IMPLEMENTING PROPERTIES

Andreas Jahn – Dominik Winter

Agenda

1. The Property Pattern - Essentials
2. Property Modeling Approaches
3. Implementing the Property Pattern
4. Known Applications / Conclusions
5. Exercise

1. The Property Pattern

Definition and Essentials

Properties - Definition

What is a Property?

- Information about an object
- Value Types or instance of class
- Obtaining by query-methods
- Updating by modifier-methods

The Property Pattern – Current Synonyms(1)

Prototype Pattern

- Behaviour reuse (Inheritance)
- Process of reusing existing objects
- Existing objects serving as Prototypes
- Prototypal, Prototype-Oriented or Instance-Based Programming

The Property Pattern – Current Synonyms(2)

Adaptive Object Modeling

- Attributes held as a collection of Properties

Do-it-yourself Reflection

- Property List and Anything

  → Flexibility in type and number of attributes/parameters

- Registry

  → Mechanisms for managing global resources/objects

2. Approaches in property modeling

Fixed and Dynamic Properties

Fixed vs. Dynamic Properties – Overview

Fixed Properties ⟷

Dynamic Properties

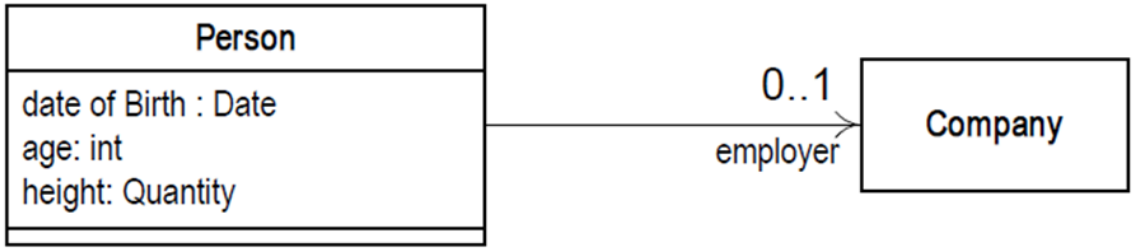
- The Flexible Dynamic Property
- The Defined Dynamic Property
- The Typed Dynamic Property
- The Separate Property
- The Typed Relationship
- The Dynamic Property Knowledge Level
- The Extrinsic Property

→ Focus on the different Dynamic Property Approaches

The Fixed Property

How do you represent a fact about an object?

- Giving a specific attribute for that fact
- Translating into a query or update method



The Fixed Property

- Implementing of Modifier Methods
- Implementing of Query Methods
- Choosing conventions and return value

```
class Person {

    public Date getDateOfBirth() {
        return DateOfBirth;
    }
    public int getAgte(){
        return age;
    }
    public Quantity getHeigth(){
        return height;
    }
    public Company getEmployer(){
        return employer;
    }
    public void setDateOfBirth(Date newDateOfBirth){
    }
    public void setEmployer(Company newEmployer){

    }
```

The Fixed Property - Summary

- ✓ Clear and explicit interface
- ✓ Simple and convenient
- ✓ Most common form and first choice
- ✗ Changing them frequently/ during runtime

The Dynamic Properties

How do you represent a fact about an object?

- Provide a paramterizable attribute which can represent different properties on the parameter

✓ Adding properties at run time

✓ Different varations of dynamic properties

✗ Unclear implementation

The Flexible Dynamic Property – Definition

How do you represent a fact about an object?

- Provide an attribute parameterzied with a string
- Declare a property by just using the string

The Flexbile Dynamic Property - Implementing

- Adding a vacation address property to a person
- Implementing getValueOf with parameter(String key)
- Implementing setValueOf with Parameter (String key, Object value)

```
class Person{

    public Object getValueOf(String key);
    public void setValueOf(String key, Object Value);

    kent.setValueOf("VacationAddress", anAddress);
    Address kentVacation = (Address) kent.getValueOf("VacationAddress")
```

The Flexbile Dynamic Property – Replacing

- Difficult substitution for operations
- Replacing a dynamic property with an operation
- Adding a trap in the general accessor

```
class Person{
  public Object getValueOf(String key){
    if (key = "VacationAddress") return calculatedVacationAddress();
    if (key = "VacationPhone") return getVacationPhone();
    // else return stored value
```

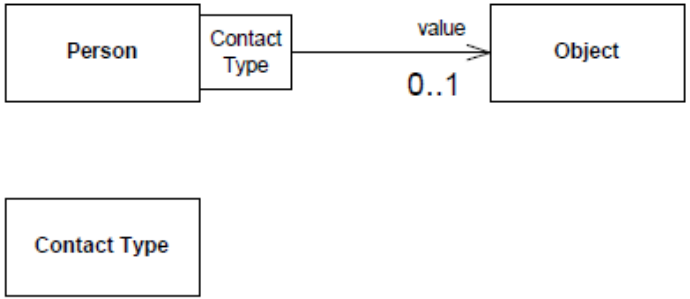
The Flexible Dynamic Property – Summary

- ✓ Simply adding of dynamic properties
- ✓ No recomiling of the person class
- ✗ Reduction of clarity of dependencies
- ✗ No possiblilty of design-time-checking
- ✗ Difficult substitution for operations

The Defined Dynamic Property - Definition

How do you represent a fact about an object?

- Provide an attribute parameterzied with an instance of some type
- Declare a property by creating a new instance of that type



The Defined Dynamic Property - Implementing

- Implementing similar to the Flexible Dynamic Property
- Key is limited by the instances of ContactType
- Adding during runtime possible

```
class Person{
  public Object getValueOf(ContactType key);
  public void setValueOf(ContactType key, Object value);

class ContactType{
  public static Enumeration instances();
  public boolean hasInstanceNamed(String name);
  public static ContactType get(String name);
```

The Defined Dynamic Property - Checking

- Difficult substitution for operations
- Checking a defined dynamic property with an operation
- Implementing a clean up code

```
class Person{
  public static ContactType get(String name){
    if (! hasInstanceNamed (name))
          throw new IllegalArgumentException
          ("No such contact type");
          // return the contact type

    // use with

    Address kentVacation =
      (Address) kent.getValueOf(ContactType.get("VacationAddress"));
```

The Defined Dynamic Property - Summary

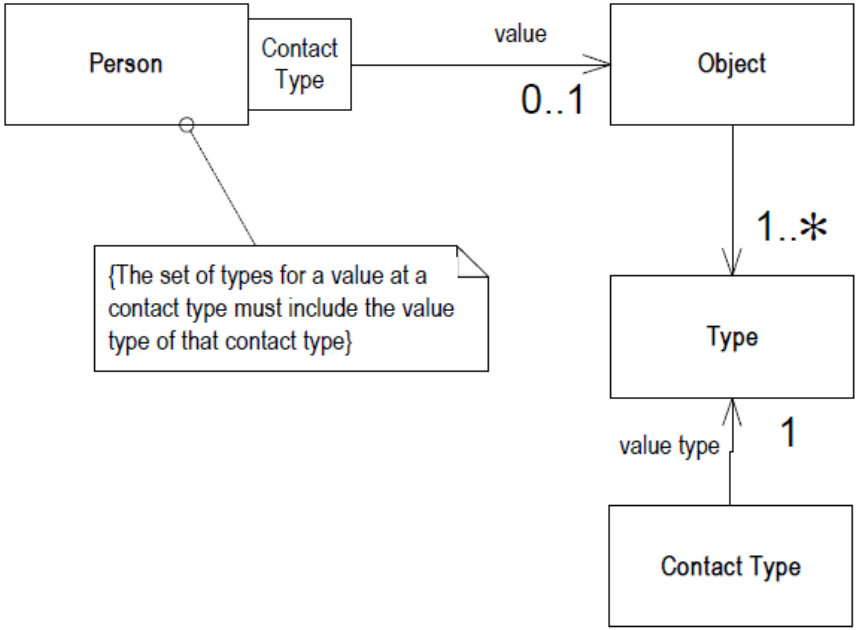
- ✓ Keys limited by the instance of contact type
- ✓ Possibility of adding properties at run time
- ✓ Look up of list with legal keys
- ✗ Difficult substitution for operations

The Typed Dynamic Property - Definition

How do you represent a fact about an object?

- Provide an attribute parameterzied with an instance of some type
- Declare a property by creating a new instance of that type
- Specify the value type of the property

The Typed Dynamic Property - Definition



The Typed Dynamic Property - Implementing

- Instances of contact type indicate the properties the person has
- Instances of contact type indicate the type of each property
- The type contrains the value

```
class Person{
    public Object getValueOf (ContactType key);
    public void setValueOf (ContactType key, object value);

class ContactType{
    public class getValueType();
    public contactType (String name, Class valueType);

class Person{
    public void setValueOf (ContactType key, object value);
    if (!key.getValueType().isInstance(value))
        throw IllegalArgumentException
        ("Incorrect type for property")
        // set the value
```

The Typed Dynamic Property - Summary

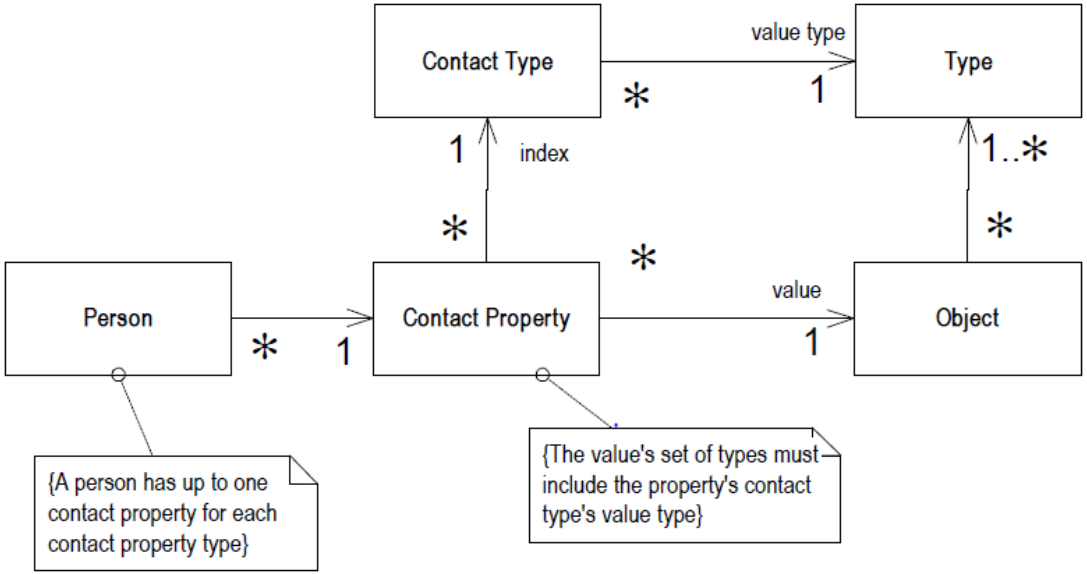
- Instances of contact type indicate what properties the person has and the type of each property
- Type constrains the value
- Runtime checking and avoiding errors
- Good usage for a strongly typed environment

The Separate Property - Definition

How do you represent a fact about an object and allow facts to be recorded about that fact?

- Create a separate object for each property
- Facts about that property can then be made properties of that object

The Separate Property - Definition



The Separate Property - Implementing

- Implementing the class ContactProperty
- Enlargement of Typed Dynamic Property

```
class Person{
    public Enumeration getProperties();

class ContactProperty{
    public Object getValue();
    public Class getType();
    public ContactType getIndex();
```

Dynamic Properties – Multi-valued associations

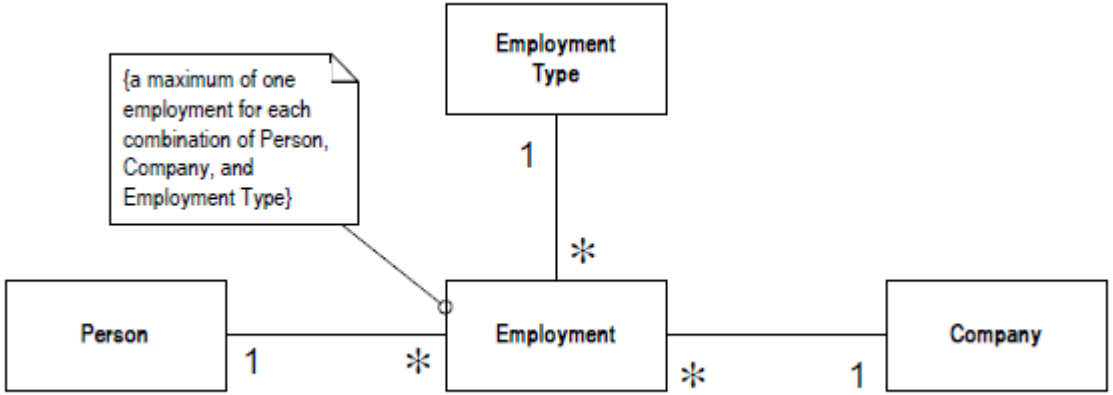
- Examples above with the focus on Single-valued associations for each key
- Possiblilty of implementing Multi-valued associations
- Possible procedure: Typed Relationship

Typed Relationship- Definition

How do you represent a relationship between two objects?

- Create a relationship object for each link between the two objects
- Giving the Relationship Object a type object to indicate the meaning of the relationship
- Type Object is the name of the Multi-Valued property

Typed Relationship – UML Diagram



Typed Relationship - Summary

- Perfect for bi-directional relationships
- Providing a simple point to add porperties into the relationship

```
class Employment {
    public Employment (Person person, Company company, Employment Type type);
    public void terminate();
}

class Person {
    public Enumeration getEmployments();
    public void addEmployment (Company company, EmploymentType type);
}
```

Further Dynamic Properties Enlargements (1)

1) Dynamic Property Knowledge Level

How do you enforce that certain kinds of objects have certain properties when you use dynamic properties?

- Creating a knowledge level to contain the rules of what types of objects use which types of properties

Further Dynamic Properties Enlargements (2)

2) Extrinsic Property

How do you give an object a property without changing its interface?

- Making another object responsible for knowing about the property

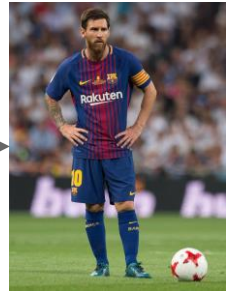
Summary Points on Dynamic Properties

- Lack of clarity of the interface
- Difficulty in using operations instead of stored data
- Replacing Compile-Time-Check for a Run-Time-Check
- Big Usage in databases in case of data migration and avoiding to change database schema

3. Implementing the Property Pattern

Basic Understanding – Property Based Modeling

Class
„FootballPlayer“



Instance of
„FootballPlayer“

- **Dribbling**: Great
- **Header**: Bad
- **Speed**: Fast

clone

Clone of
Lionel Messi

- **Dribbling**: Great
- **Header: Bad**
- **Speed: Fast**
- ***Defensive: Great***

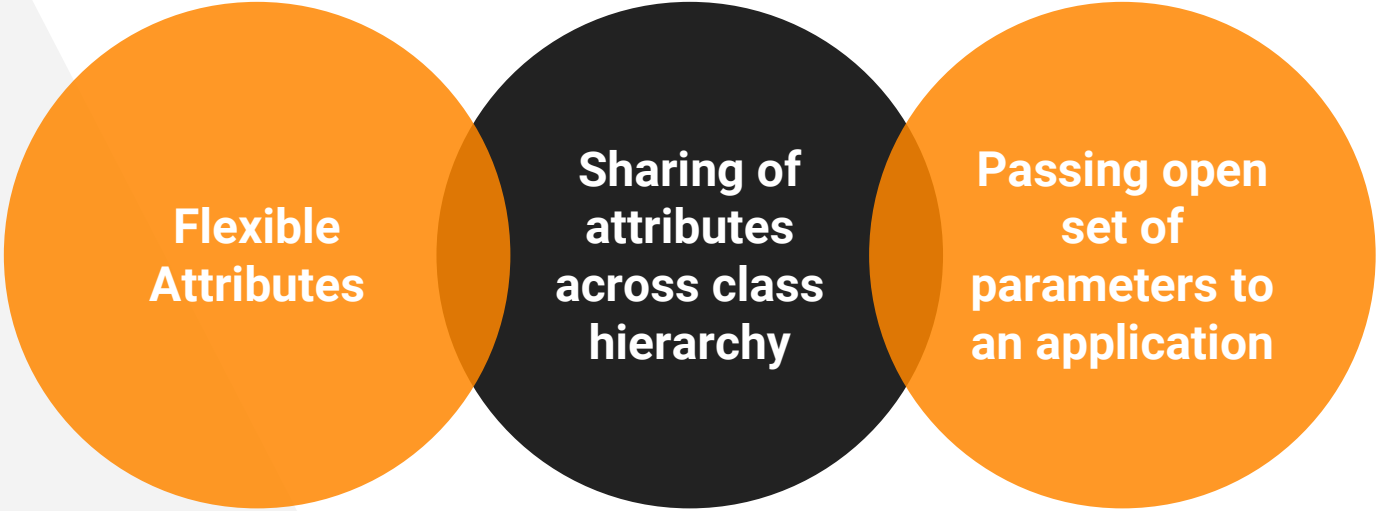
Main use case

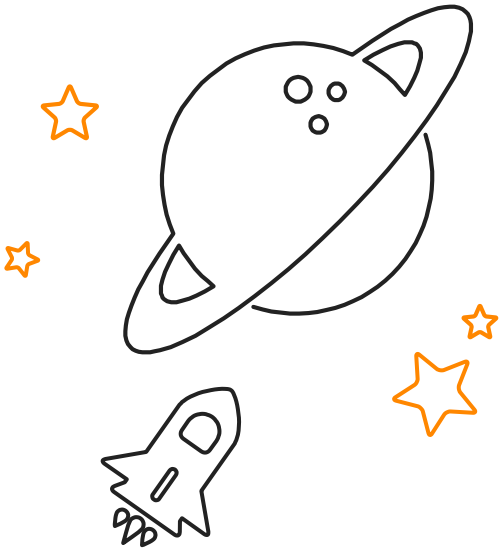
The Property pattern is used to attach a ***flexible set of attributes*** *to an object at* ***run-time****.*

Questions to be solved

- How do you define parameters in a flexible way?
- How do you define the attributes of your components in a way they can be extended by client components?
- How do you implement these common attributes, showing that they are really the same to a programmer of a client component?
- How do you implement attributes that should be attached or detached during runtime?

Problems to be addressed





SOLUTION:

Providing a data structure that allows to associate names (e.g. string values) with other values or objects.

Core API to access the Property Collection

- **`get(name)`**

Return the value for a given name

- **`put(name, value)`**

Add new key-value pair to the Property Collection

- **`has(name)`**

Check if give value exists in the Property Collection

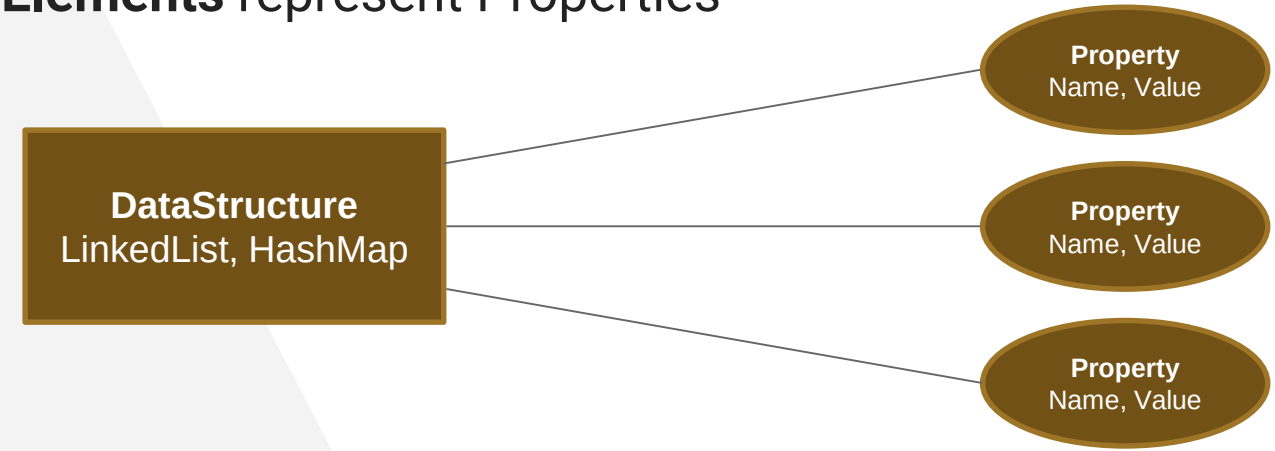
- **`remove(name)`**

Remove the given value from the Property Collection

Basic Requirements for Implementation

- **Objects**: Implementation of a key-value data structure (e.g. Map) to access

- **Elements** represent Properties



Data Structures

- Simple way: **LinkedList**
  - only for small property lists!! Performance issue

- **HashTable** – almost constant performance in find/insert/remove
  - Cost of more memory
  - Cost of the hash function per access

- **Hybrid Approach**
  - LinkedList as long as list is small enogh (40 to 50 items)
  - Swich to HashTable by growing lists

Reserved Property for parent link

- **Each Prototype can have a parent list** (compare to football player example)
  - First: Check the „local“ property list
  - Then: Look in the parent for the property

- **Implementation**: Reserved property pointing to parent list (e.g. „prototype“, „parent“ or „class“)

„Look in my list, and if the property isn‘t there, look in my parents list“

put - Adding Properties

```
public V put(final String name, final V value) {

    if (PARENT.equals(name) && !(value instanceof kp.propertypattern.Properties)) {

        throw new IllegalArgumentException(
                "parent must be an instance of Properties");
    }

    return properties.put(name, value);
}
```

get - Retrieve properties from Property List

```
public V get(final String name) {
    if (properties.containsKey(name)) {

        return properties.get(name);
    }

    if (properties.containsKey(PARENT)) {

        kp.propertypattern.Properties<String, V> parent =
                (kp.propertypattern.Properties<String, V>) properties.get(PARENT);

        return parent.get(name);
    }

    return null;
}
```

The Deletion Problem

Problem: Deletion of a property which is inherited from the parent.

- Property should not be deleted for other instances
- A missing key does **not** mean „*not found*“!

Solution:

-> Look in my parent for this key

remove – Delete Properties

```
public V remove(final String name) {

    if (properties.containsKey(PARENT)) {

        kp.propertypattern.Properties<String, V> parent =
                (kp.propertypattern.Properties<String, V>) properties.get(PARENT);

        if (parent.has(name)) {

            properties.put(name, null);

            return parent.get(name);
        }
    }

    if (properties.containsKey(name)) {

        return properties.remove(name);
    }

    return null;
}
```

Performance Optimization

Using the Property Pattern can lead to performance problems.

Approaches for optimizatiion:

- Interning Strings
- Copy-on-read-Cashing „*Plundering*“
- Refactoring to fields
- Perfect Hashing

Perfect Hashing

- **Problem**: Collisions
  - 2 or more distinct keys generate the same hash
  - The more collisions happen, the worse the performance
- **Approach**: Using a perfect HashFunction
  - LookUps speed up
  - Not necessary to use on alle Property List
  - Only useful if properties are known at compile time (or early runtime)
- Apache Commons HashCodeBuilder

4. Conclusion

Some Use Cases

- **JavaScript**
  Using the properties at its core
- **Wyvern** – Role Play Game
  Very flexible RPG
- **Eclipse Backend**– Java language modeling
  Used to model the Java language in Eclipse

Tradeoffs

- Huge flexibilty…

  …but tradeoff in **type safety** and **performance**.

  …but tradeoff in **queryability**.

  …but **harder to implement** in languages that don‘t provide properties from scratch.

Conclusion

- Rare documented pattern (few blogposts & papers)
- No common naming of the pattern
- Developers use the pattern implicitly without knowing
- Great approach in designing open-ended systems

Exercise

https://goo.gl/o6z2MZ