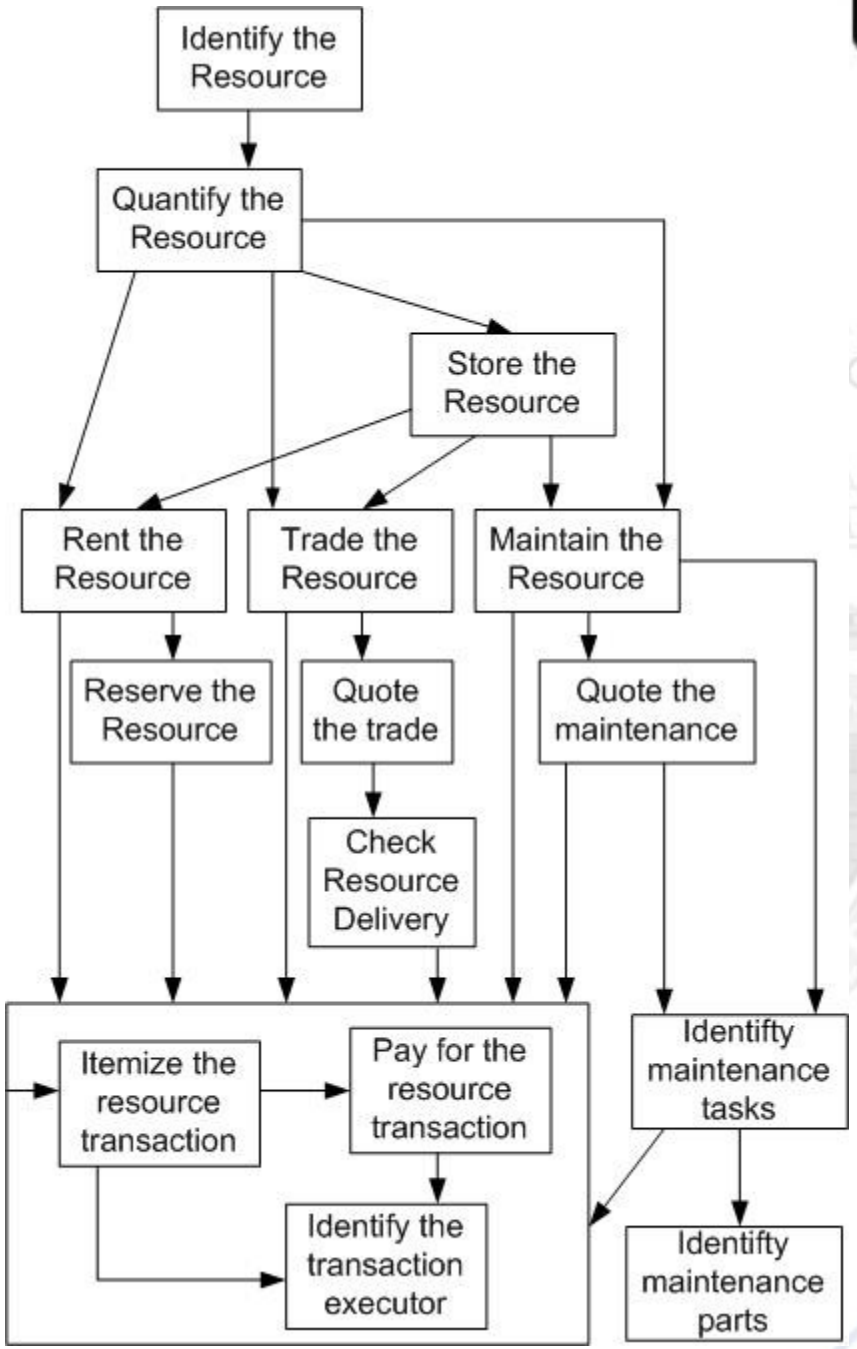


After

```
public abstract aspect R3<X extends Class2,
  Y extends Class4>
{
  interface InterfaceR3<T> {}
  declare parents : X implements
    InteFaceR3<Y>;

  private T InterfaceR3.class4;

  public T getClass4() { return class4; }

  public void setClass4(T c4) { class4 = c4; }

}
```

Case study

- GRN [Braga et al 1999] is an analysis pattern language to model systems for business resource management
  - Rental of resources (e.g. books, cars)
  - Trade/Sale of resources (e.g. products)
  - Maintenance of resources (e.g. car repair)
- Provides a solution in terms of class diagrams
- Patterns can be combined in many ways to produce concrete applications



GRN Pattern Language

(Braga et al 1999)

Example of an analysis pattern (Braga et al 1999)

Pattern 2: QUANTIFY THE RESOURCE

Context

You have identified a resource that your application deals with and its relevant qualities. An important issue to be considered now is the form of resource quantification. There are certain applications in which it is important to trace specific instances of a resource, because they are transacted individually. For example, a book in a library can have several copies, each lent to a different reader. Some applications deal with a certain quantity of the resource or with resource lots. In these applications, it is not necessary to know what particular instance of the resource was actually transacted. For example, a certain weight of steel is sold. In other applications, the resource is dealt with as a whole, as for example a car that goes to maintenance or a doctor that examines a patient.

Problem

How does the application quantify the business resource?

Forces

- Knowing exactly what is the form of quantification adopted by the application is important during analysis. A wrong decision at this point may compromise future evolution.

. . .

Example of an analysis pattern (Braga et al 1999)

Structure

There are four slightly different solutions for this problem, depending on the form of quantification. Figures 4 through 7 show the four QUANTIFY THE RESOURCE sub-patterns.

When it is important to distinguish among resource instances, use INSTANTIABLE RESOURCE sub-pattern (Figure 4). When the resource is managed in a certain quantity, use the MEASURABLE RESOURCE sub-pattern (Figure 5). When the resource is unique, use the SINGLE RESOURCE sub-pattern (Figure 6). When the resource is dealt with in lots, use the LOTABLE RESOURCE sub-pattern (Figure 7).

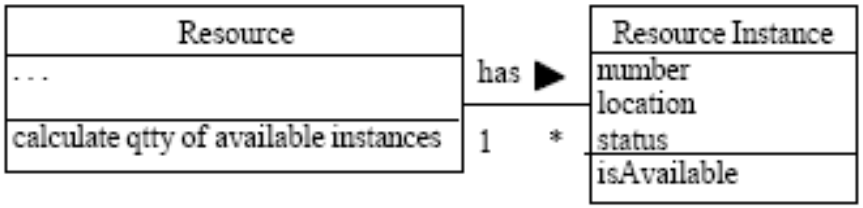


Figure 4: INSTANTIABLE RESOURCE sub-pattern

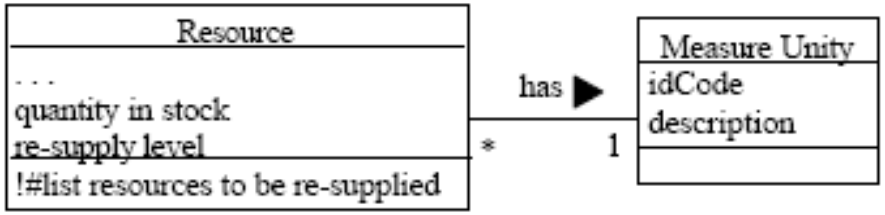
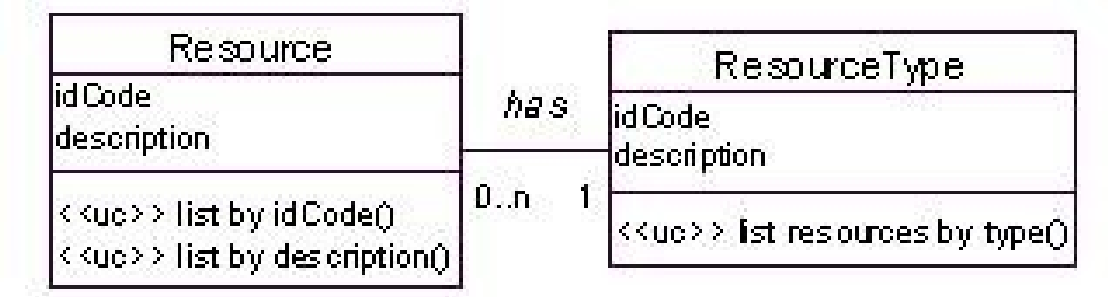
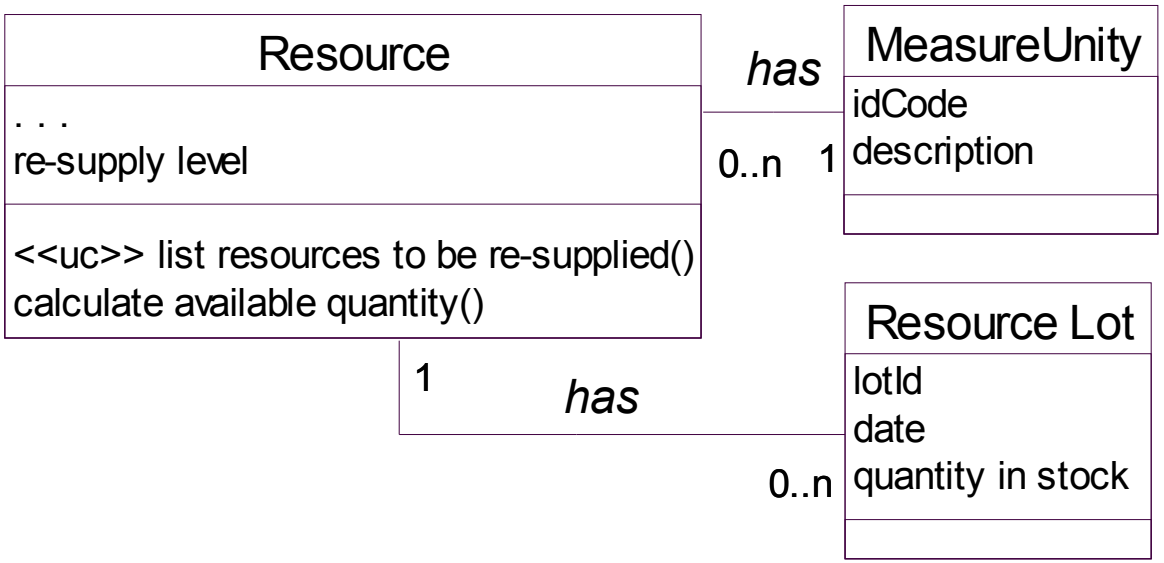


Figure 5: MEASURABLE RESOURCE sub-pattern

Example of analysis patterns (Braga et al 1999)



Pattern 1



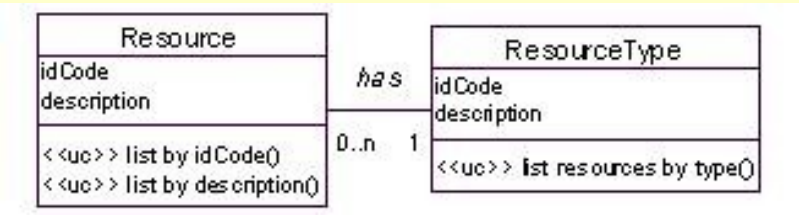
Pattern 2

Notice that class Resource is present in both patterns

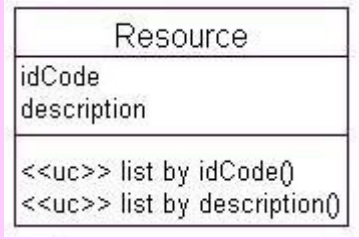
Pattern Implementation using our incremental approach

- First pattern (Identify the Resource): default solution and 2 variants
- The design has considered the variants, so the relationship between classes Resource and ResourceType was implemented using aspects.

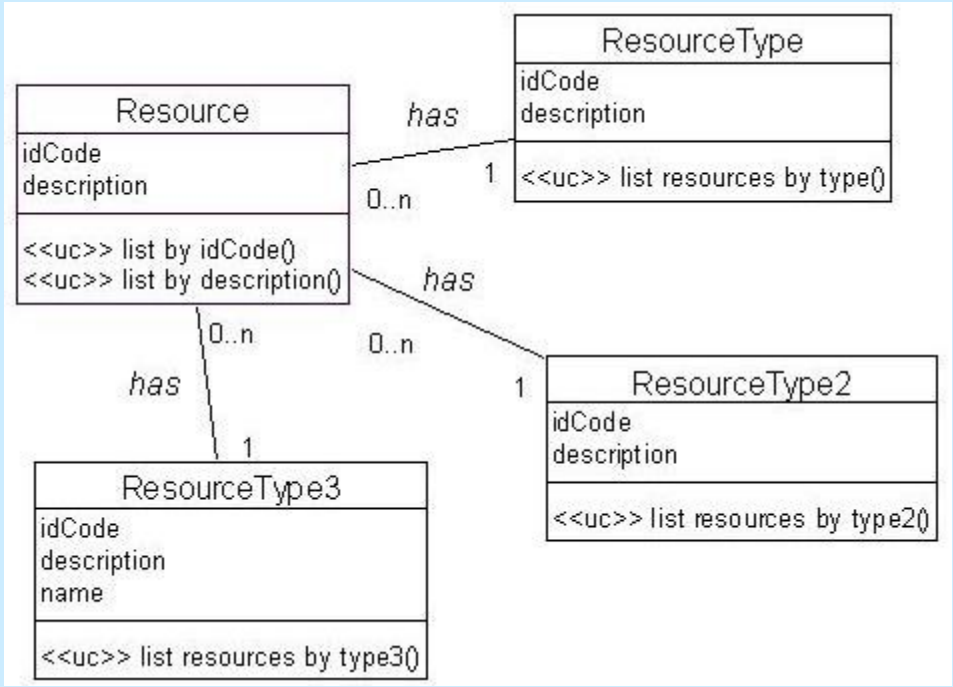
Pattern 1 and its variants (Braga et al 1999)



Pattern 1



Pattern 1 – V1



Pattern 1 – V2

Pattern Implementation using our incremental approach

```
public abstract class StaticObject {
private int idCode;
private String description;
public StaticObject(int idCode, String
  description) {
    this.idCode = idCode;
    this.description = description;
}
// getters and setters omitted
// validation code (such as not allowing
// idCode less than 1 or null description)
// were also omitted
}
```

```
public class Resource extends StaticObject
{ ... }

public class ResourceType extends StaticObject
{ ... }
```

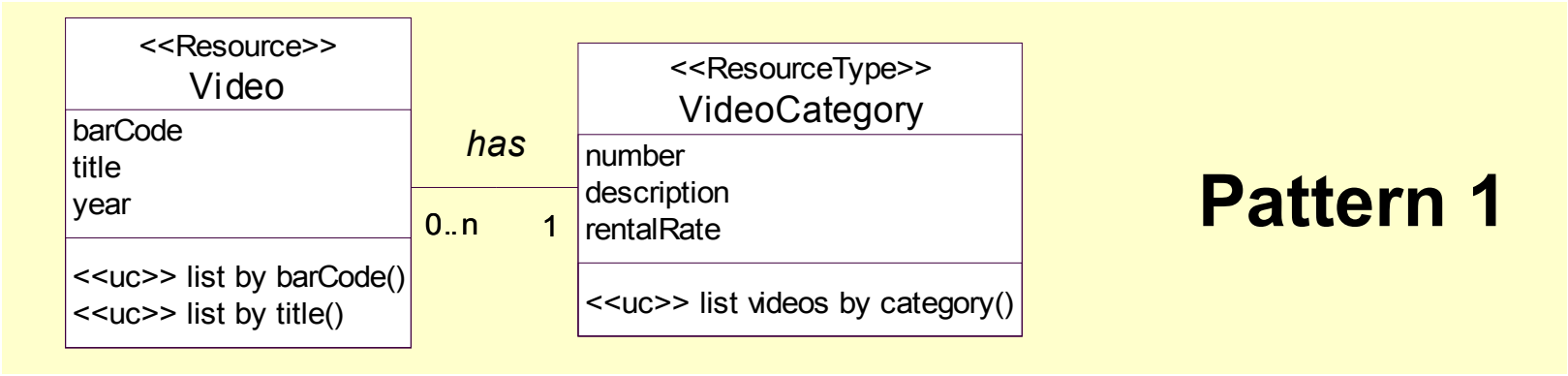
Classes Resource and ResourceType

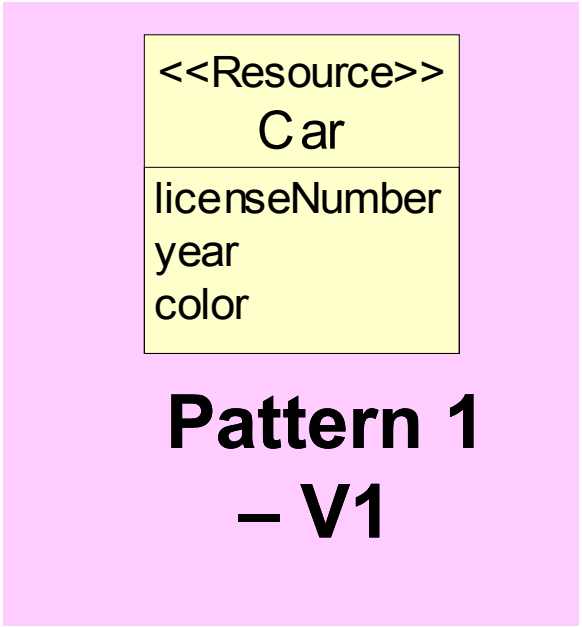
Pattern Implementation using our incremental approach

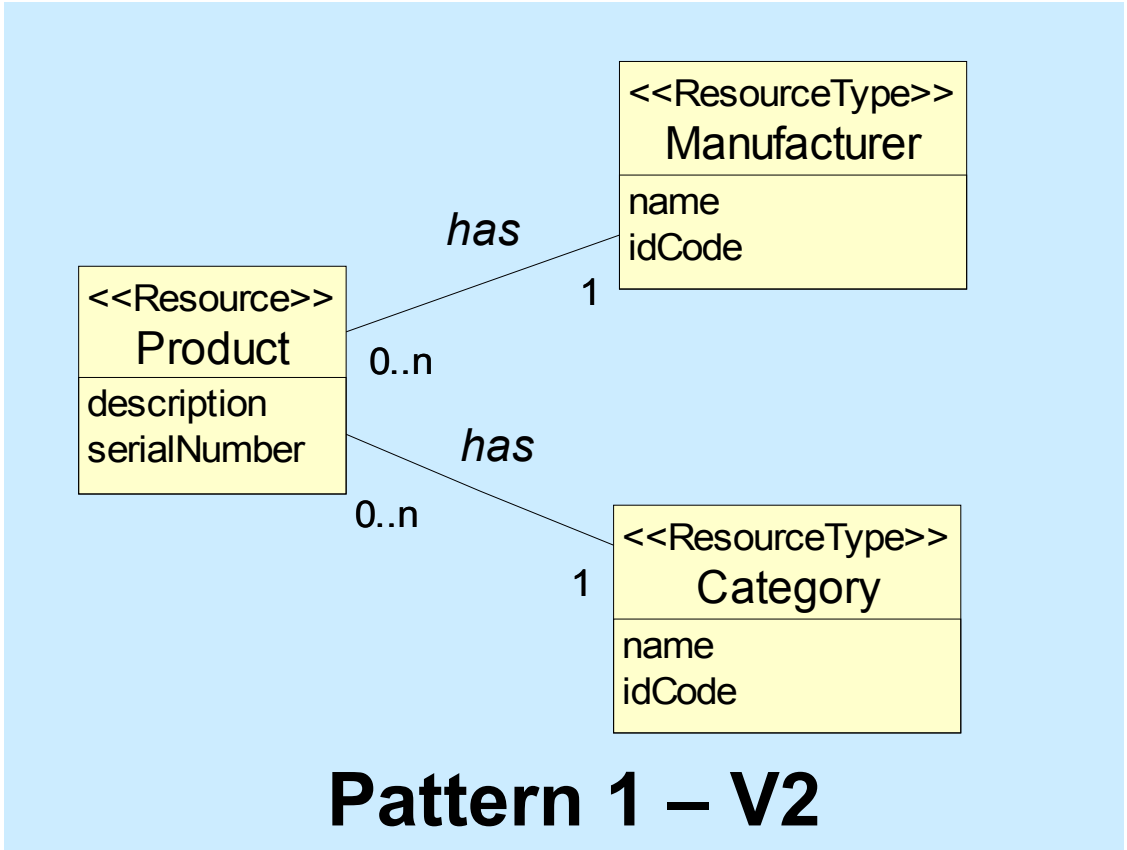
```
public abstract aspect Has<RES extends Resource,
 TYPE extends ResourceType>
{
   interface ISimpleResourceType<T>  {}
   declare parents : RES implements
        ISimpleResourceType<TYPE>;

   private T SimpleResourceType.resourceType;

   public T getResourceType() { return
      resourceType; }
   public void setResourceType (T type)
     { resourceType = type; }
}
```

Abstract aspect implementing the relationship

Example of pattern 1 instantiation







Pattern Instantiation Code (P1-v2)

```
public class Product extends Resource
{ ... }
public class Category extends ResourceType
{ ... }
public class Manufacturer extends ResourceType
{ ... }
```

Concrete Classes

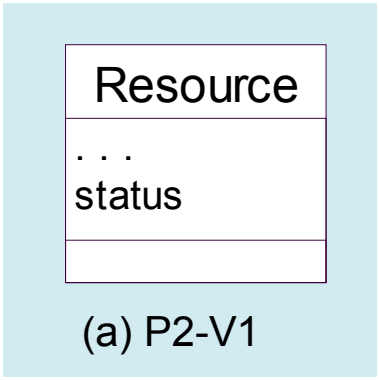
```
public aspect Has<Product,Category>{ }
public aspect Has<Product,Manufacturer>{ }
/*no extra code required*/
```

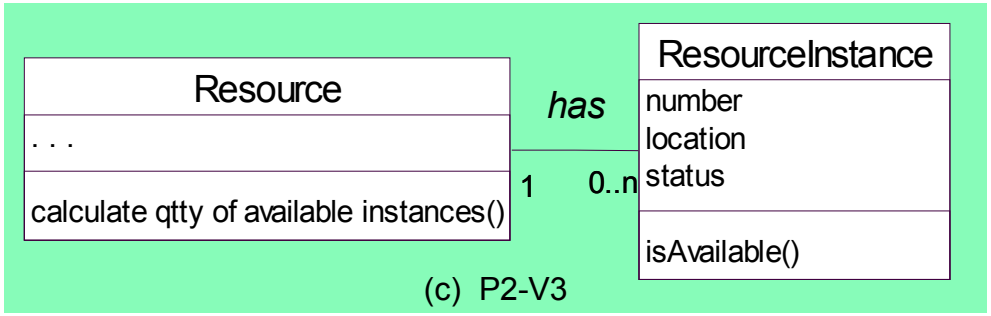
Concrete Aspects

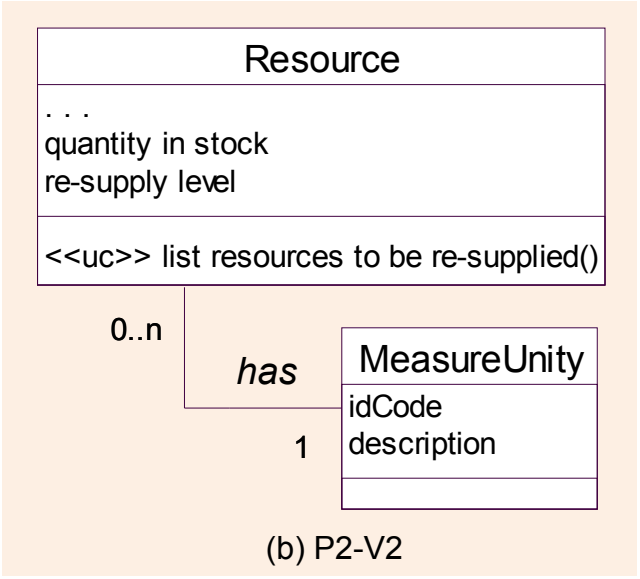
Pattern Implementation using our incremental approach

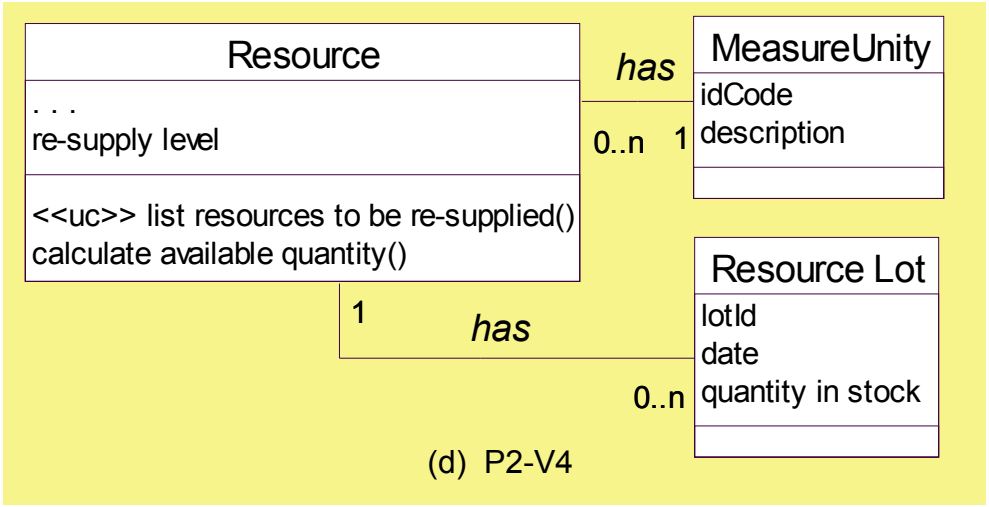
- Second pattern (Quantify the Resource): default solution and 3 variants.
- The design has considered the previous pattern, because a class of the new pattern ("Resource") is shared with pattern 1.
- Variants were also considered and designed in this step.

Pattern Implementation using our incremental approach

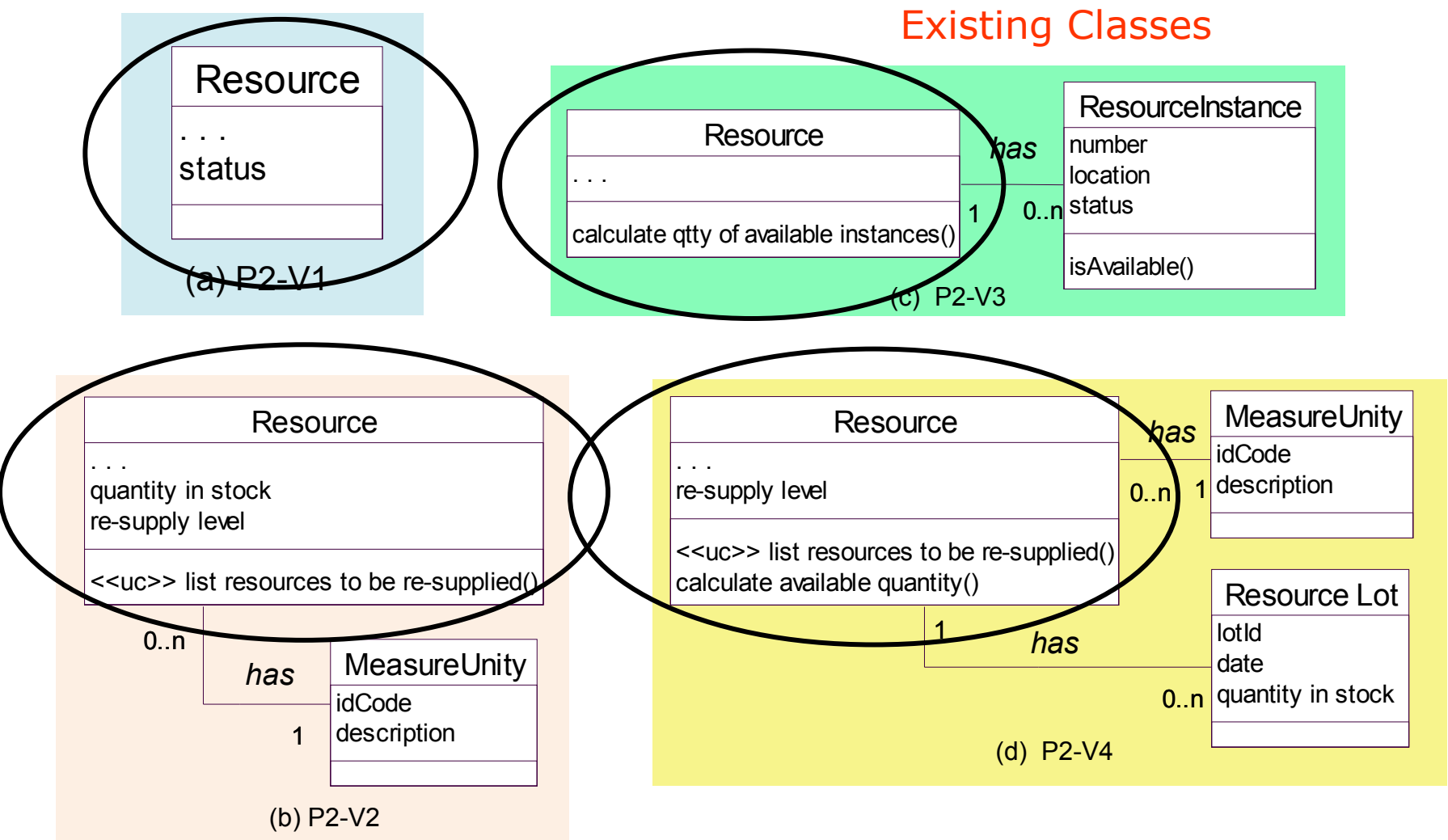






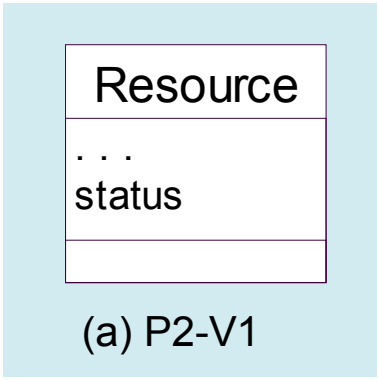


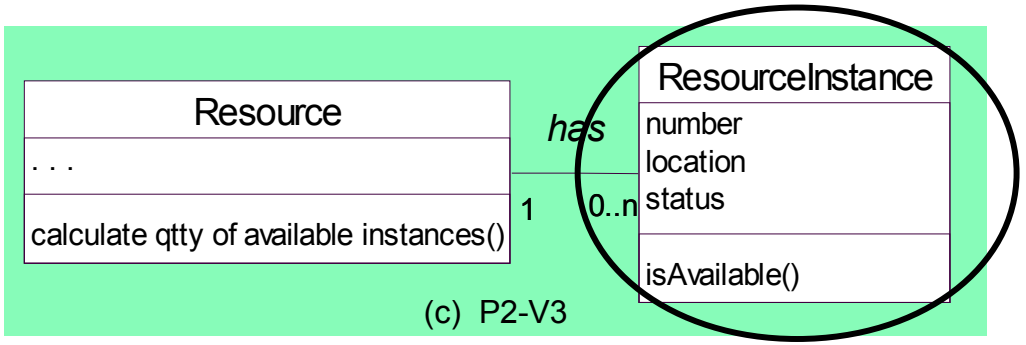
Pattern Implementation using our incremental approach

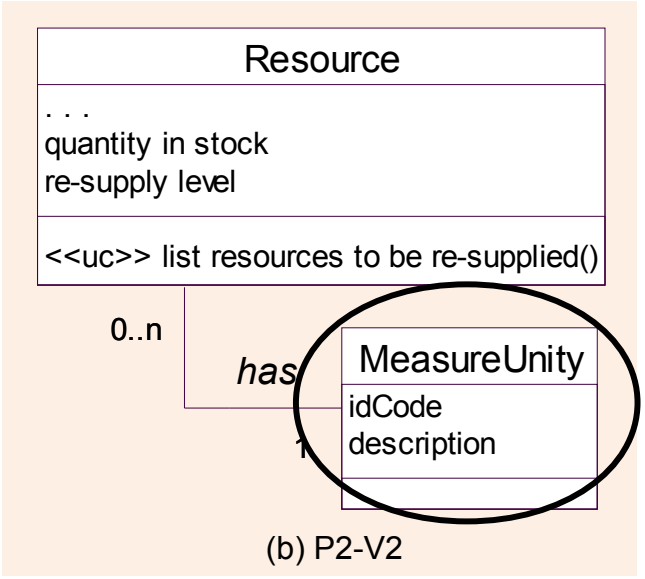


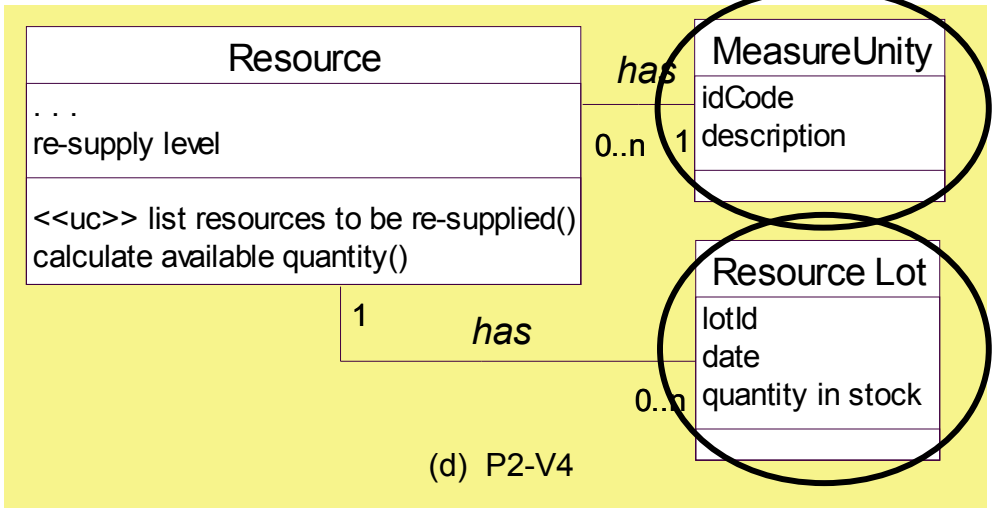
Pattern Implementation using our incremental approach

New Classes



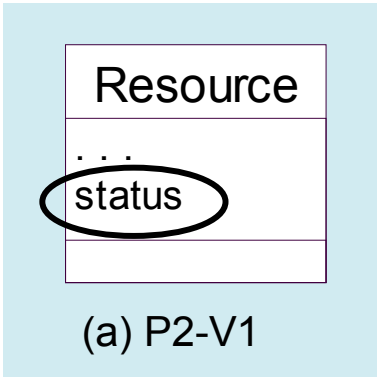


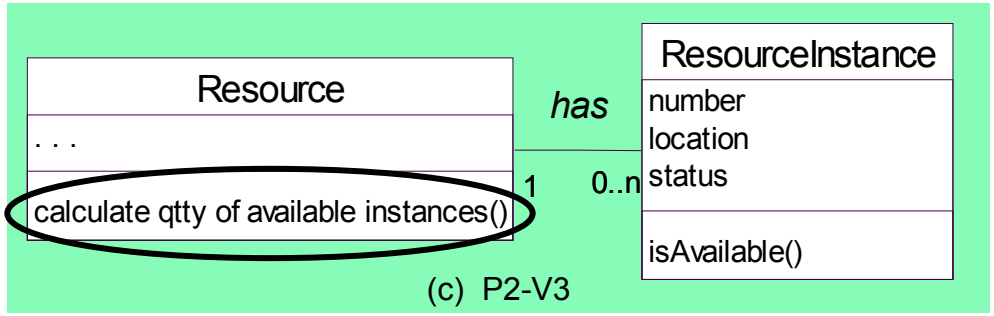


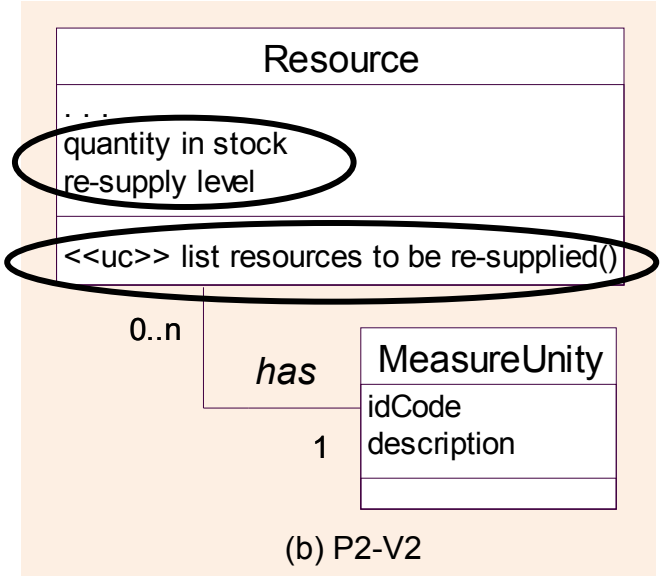


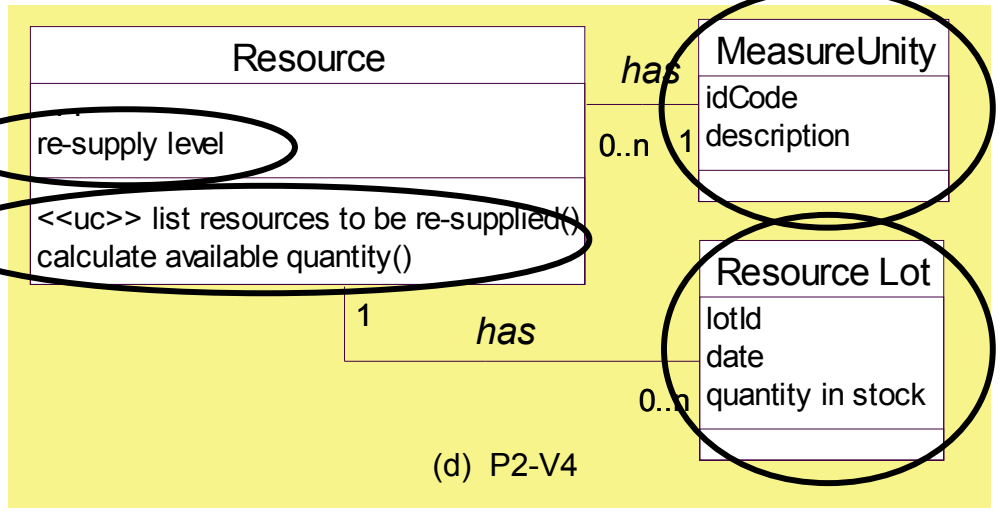
Pattern Implementation using our incremental approach

New Attributes/Methods









Pattern Implementation using our incremental approach

- New classes were added in the conventional way.
- Relationships to existing classes were implemented using aspects
- Aspects were used also to include new attributes and methods present in the variants.

Mapping Table

Useful to guide the framework user in the instantiation process

Pattern	Variant	Classes to include	What is included by the aspects?
1 – Identify the Resource	Default	Resource ResourceType	Has relationship (Resource to ResourceType)
	No Type	Resource	-
	Multiple Types	Resource ResourceType (one for each type)	Has relationship (one for each type)
2 – Quantify the Resource	Default – single resource	-	Attribute: status
	Measurable	MeasureUnit	Has relationship (Resource to MeasureUnit) *Attributes*: quantity in stock, re-supply level *Method*: list resources to be re-supplied
	Instantiable	ResourceInstance	Has relationship (Resource to ResourceInstance) *Method*: calculate qtty of available instances
	In lots	MeasureUnit ResourceLot	Has relationship (Resource to MeasureUnit) Has relationship (Resource to ResourceLot) *Methods*: list resources to be re-supplied and calculate avalailable quantity

Concluding remarks

- We have proposed an approach for incremental development of frameworks to support pattern languages.
- The framework instantiation process is more flexible than in traditional approaches where relationships are hard-coded in the classes.
- The framework construction (and future evolution) is also easier, because we do not need to anticipate the design of the next patterns.
- Application generators can automate the instantiation process.

CONTACT

- *Rosana*
  - rtvb@icmc.usp.br
  - www.icmc.usp.br/~rtvb
- *Rodrigo*
  - marchesini@gmail.com

- *ACKNOWLEDGMENTS*
  - *The authors thank FAPESP for the financial support of this research and both the University of Sao Paulo and ICMC Department of Computing Systems to support the presentation of this work at ECOOP/RAOOL.*