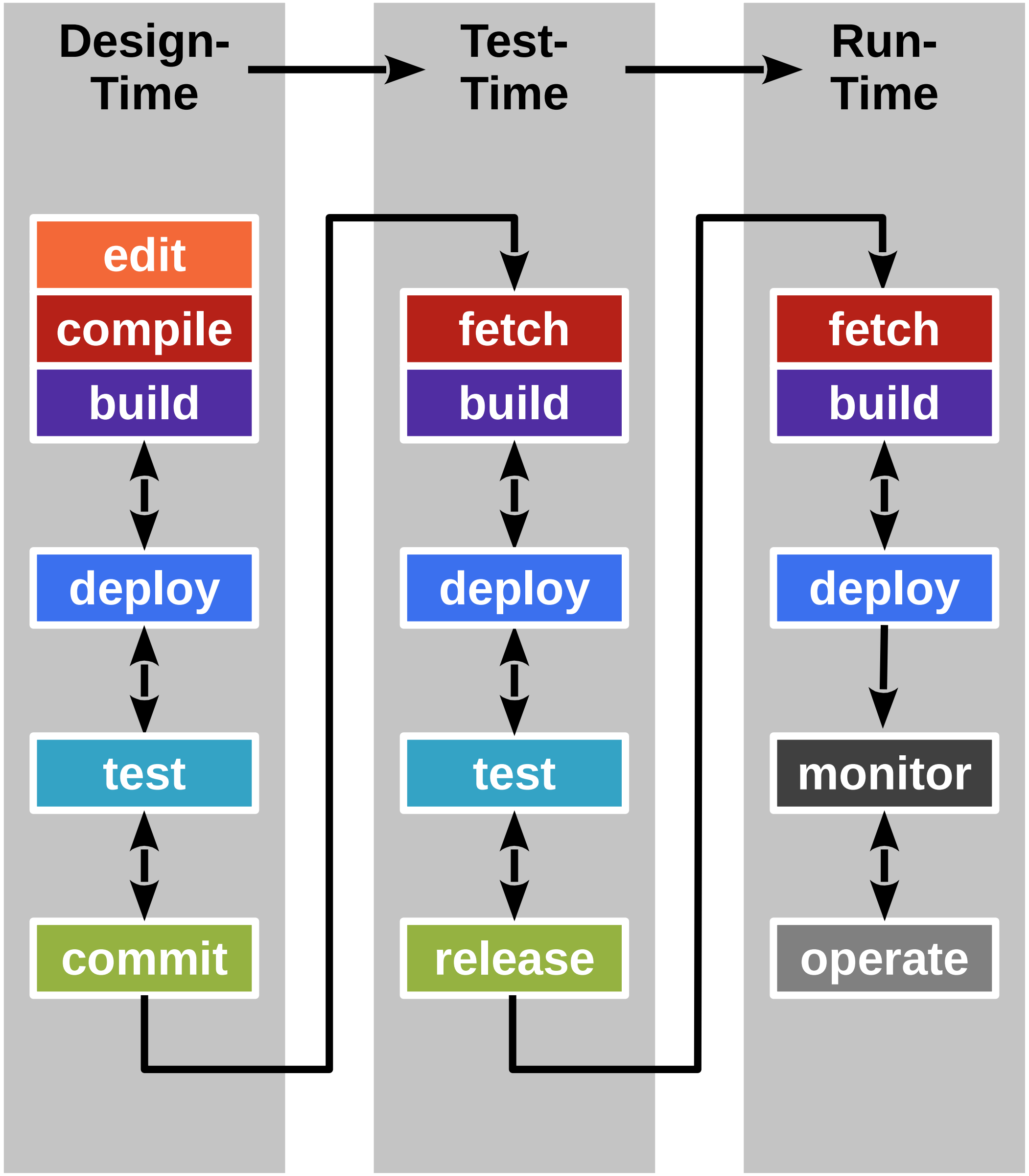
Type Objects

Prof. Dr. Dirk Riehle

Friedrich-Alexander University Erlangen-Nürnberg

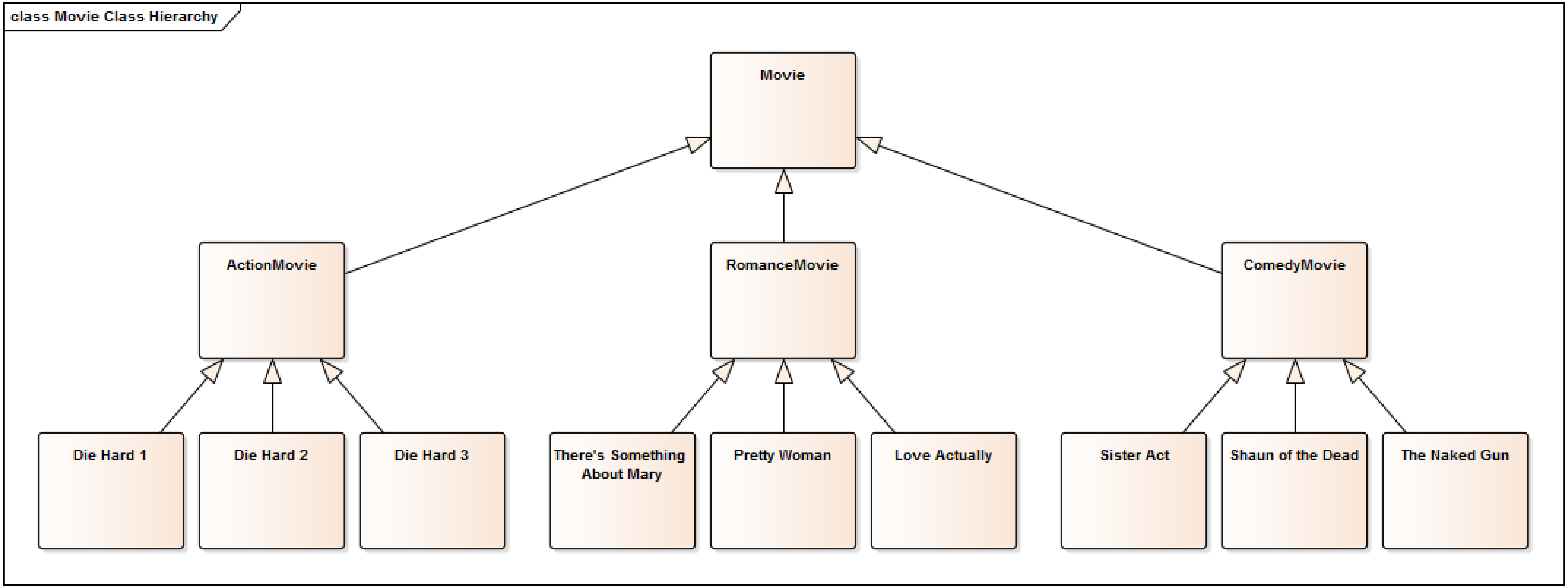
ADAP C09

Design-Time vs. Test-Time vs. Run-Time

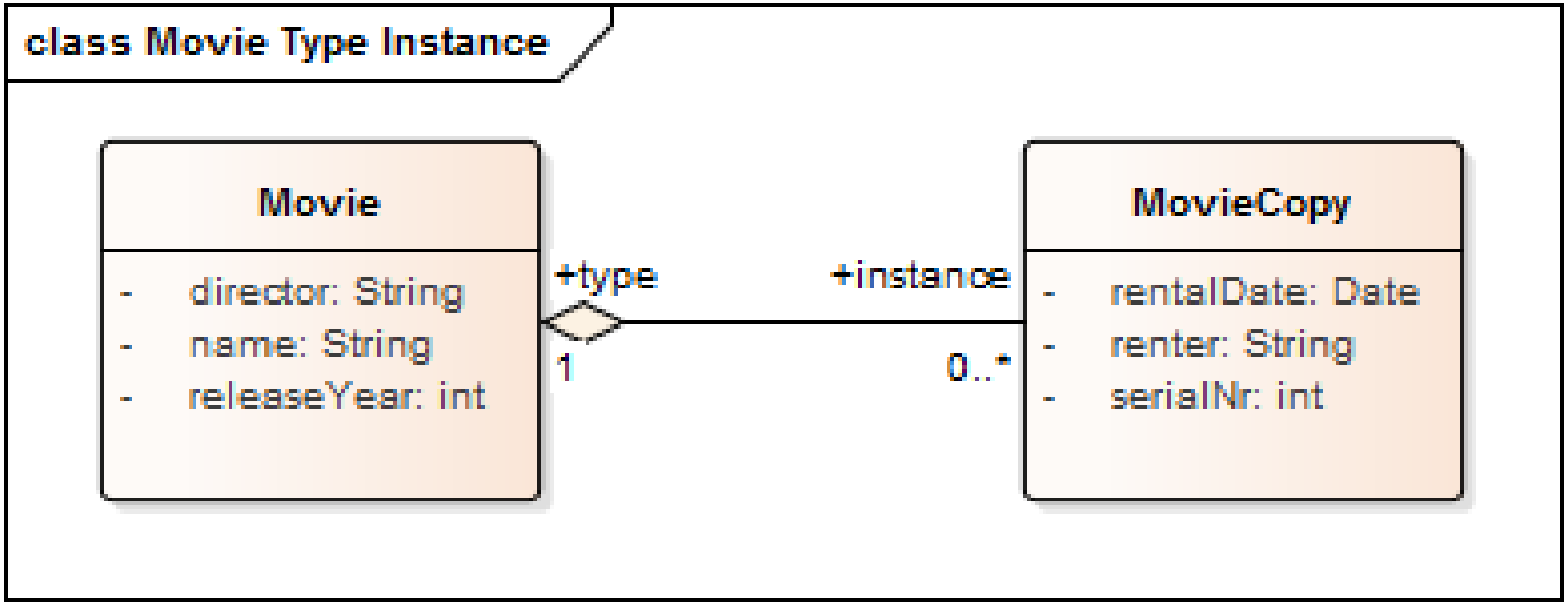


- **Design-Time**
  - **Change classes**
- Test-Time
  - Find and file bugs
- **Run-Time**
  - **Change objects**
    - Class loading
    - Configuration
    - Execution

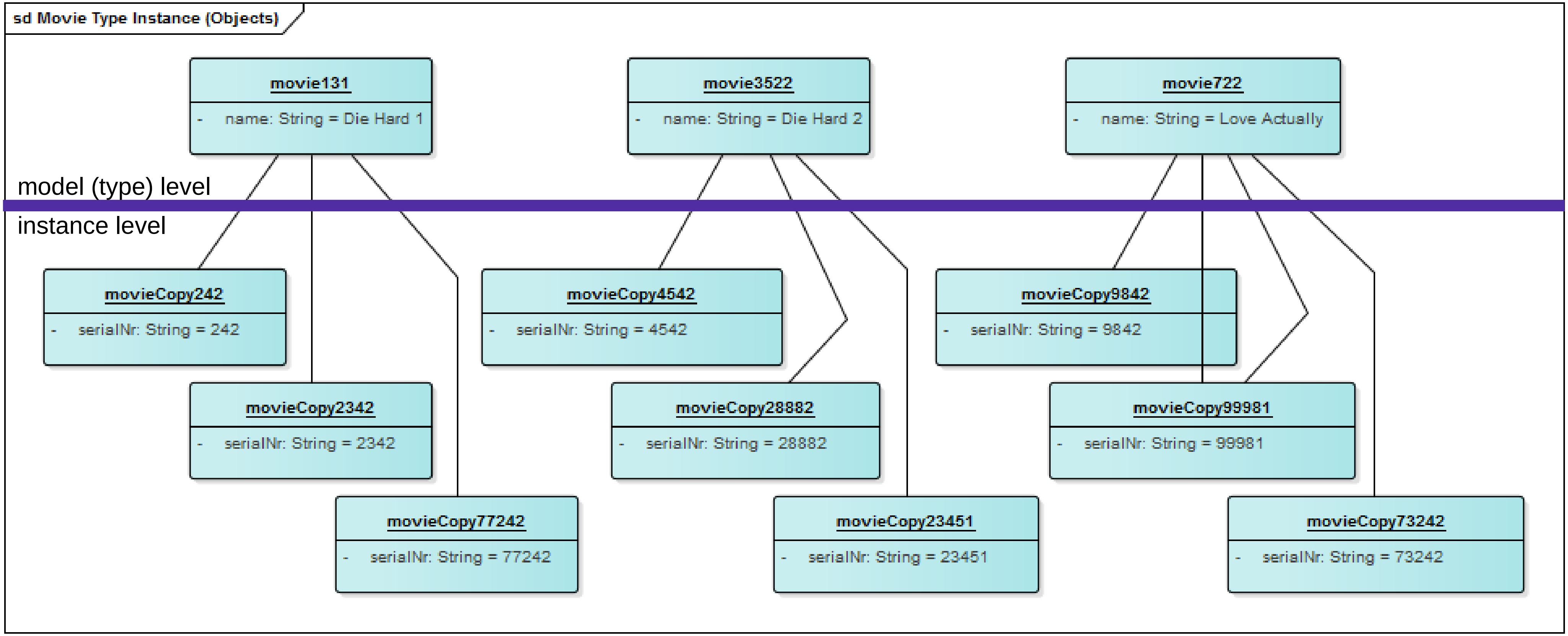
Exploding Class Hierarchies

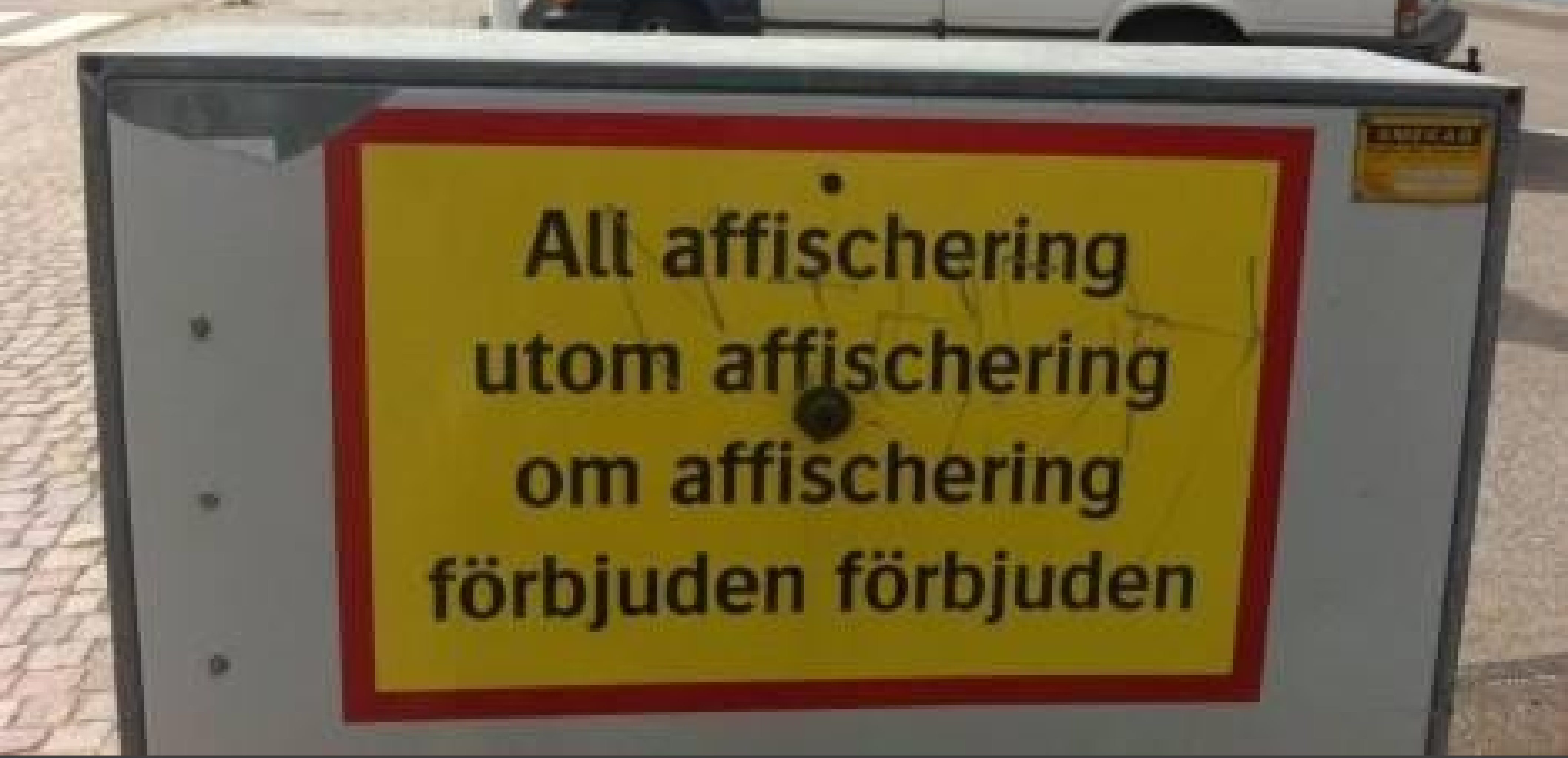


Design-Time / Class Model



Run-Time / Instances


sd Movie Type Instance (Objects)
movie131
- name: String = Die Hard 1
movie3522
- name: String = Die Hard 2
movie722
- name: String = Love Actually
model (type) level
instance level
movieCopy242
- serialNr: String = 242
movieCopy4542
- serialNr: String = 4542
movieCopy9842
- serialNr: String = 9842
movieCopy2342
- serialNr: String = 2342
movieCopy28882
- serialNr: String = 28882
movieCopy99981
- serialNr: String = 99981
movieCopy77242
- serialNr: String = 77242
movieCopy23451
- serialNr: String = 23451
movieCopy73242
- serialNr: String = 73242

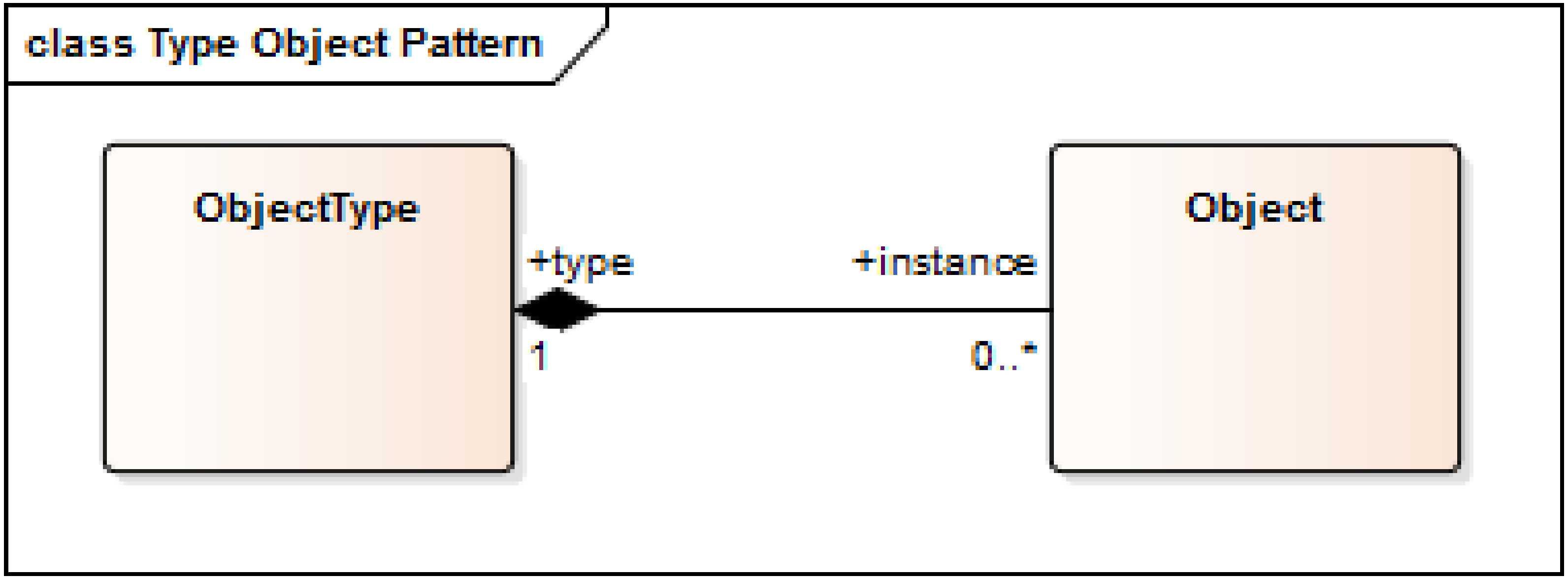


“All posters except posters about posters being prohibited are prohibited.”

Intent of Type Object Pattern [JW98]

Decouple instances from their classes so that those classes can be implemented as instances of a class. Type Object allows new “classes” to be created dynamically at runtime, lets a system provide its own type-checking rules, and can lead to simpler, smaller systems.

Structure of Type Object Pattern



Collaborations of Type Object Pattern

- Object
  - Provides instance specific functionality
  - Delegates type-specific requests to type object
- ObjectType
  - Handles type-common requests for instances
  - May create and/or manage its instances

Examples of Type Object Pattern

- Object and Class
  - java.lang.Object: base object class (instances)
  - java.lang.Class: Java object type-object-class
- Flower and FlowerType
  - org.wahlzeit.flowers.Flower: flower object class (instances)
  - org.wahlzeit.flowers.FlowerType: flower type-object-class
- PersonRole and PersonRoleType
  - com.app.model.PersonRole: person-role class
  - com.app.model.PersonRoleType: person-role type-object-class

Quiz: Defining Keyboard Models [1]

- You are developing software for configuring computers. You are implementing a Keyboard class to represent a keyboard that a customer might choose. However, there are many types of keyboards available and new types keep coming up.

  Using the Type Object pattern, how would you design the Keyboard class to make it easy to introduce new keyboard types later on?

  Select all correct statements.

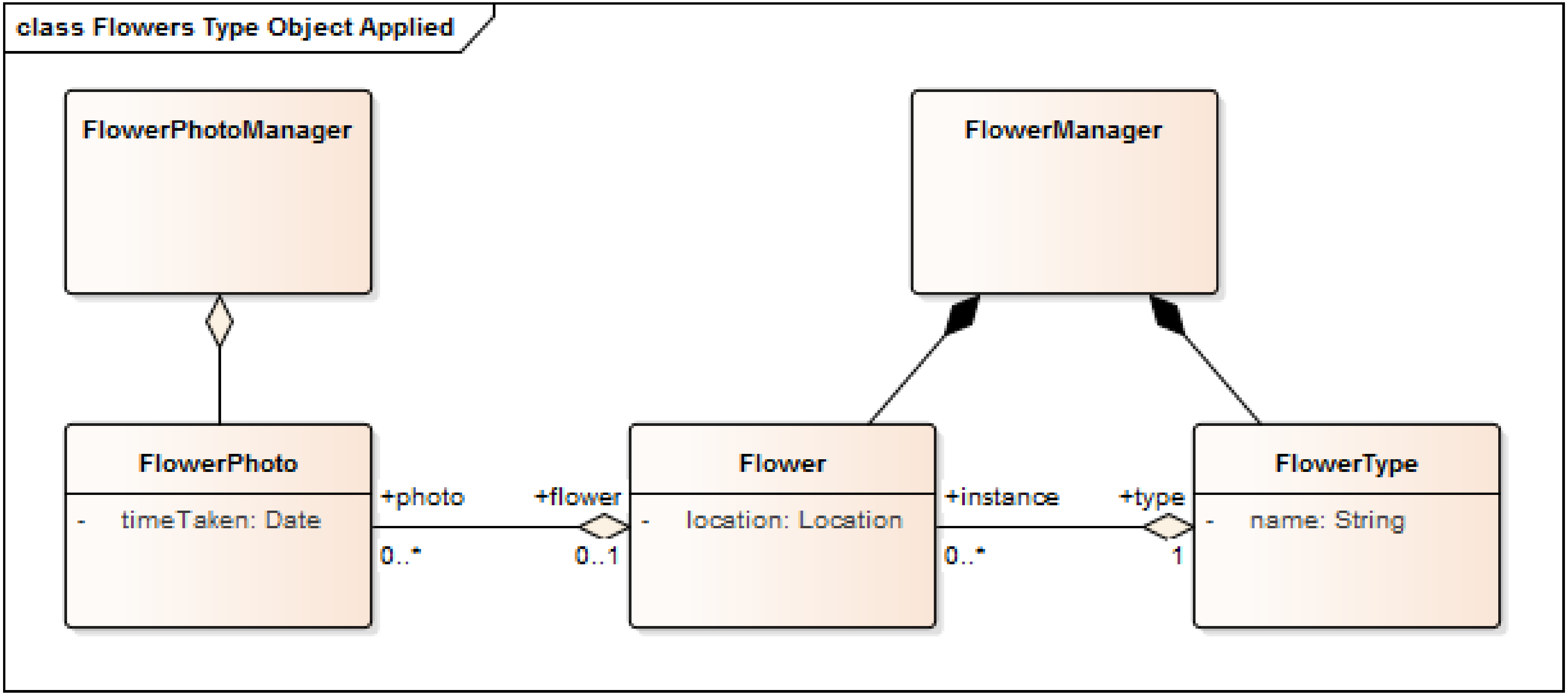
  - The Keyboard class defines two string attributes, model and make.
  - A separate KeyboardType class defines two attributes, model and make.
  - The Keyboard class defines a reference to a KeyboardType class.
  - A KeyboardType class defines a collection references to Keyboard objects.

[1] Model and make are (common) synonyms for types.

Answer: Defining Keyboard Models

- **Using the Type Object pattern, how would you design the Keyboard class to make it easy to introduce new keyboard types later on?**
  - The Keyboard class defines two string attributes, model and make.
    - **No. This type information belongs into a Type Object class.**
  - A separate KeyboardType class defines two attributes, model and make.
    - **Yes: You need a KeyboardType class to collect type information.**
    - **Maybe: Model and make are reasonable attributes of KeyboardType.**
  - The Keyboard class defines a reference to a KeyboardType class.
    - **Yes. A Keyboard object needs to access a KeyboardType object.**
      A direct reference is the easiest way to access the object.
  - A KeyboardType class defines a collection references to Keyboard objects.
    - **No. It is unusual to make the type object track its instances.**
      If anything, you'll use a Manager object for this task.

Type Object Applied to Flowers



Creating Flower Instances

```
public Flower FlowerManager#createFlower(String typeName) {
  assertIsValidFlowerTypeName(typeName);
  FlowerType ft = getFlowerType(typeName);
  Flower result = ft.createInstance(...);
  flowers.put(result.getId(), result);
  return result;
}
```

```
public Flower FlowerType#createInstance() {
  return new Flower(this);
}
```

```
protected FlowerType Flower#flowerType = null;

public Flower#Flower(FlowerType ft) {
  flowerType = ft;
}
```

Benefits of Type Object Pattern

- Reduces an exploding class hierarchy to two classes
- Allows for managing new classes (types) are runtime

Downsides of Type Object Pattern

- Increased run-time complexity
  - Makes code more difficult to read
  - Makes debugging more difficult

Quiz: Type Object Hierarchy

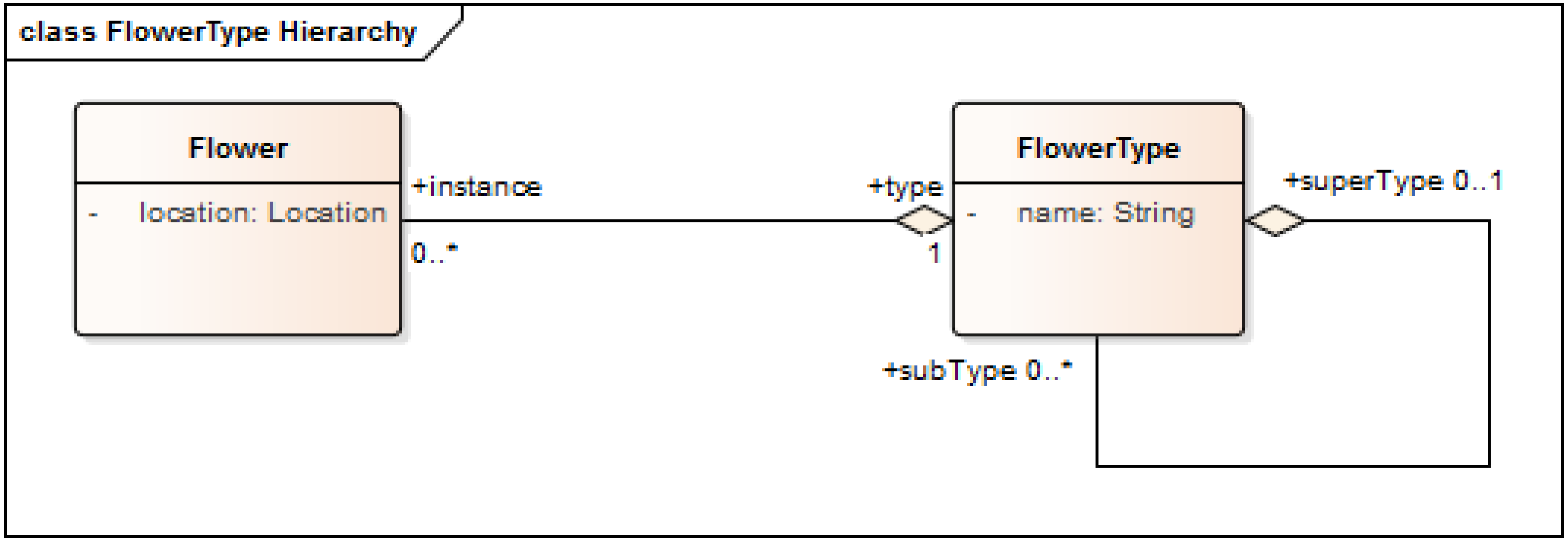
St Bernard's Lily (Anthericum liliago)
Bermuda Buttercup (Oxalis pes-caprae)
Oleander (Nerium oleander)
Lantana (Lantana camara)
Scarlet Pimpernel (Anagallis arvensis)
Verbascum (Verbascum sinuatum)
Common Mallow (Malva sylvestris)
Spanish Oyster (Scolymus hispanicum)
Stork's bill (Erodium malacoides)
Bindweed (Convolvulus arvensis)
Blue Gem (Hebes x franciscana)
Calla Lily (Zantedeschia aethiopica)

Plants are classified (in a hierarchy) according to the *International Code of Nomenclature for Cultivated Plants.*

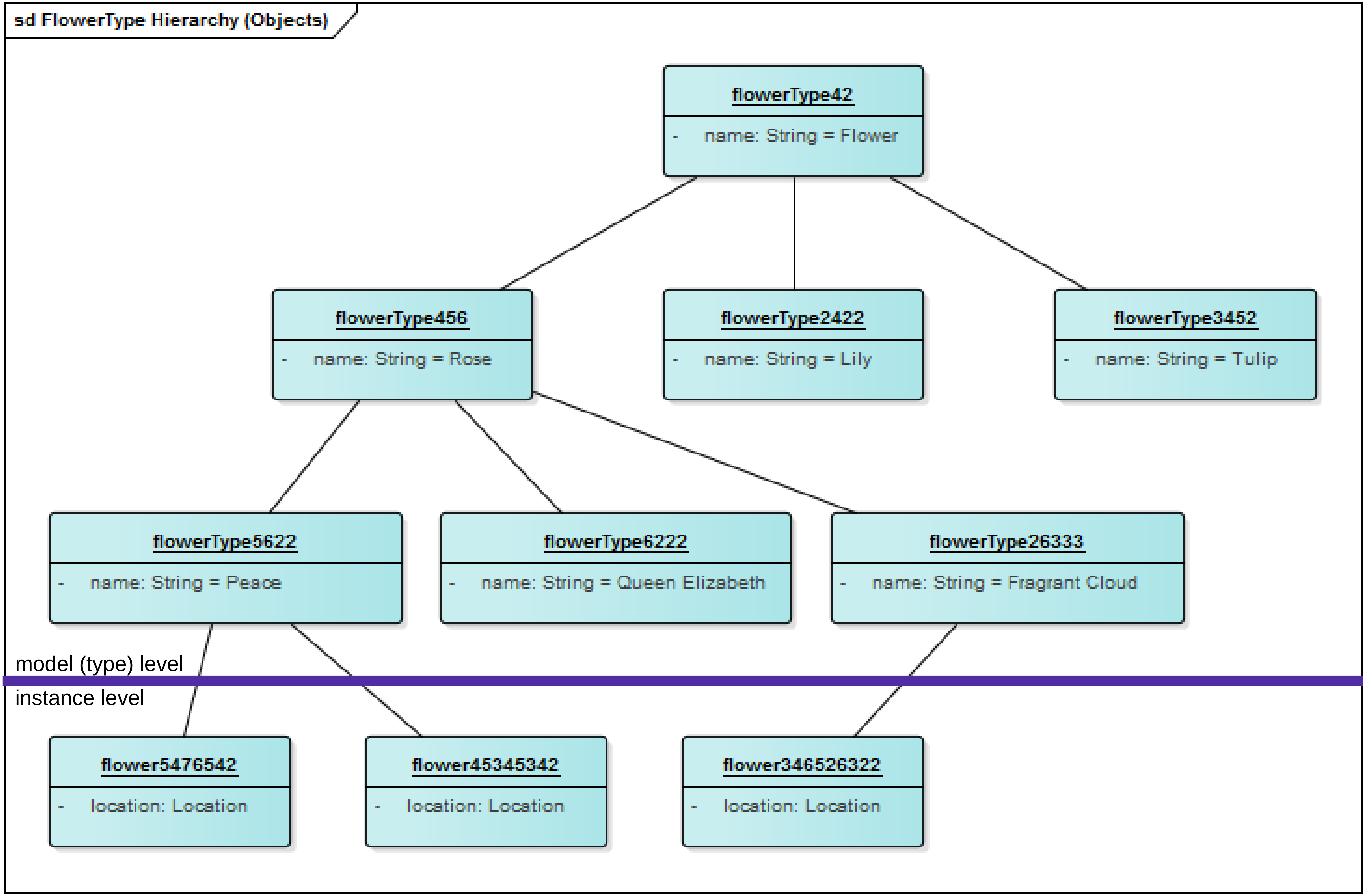
How to represent a type hierarchy for flowers in Flowers?

Flower families photo courtesy of Wikipedia

Answer 1 / 2: Type Object Hierarchy



Answer 2 / 2: Type Object Hierarchy

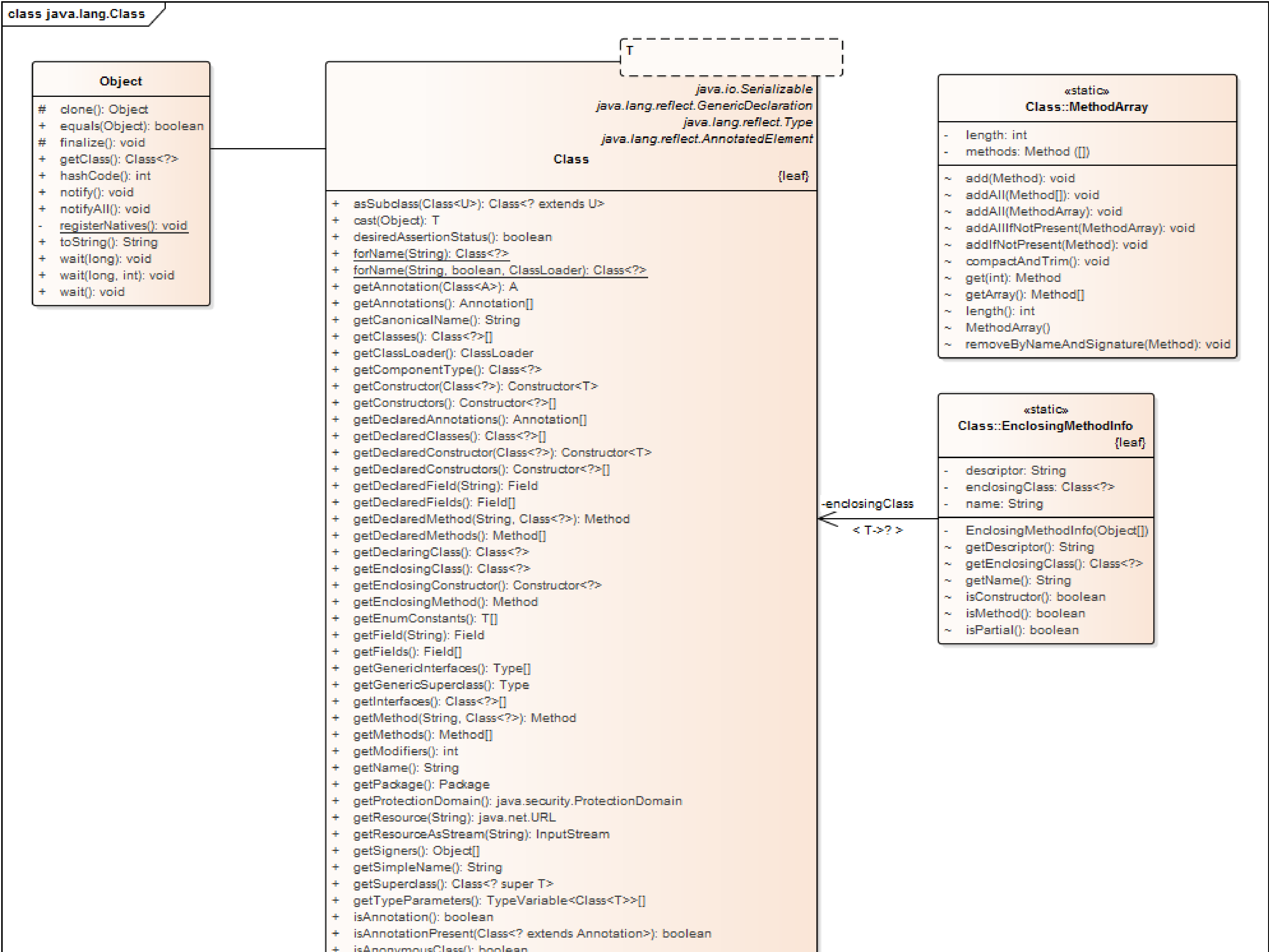


Implementing the FlowerType Hierarchy

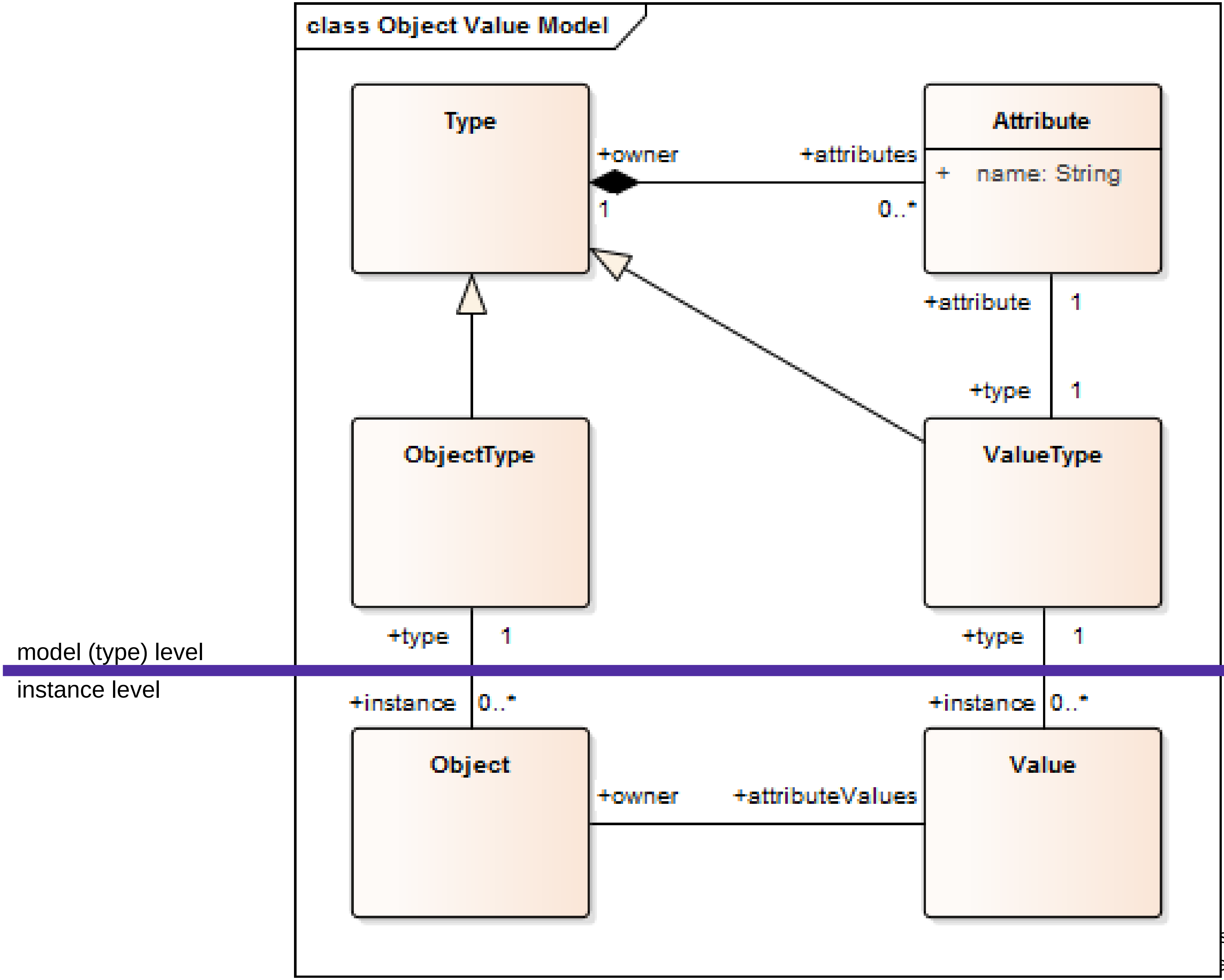
```
public class FlowerType extends DataObject {
  protected FlowerType superType = null;
  protected Set<FlowerType> subTypes = new HashSet<FlowerType>();

  public FlowerType getSuperType() {
    return superType;
  }

  public Iterator<FlowerType> getSubTypeIterator() {
    return subTypes.iterator();
  }

  public void addSubType(FlowerType ft) {
    assert (ft != null) : “tried to set null sub-type”;
    ft.setSuperType(this);
    subTypes.add(ft);
  }

  ...
}
```

Using a FlowerType Object

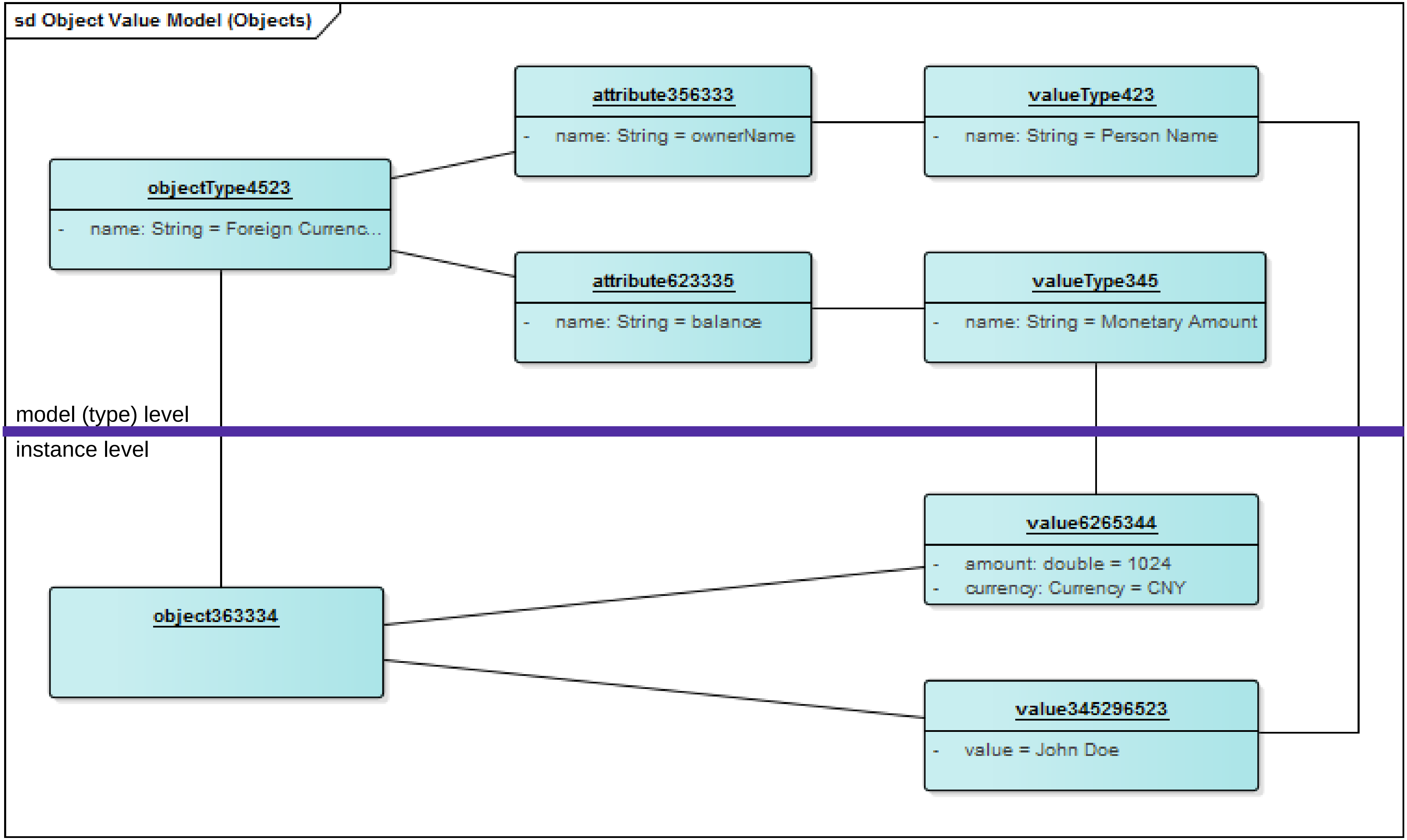
```
public class FlowerType {

  public boolean hasInstance(Flower flower) {
    assert (flower != null) : “asked about null object”;

    if (flower.getType() == this) {
      return true;
    }

    for (FlowerType type : subTypes) {
      if (type.hasInstance(flower)) {
        return true;
      }
    }

    return false;
  }

  ...
}
```


class java.lang.Class
Object
# clone(): Object
+ equals(Object): boolean
# finalize(): void
+ getClass(): Class<?>
+ hashCode(): int
+ notify(): void
+ notifyAll(): void
- registerNatives(): void
+ toString(): String
+ wait(long): void
+ wait(long, int): void
+ wait(): void
T
java.io.Serializable
java.lang.reflect.GenericDeclaration
java.lang.reflect.Type
java.lang.reflect.AnnotatedElement
Class
{leaf}
+ asSubclass(Class<U>): Class<? extends U>
+ cast(Object): T
+ desiredAssertionStatus(): boolean
+ forName(String): Class<?>
+ forName(String, boolean, ClassLoader): Class<?>
+ getAnnotation(Class<A>): A
+ getAnnotations(): Annotation[]
+ getCanonicalName(): String
+ getClasses(): Class<?>[]
+ getClassLoader(): ClassLoader
+ getComponentType(): Class<?>
+ getConstructor(Class<?>): Constructor<T>
+ getConstructors(): Constructor<?>[]
+ getDeclaredAnnotations(): Annotation[]
+ getDeclaredClasses(): Class<?>[]
+ getDeclaredConstructor(Class<?>): Constructor<T>
+ getDeclaredConstructors(): Constructor<?>[]
+ getDeclaredField(String): Field
+ getDeclaredFields(): Field[]
+ getDeclaredMethod(String, Class<?>): Method
+ getDeclaredMethods(): Method[]
+ getDeclaringClass(): Class<?>
+ getEnclosingClass(): Class<?>
+ getEnclosingConstructor(): Constructor<?>
+ getEnclosingMethod(): Method
+ getEnumConstants(): T[]
+ getField(String): Field
+ getFields(): Field[]
+ getGenericInterfaces(): Type[]
+ getGenericSuperclass(): Type
+ getInterfaces(): Class<?>[]
+ getMethod(String, Class<?>): Method
+ getMethods(): Method[]
+ getModifiers(): int
+ getName(): String
+ getPackage(): Package
+ getProtectionDomain(): java.security.ProtectionDomain
+ getResource(String): java.net.URL
+ getResourceAsStream(String): InputStream
+ getSigners(): Object[]
+ getSimpleName(): String
+ getSuperclass(): Class<? super T>
+ getTypeParameters(): TypeVariable<Class<T>>[]
+ isAnnotation(): boolean
+ isAnnotationPresent(Class<? extends Annotation>): boolean
+ isAnonymousClass(): boolean
«static»
Class::MethodArray
- length: int
- methods: Method ([])
~ add(Method): void
~ addAll(Method[]): void
~ addAll(MethodArray): void
~ addAllIfNotPresent(MethodArray): void
~ addIfNotPresent(Method): void
~ compactAndTrim(): void
~ get(int): Method
~ getArray(): Method[]
~ length(): int
~ MethodArray()
~ removeByNameAndSignature(Method): void
-enclosingClass
< T->? >
«static»
Class::EnclosingMethodInfo
{leaf}
- descriptor: String
- enclosingClass: Class<?>
- name: String
- EnclosingMethodInfo(Object[])
~ getDescriptor(): String
~ getEnclosingClass(): Class<?>
~ getName(): String
~ isConstructor(): boolean
~ isMethod(): boolean
~ isPartial(): boolean

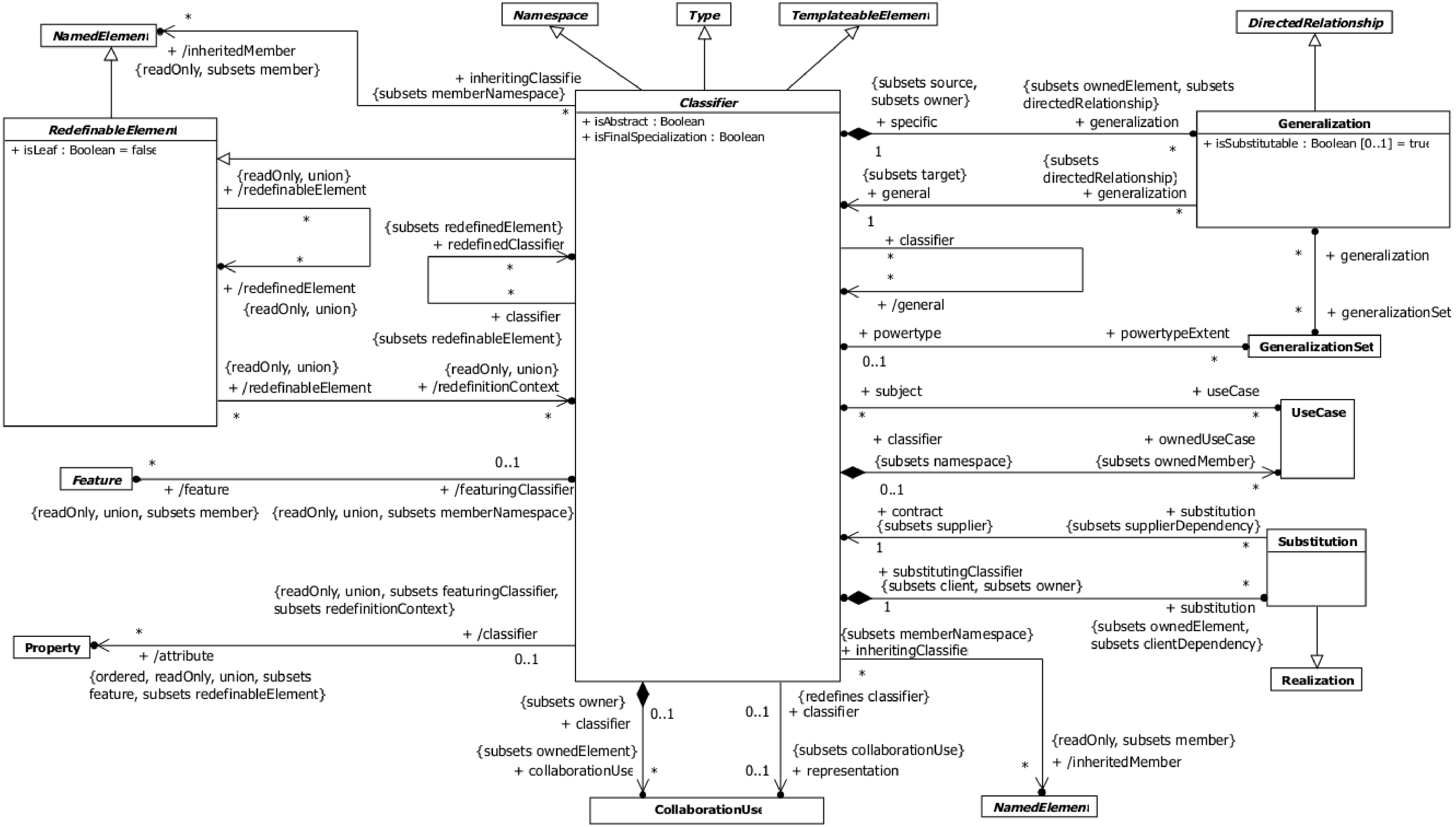
Simple Object Value Model



Example of Simple Object Value Model



The Classifier Part of the UML Metamodel



Java, UML, Flowers

M2 Language Level	M1 Model / Code Level	M0 Run-time / Obj. Level
java.lang.Class ...	java.lang.Object flowers.FlowerPhoto flowers.FlowerType flowers.Flower ...	fragrantCloud : FlowerType flower34563 : Flower ...
uml.Classifier uml.Generalization uml.StereoType uml.Component ...	flowers.FlowerPhoto flowers.FlowerType flowers.Flower ...	fragrantCloud : FlowerType flower34563 : Flower ...
flowers.FlowerType ...	flowers.Flower : FlowerType fragrantCloud : FlowerType ...	flower34563 : Flower ...

Model / Instance Relationships (Static)

M2 Language Level	M1 Model / Code Level	M0 Run-time / Obj. Level
java.lang.Class ...	java.lang.Object flowers.FlowerPhoto flowers.FlowerType flowers.Flower ...	fragrantCloud : FlowerType flower34563 : Flower ...
uml.Classifier uml.Generali uml.St um ...	flowers.FlowerPhoto flowers.FlowerType	fragrantCloud : FlowerType flower3456 wer
flowers.FlowerType ...	flowers.Flower : FlowerType fragrantCloud : FlowerType ...	flower34563 : Flower ...

static model / instance relationships

Model / Instance Relationships (Dynamic)

M2 Language Level	M1 Model / Code Level	M0 Run-time / Obj. Level
java.lang.Class ...	java.lang.Object flowers.FlowerPhoto flowers.FlowerType flowers.Flower ...	fragrantCloud : FlowerType flower34563 ver ...
uml.Classifier uml.Generalization uml.StereoType uml.Component ...	flowers.FlowerPhoto flowers.FlowerType flowers.Flower ...	fragrantCl erType flower345 ...
flowers.FlowerType ...	flowers.Flower : FlowerType fragrantCloud : FlowerType ...	flower345 ...

dynamic model / instance relationships

Vocabulary

- Metaclass, class and object
  - Usually used in absolute terms, covering levels M2, M1, M0
    - Metaclass = element of M2 level (language level)
    - Class = element of M1 level (model level)
    - Object = element of M0 level (object/instance/run-time level)
- Meta-object and base-object
  - Usually use as relative terms: the meta-object describes the base-object
    - A meta-object can be the base-object for another meta-object
- Type and instance
  - Usually used as relative terms, similary to meta and base-object

Meta-Object Protocols (MOPs)

- Introspection
  - Provide information about base objects
- Intercession
  - Manipulate structure and behavior of base objects

Review / Summary of Session

- Type object design pattern
  - Definition, purpose, examples
  - In application domains (movies, flowers)
  - In technical domains (object/class, UML)
- Meta-modeling
  - UML metamodel
  - Static and dynamic relationships

Thanks! Questions?

dirk.riehle@fau.de – http://osr.cs.fau.de

dirk@riehle.org – http://dirkriehle.com – @dirkriehle

Credits and License

- Original version
  -
  -

- Contributions
  - ...