

# Type Objects

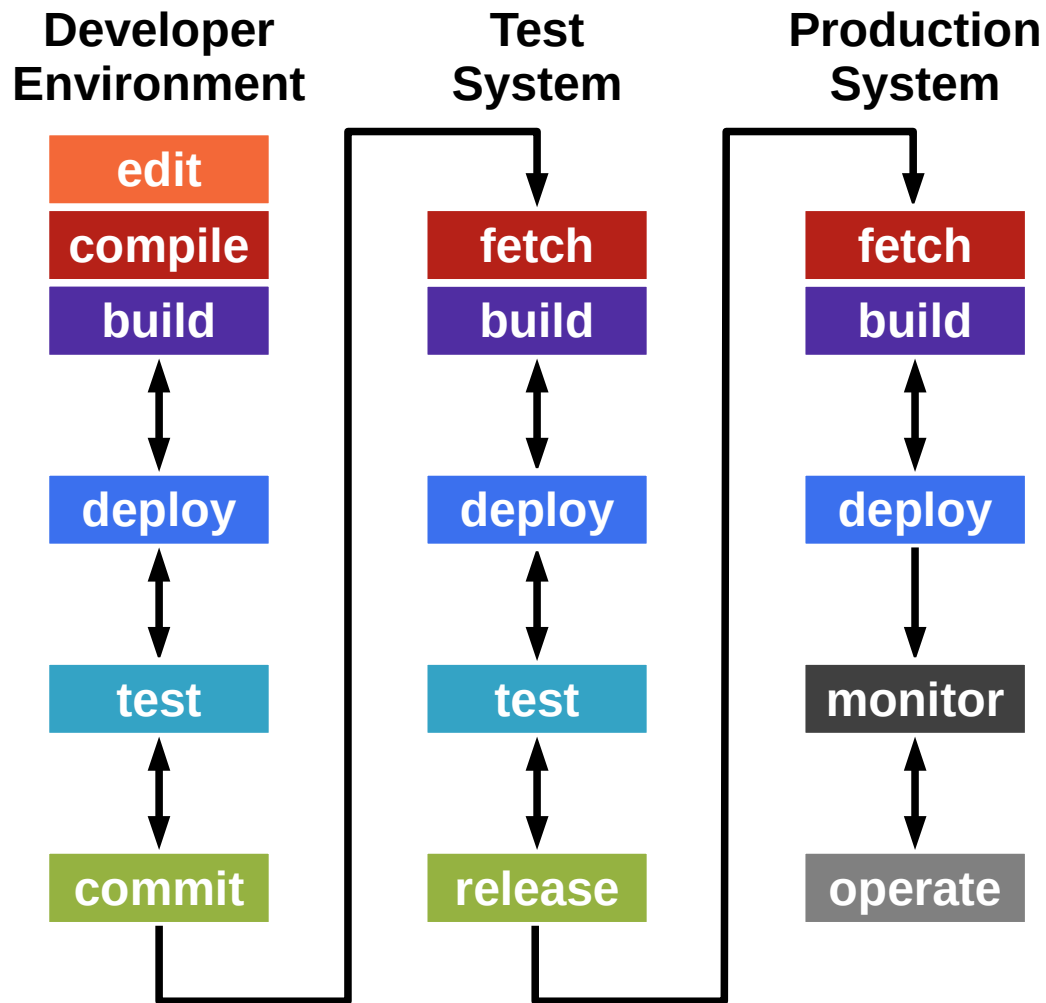
**Prof. Dr. Dirk Riehle**

**Friedrich-Alexander University Erlangen-Nürnberg**

**ADAP C09**

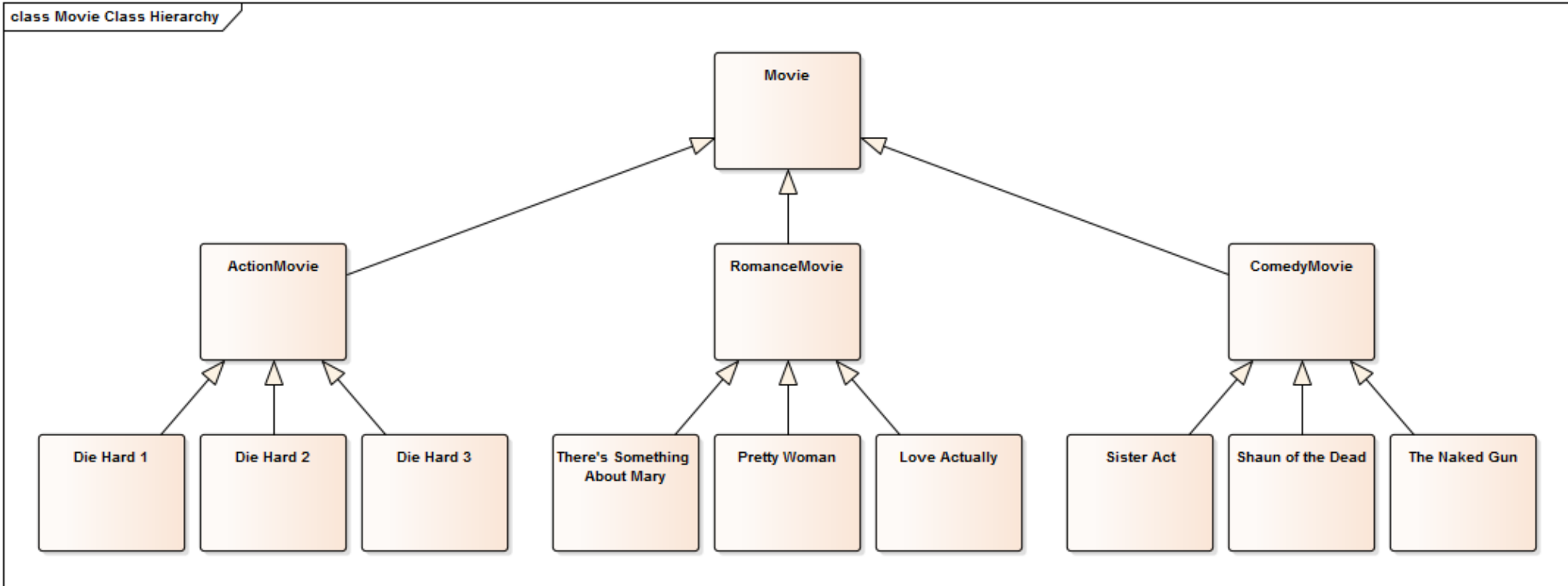
Licensed under [CC BY 4.0 International](https://creativecommons.org/licenses/by/4.0/)

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

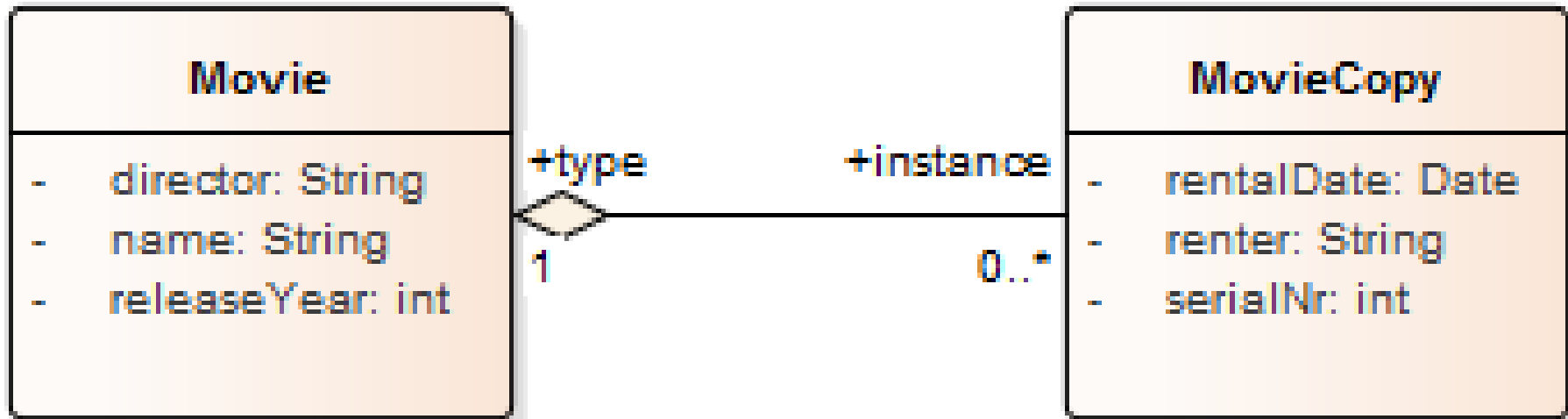


- **Design-Time**
  - **Change classes**
- Test-Time
  - Find and file bugs
- **Run-Time**
  - **Change objects**
    - Class loading
    - Configuration
    - Execution

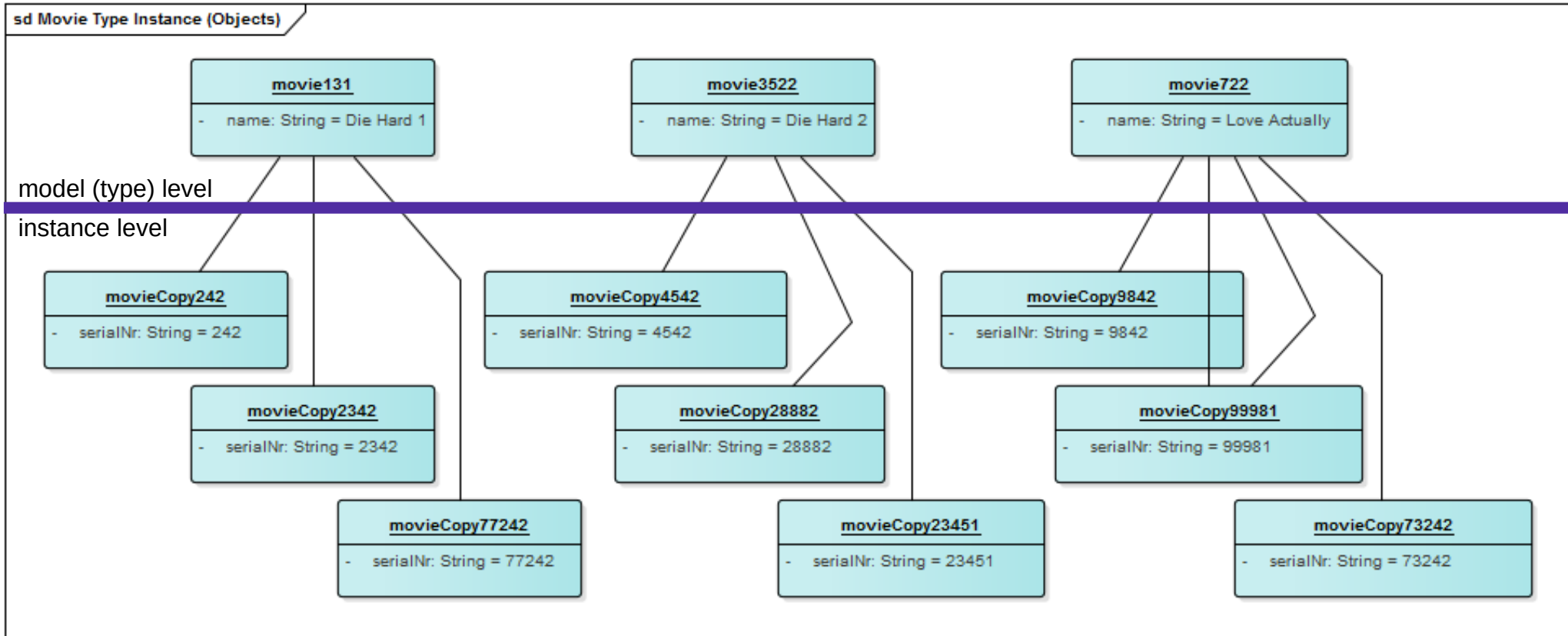
# Exploding Class Hierarchies




class Movie Type Instance



# Run-Time / Instances



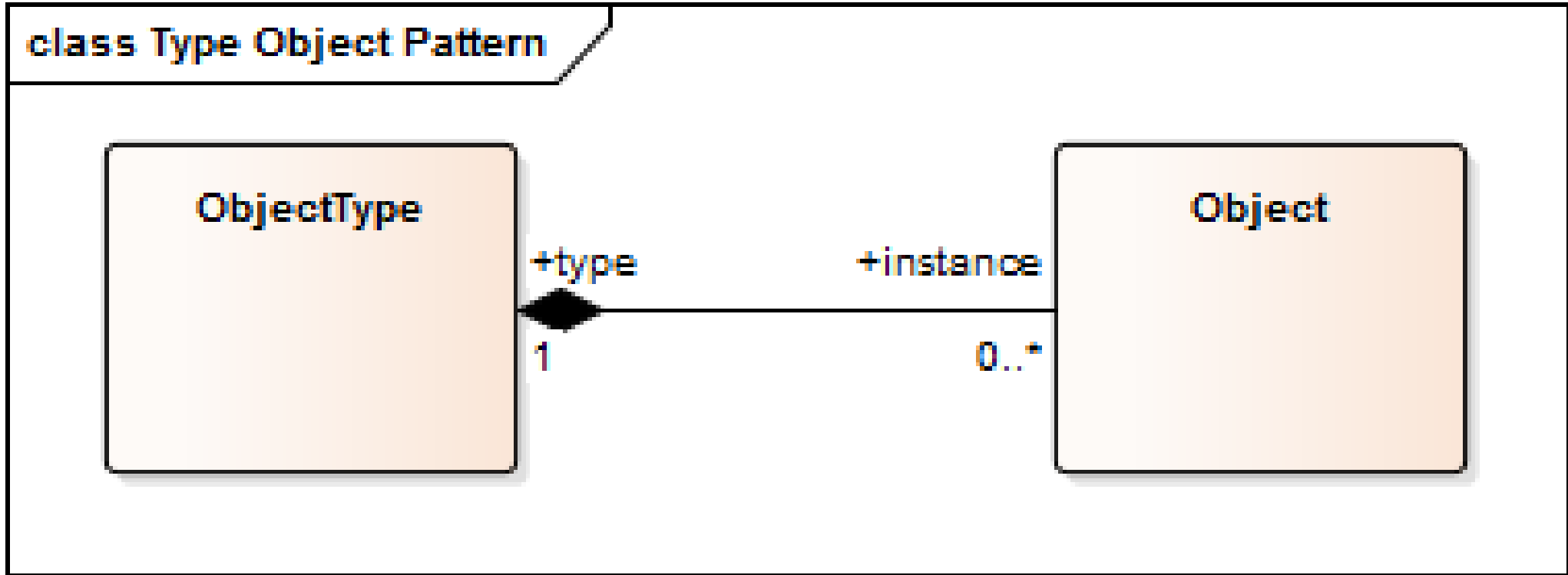
**“All posters except posters about  
posters being prohibited are prohibited.”**



**All affischering  
utom affischering  
om affischering  
förbjuden förbjuden**

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?**

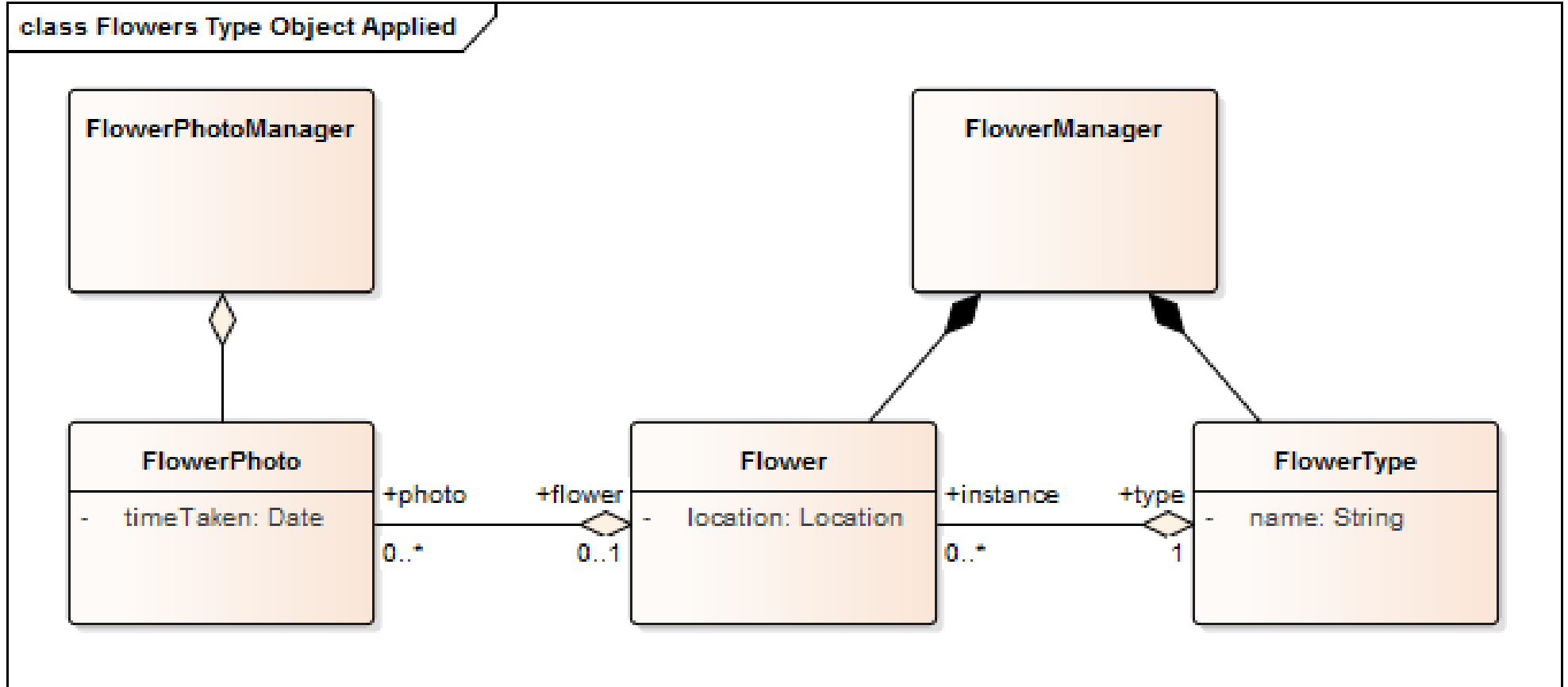
Select all correct statements.

- 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.

# 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) at runtime

# Downsides of Type Object Pattern

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



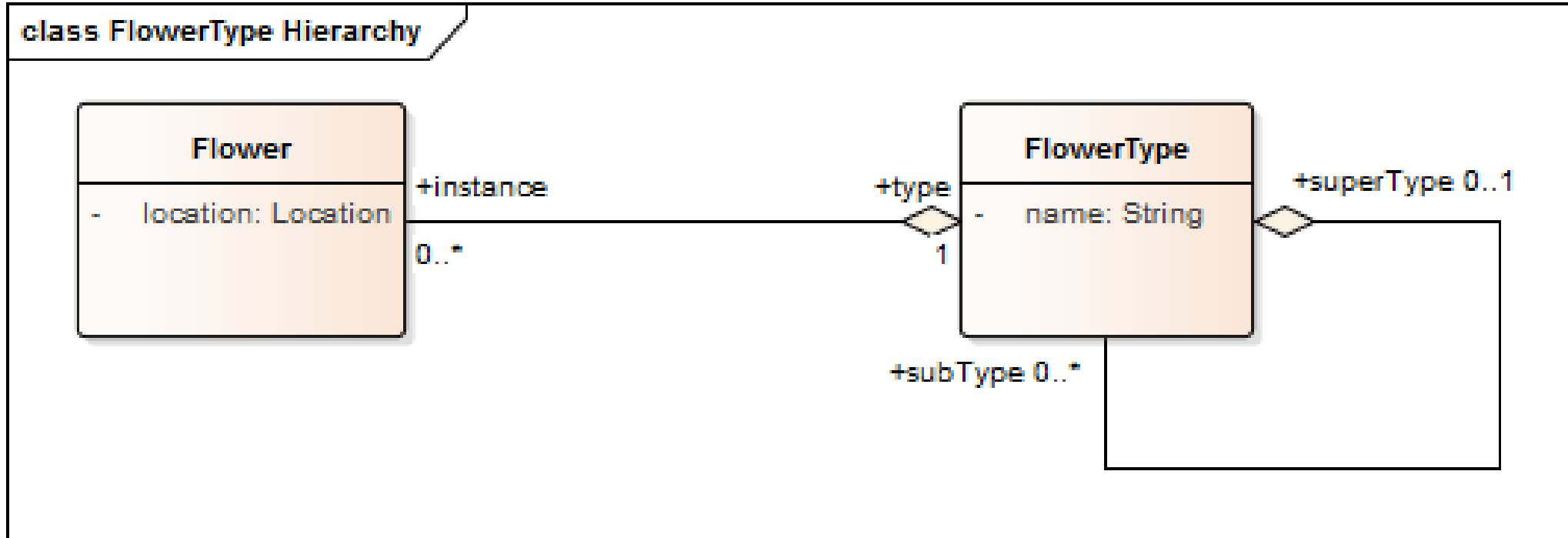
# Quiz: Type Object Hierarchy



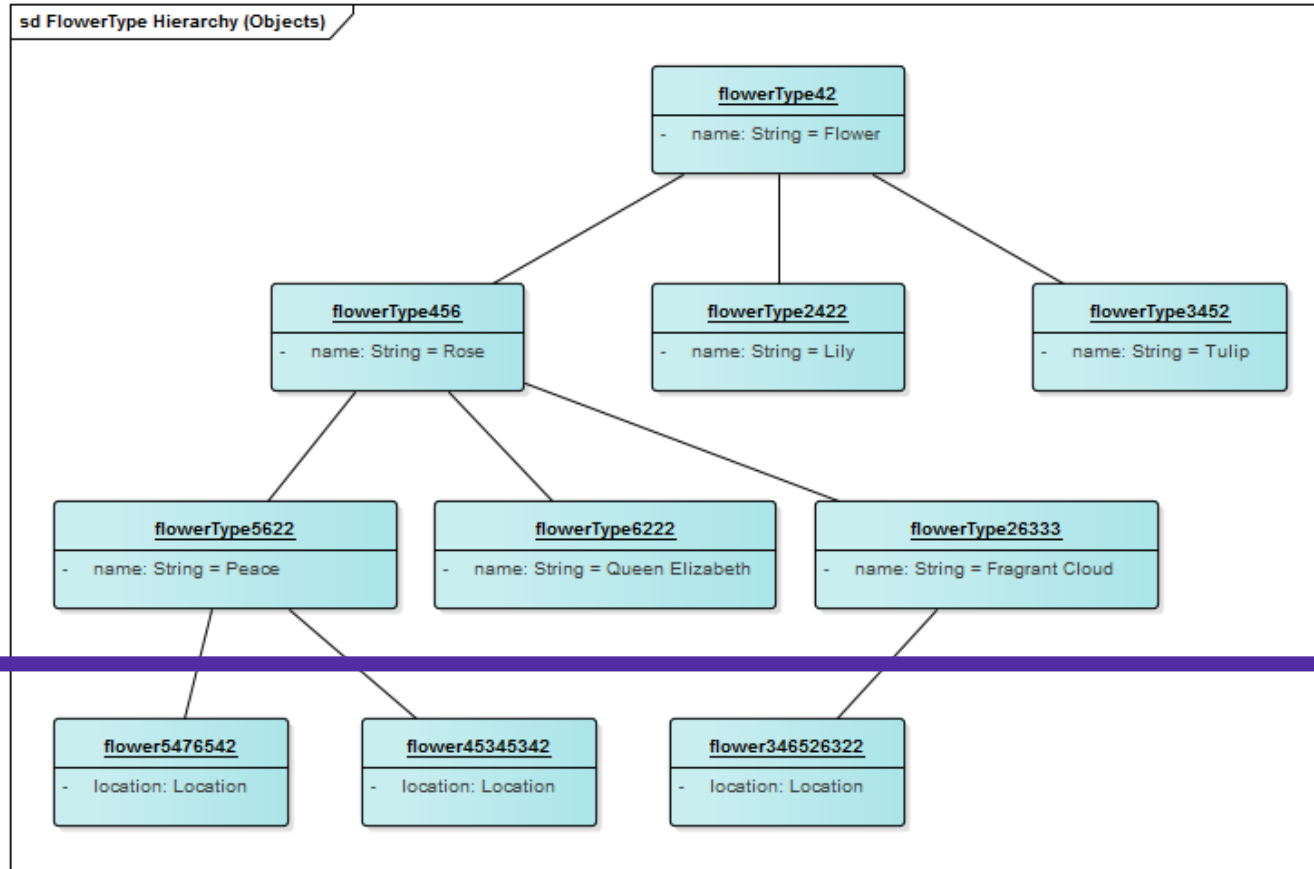
Plants are classified (in a hierarchy) according to the *International Code of Nomenclature for Cultivated Plants*.

**How to represent a type hierarchy for flowers in Flow-ers?**

## Answer 1 / 2: Type Object Hierarchy



## Answer 2 / 2: Type Object Hierarchy



model (type) level

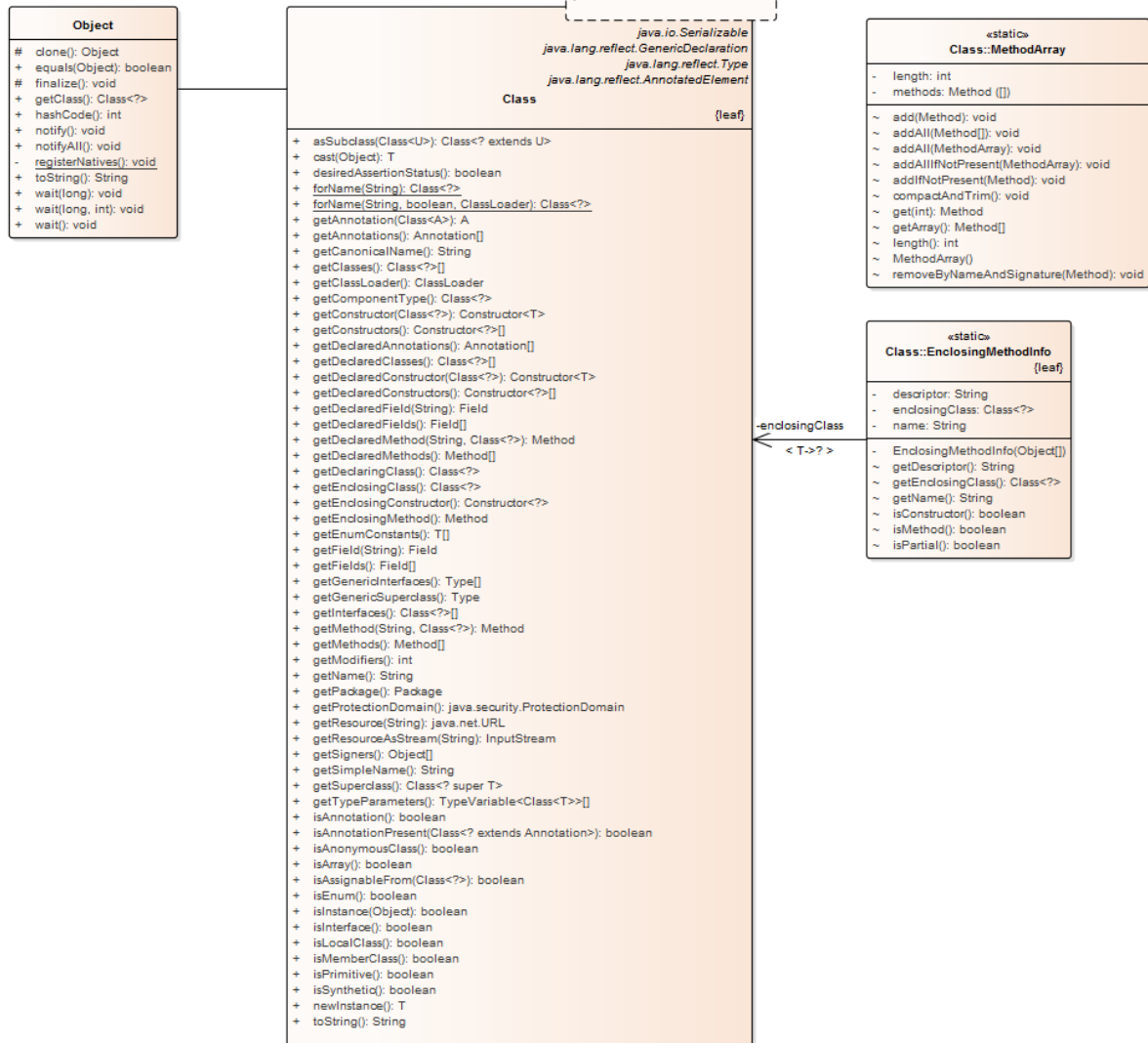
instance level

# 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

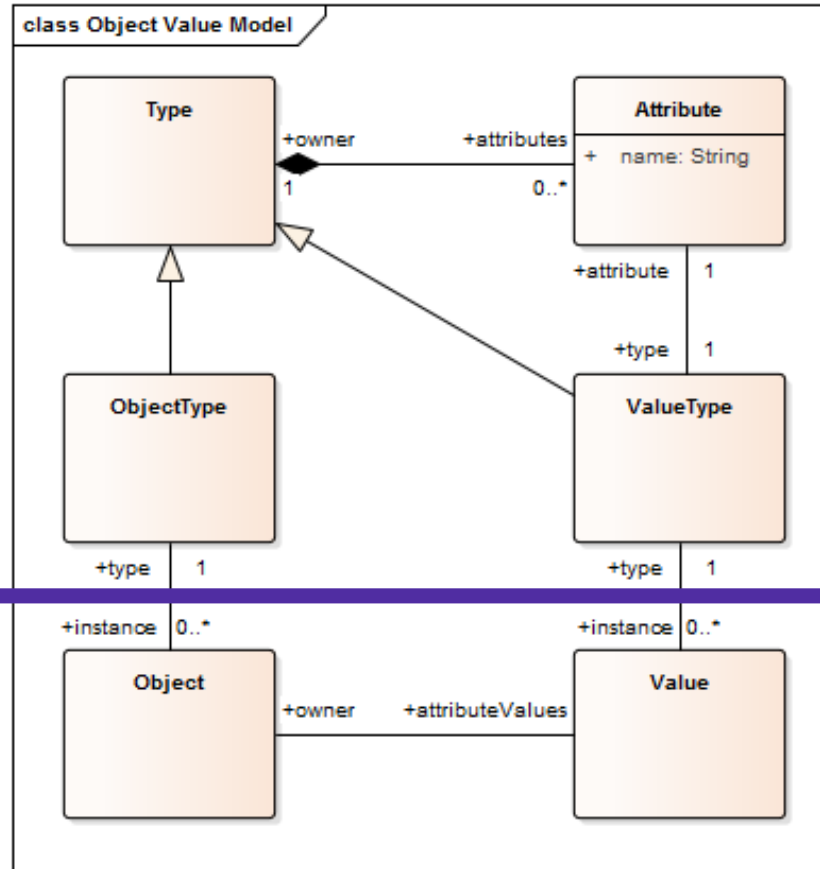
```
public class FlowerType {  
    public boolean hasInstance(Flower flower) {  
        assert (flower != null) : "asked about null object";  
  
        if (flower.getType() == this) {  
            return true;  
        }  
  
        for (FlowerType type : subTypes) {  
            if (type.hasInstance(flower)) {  
                return true;  
            }  
        }  
  
        return false;  
    }  
    ...  
}
```



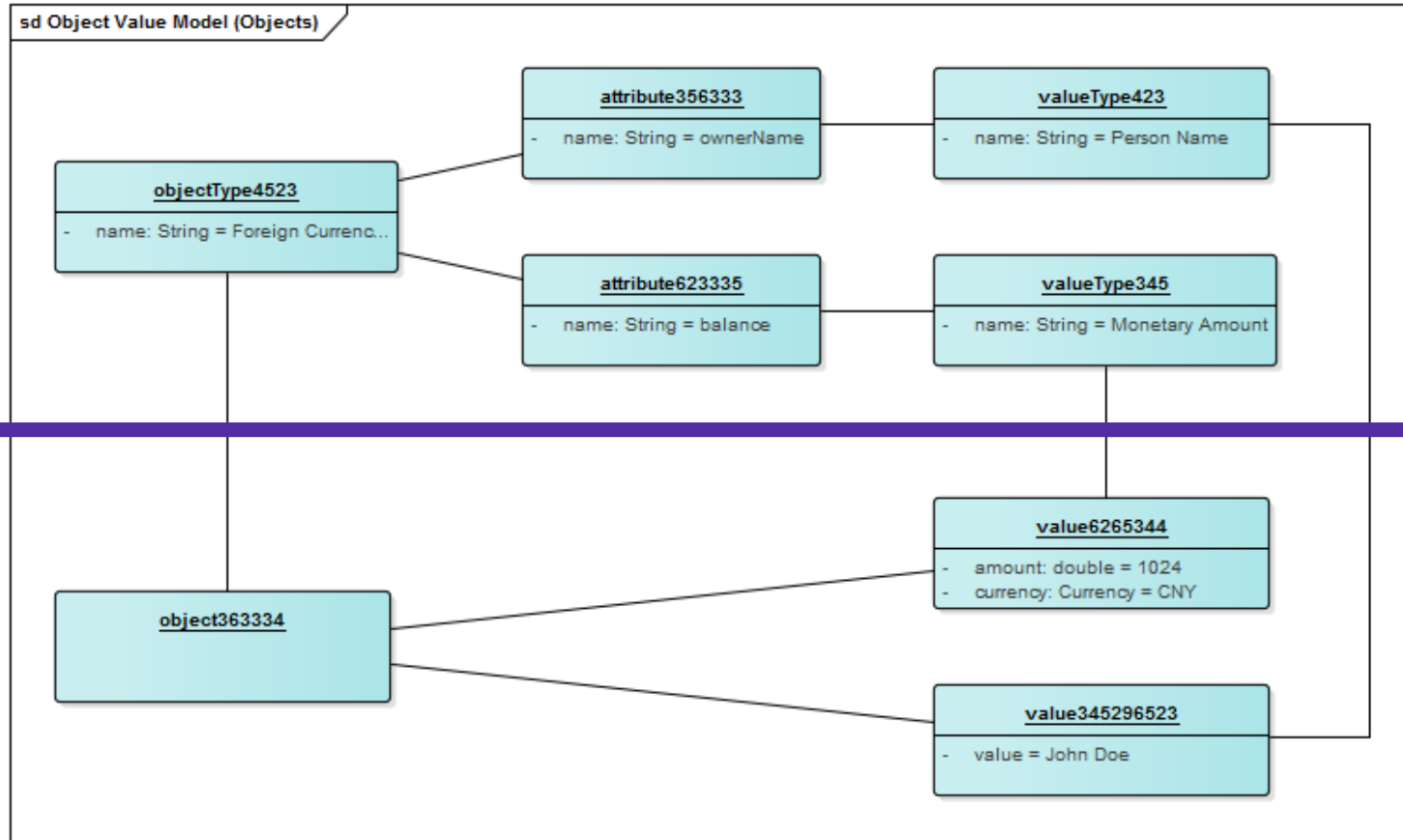
# Simple Object Value Model

model (type) level

instance level



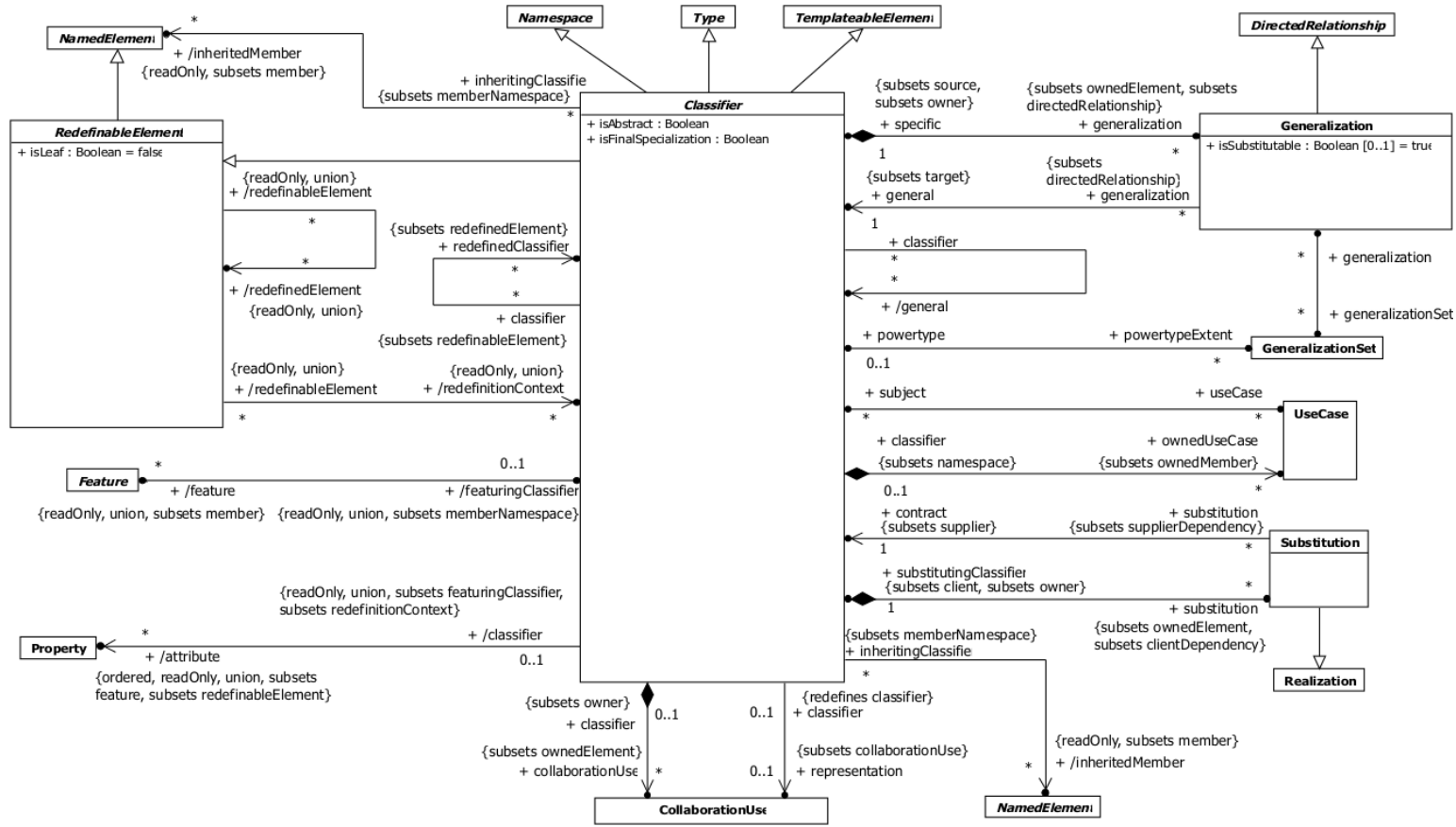
# Example of Simple Object Value Model



model (type) level  
instance level



# 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)

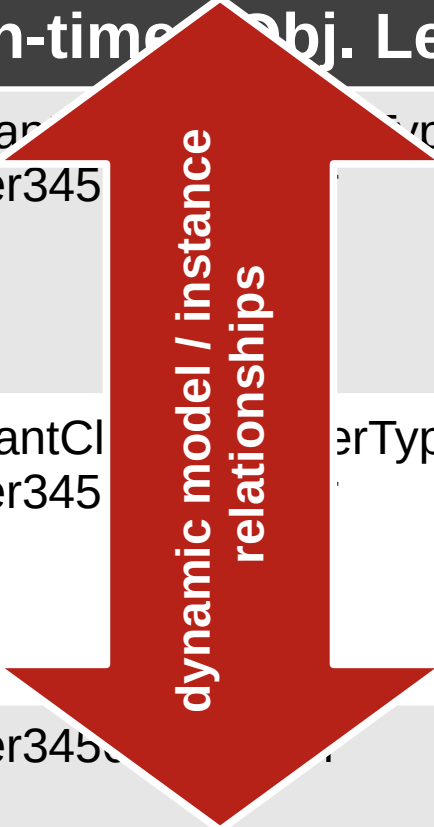
M2 Language Level	M1 Model / Code Level	M0 Run-time / Obj. Level
java.lang.Class ...	java.lang.Object flowers.FlowerPhoto flowers.FlowerType flowers.Flower	fragrantCloud : FlowerType flower34563 : Flower ...
uml. uml.Generalization uml.Stereotype uml.Component ...	flowers.Flower ...	...
flowers.FlowerType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



- 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, similar 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

# Thank you! Questions?

**[dirk.riehle@fau.de](mailto:dirk.riehle@fau.de) – <http://osr.cs.fau.de>**

**[dirk@riehle.org](mailto:dirk@riehle.org) – <http://dirkriehle.com> – [@dirkriehle](#)**



# Credits and License

- Original version
  - © 2012-2019 [Dirk Riehle](#), some rights reserved
  - Licensed under [Creative Commons Attribution 4.0 International License](#)
- Contributions
  - ...

# Type Objects

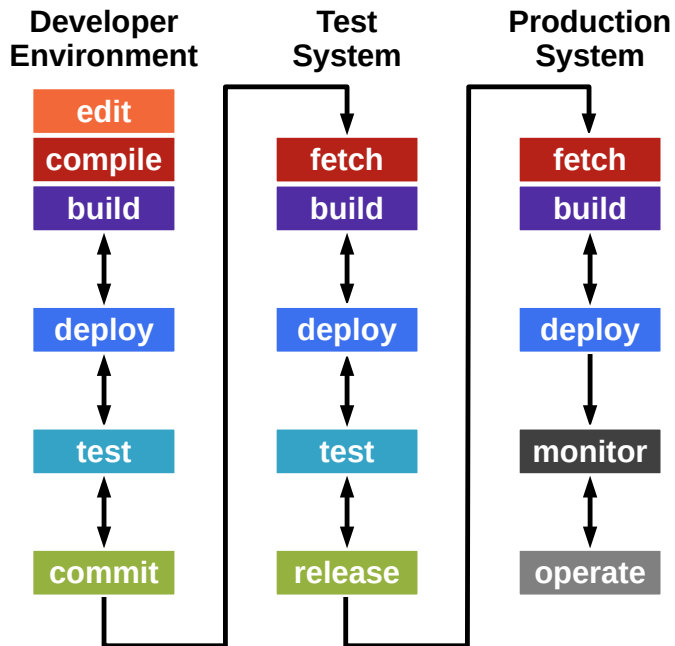
**Prof. Dr. Dirk Riehle**  
**Friedrich-Alexander University Erlangen-Nürnberg**

## **ADAP C09**

Licensed under [CC BY 4.0 International](#)

It is Friedrich-Alexander University Erlangen-Nürnberg – FAU, in short.  
Corporate identity wants us to say “Friedrich-Alexander University”.

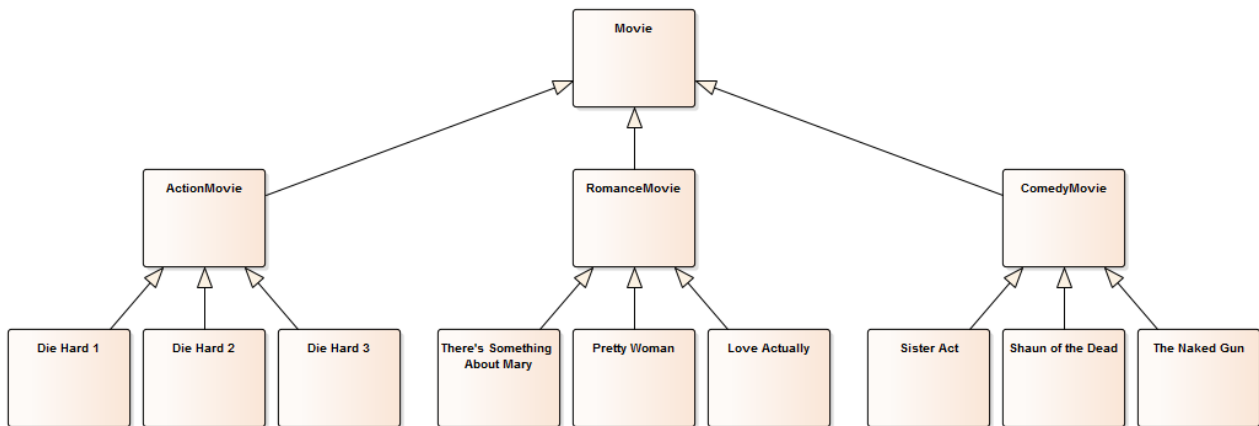
## Design-Time vs. Test-Time vs. Run-Time



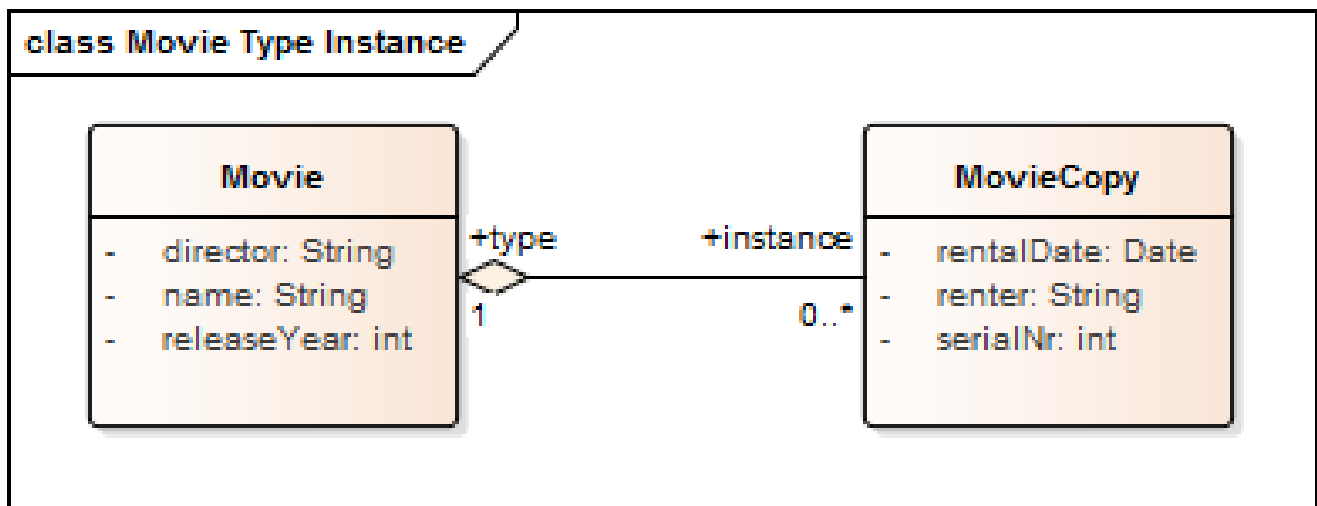
- **Design-Time**
  - **Change classes**
- **Test-Time**
  - Find and file bugs
- **Run-Time**
  - **Change objects**
    - Class loading
    - Configuration
    - Execution

## Exploding Class Hierarchies

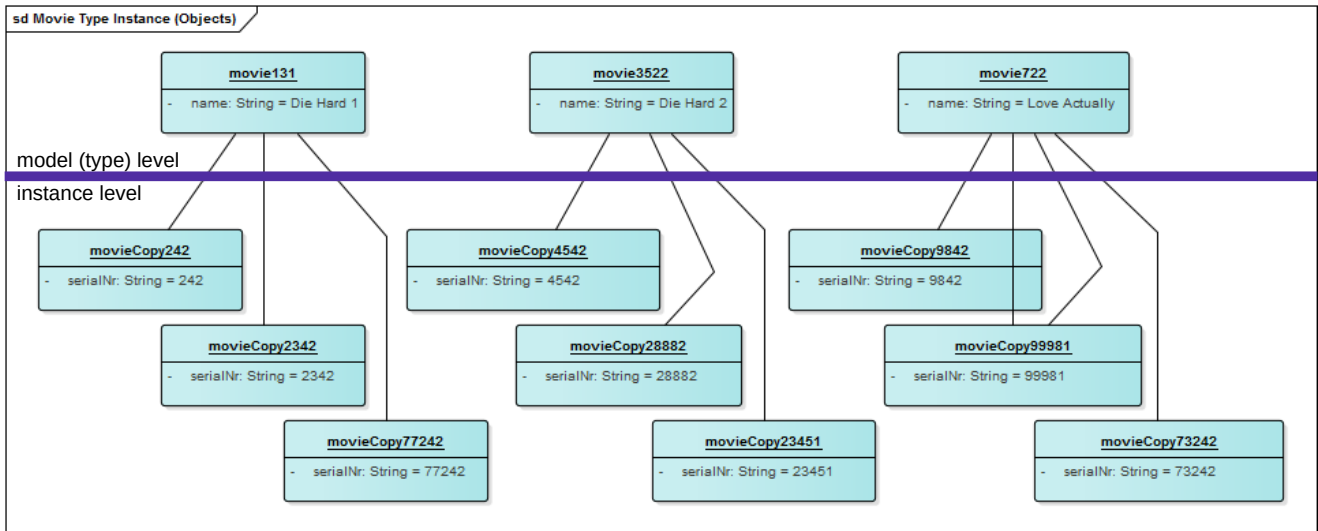
class Movie Class Hierarchy



## Design-Time / Class Model



## Run-Time / Instances



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

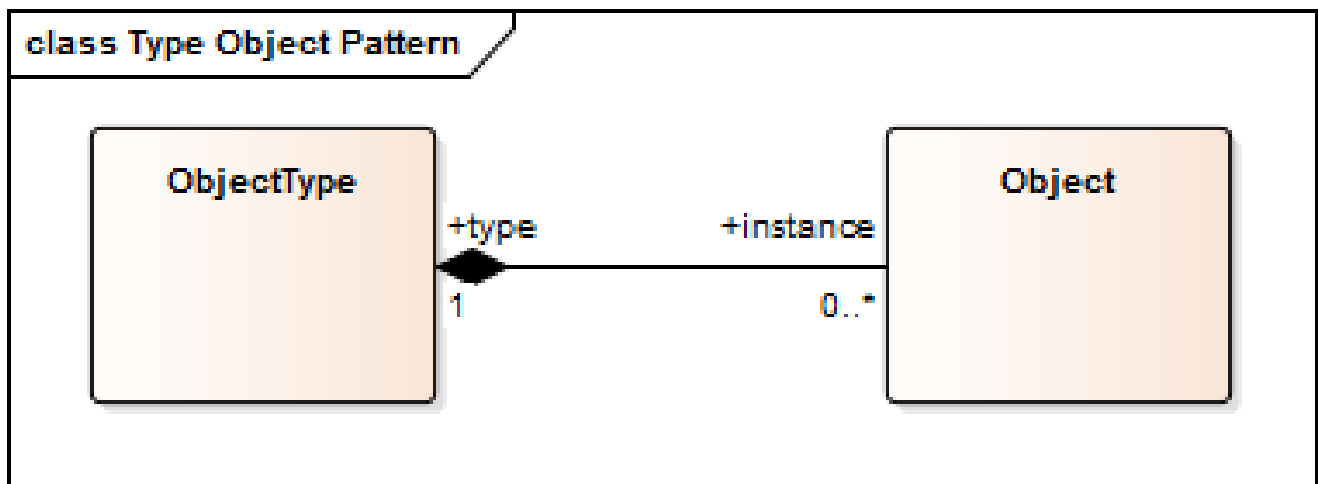


## Intent of Type Object Pattern [JW98]

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?**

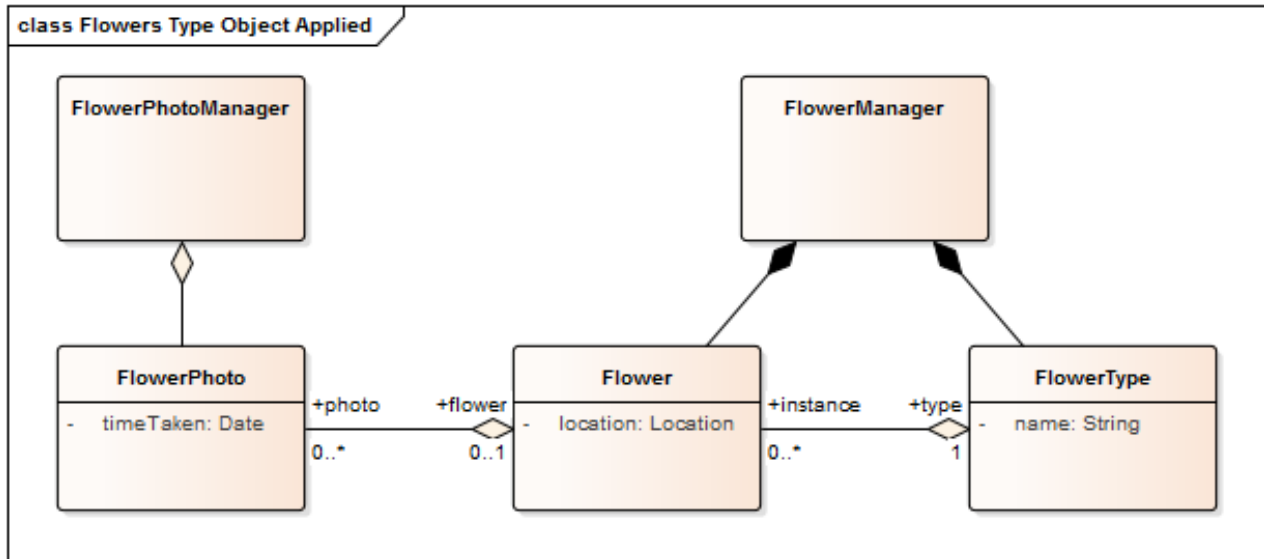
Select all correct statements.

- 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.

## 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) at runtime



## Downsides of Type Object Pattern

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

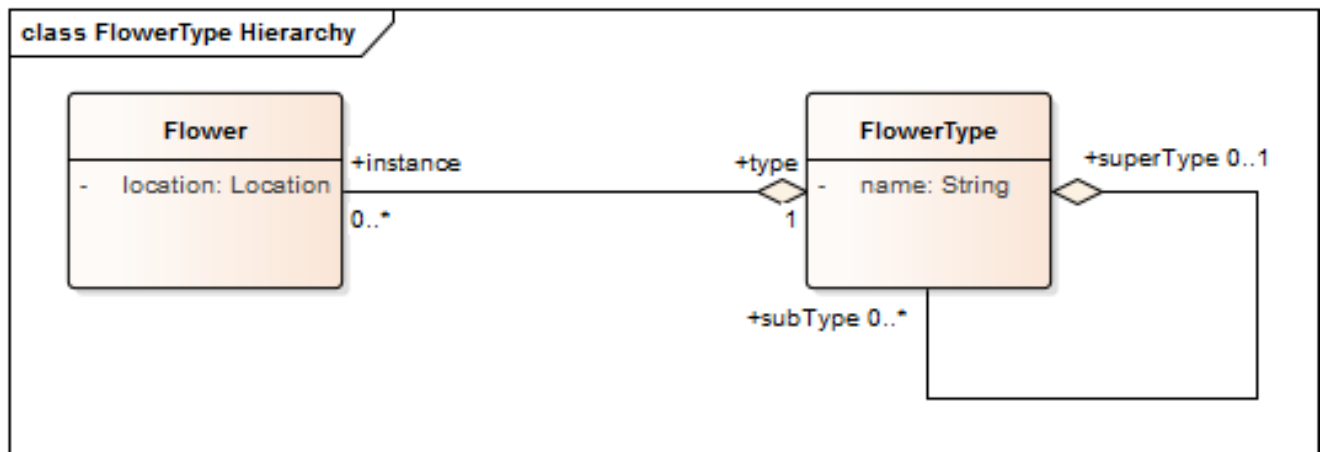
## Quiz: Type Object Hierarchy



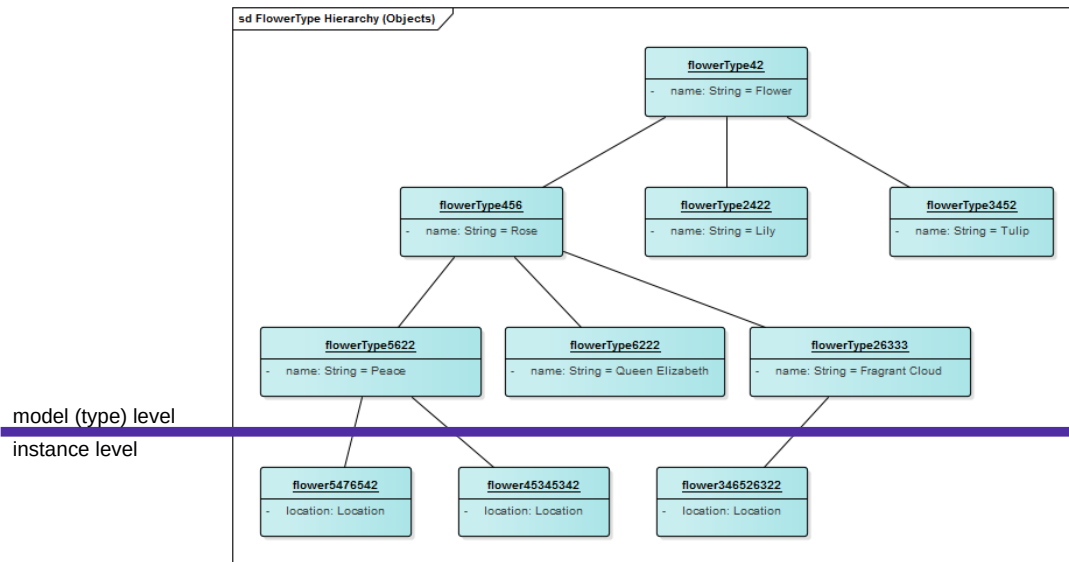
Plants are classified (in a hierarchy) according to the *International Code of Nomenclature for Cultivated Plants*.

**How to represent a type hierarchy for flowers in Flow-ers?**

## 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**

- done(): Object
- equals(Object): boolean
- finalize(): void
- getClass(): Class<T>
- hashCode(): int
- notify(): void
- notifyAll(): void
- registerNatives(): void
- toString(): String
- wait(long): void
- wait(long, int): void
- wait(): void

**java.lang.reflect.GenericDeclaration**  
**java.lang.reflect.Type**  
**java.lang.reflect.AnnotatedElement**

**Class** **[leaf]**

- asSubclass(Class<U>): Class<T> extends U>
- cast(Object): T
- desiredAssertionStatus(): boolean
- forName(String): Class<T>
- forName(String, boolean, ClassLoader): Class<T>
- getAnnotation(Class<A>): A
- getAnnotations(): Annotation[]
- getCanonicalName(): String
- getClass(): Class<T>[]
- getClassLoader(): ClassLoader
- getComponentType(): Class<T>
- getConstructor(Class<T>): Constructor<T>
- getConstructors(): Constructor<T>[]
- getDeclaredAnnotation(): Annotation
- getDeclaredAnnotations(): Annotation[]
- getDeclaredClasses(): Class<T>[]
- getDeclaredConstructor(Class<T>): Constructor<T>
- getDeclaredConstructors(): Constructor<T>[]
- getDeclaredField(String): Field
- getDeclaredFields(): Field[]
- getDeclaredMethod(String, Class<T>): Method
- getDeclaredMethods(): Method[]
- getDeclaringClass(): Class<T>
- getEnclosingClass(): Class<T>
- getEnclosingConstructor(): Constructor<T>
- getEnclosingMethod(): Method
- getEnumConstants(): T[]
- getField(String): Field
- getFields(): Field[]
- getGenericInterfaces(): Type[]
- getGenericSuperclass(): Type
- getInterfaces(): Class<T>[]
- getMethod(String, Class<T>): 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<T> super T>
- getTypeParameters(): TypeVariable<Class<T>>[]
- isAnnotation(): boolean
- isAnnotationPresent(Class<? extends Annotation>): boolean
- isAnonymousClass(): boolean
- isArray(): boolean
- isAssignableFrom(Class<T>): boolean
- isEnum(): boolean
- isInstantiable(): boolean
- isInterface(): boolean
- isLocalClass(): boolean
- isMemberClass(): boolean
- isPrimitive(): boolean
- isSynthetic(): boolean
- newInstance(): T
- toString(): String

**static**  
**Class: MethodArray**

- length: int
- methods: Method []
- addMethod(): void
- addAll(Method[]): void
- addAll(MethodArray): void
- addAllNotPresent(MethodArray): void
- addNotPresent(Method): void
- compactAndTrim(): void
- get(int): Method
- getArray(): Method[]
- length(): int
- MethodArray()
- removeByIndexAndSignature(Method): void

**static**  
**Class: EnclosingMethodInfo** **[leaf]**

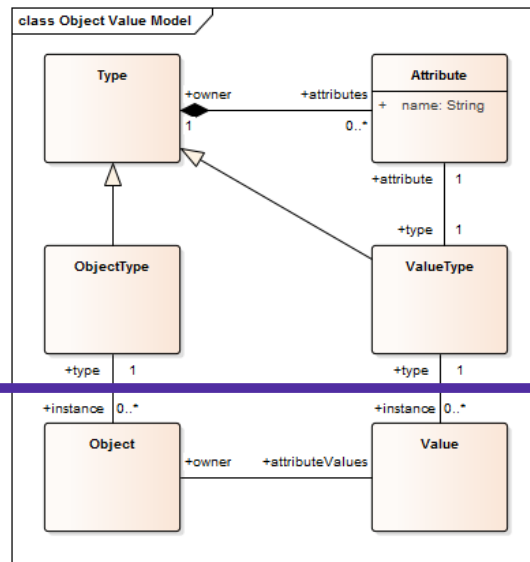
- descriptor: String
- enclosingClass: Class<T>
- name: String
- EnclosingMethodInfo(Object):
- getDescriptor(): String
- getEnclosingClass(): Class<T>
- getName(): String
- isConstructor(): boolean
- isMethod(): boolean
- isPartial(): boolean

←EnclosingClass

<T>

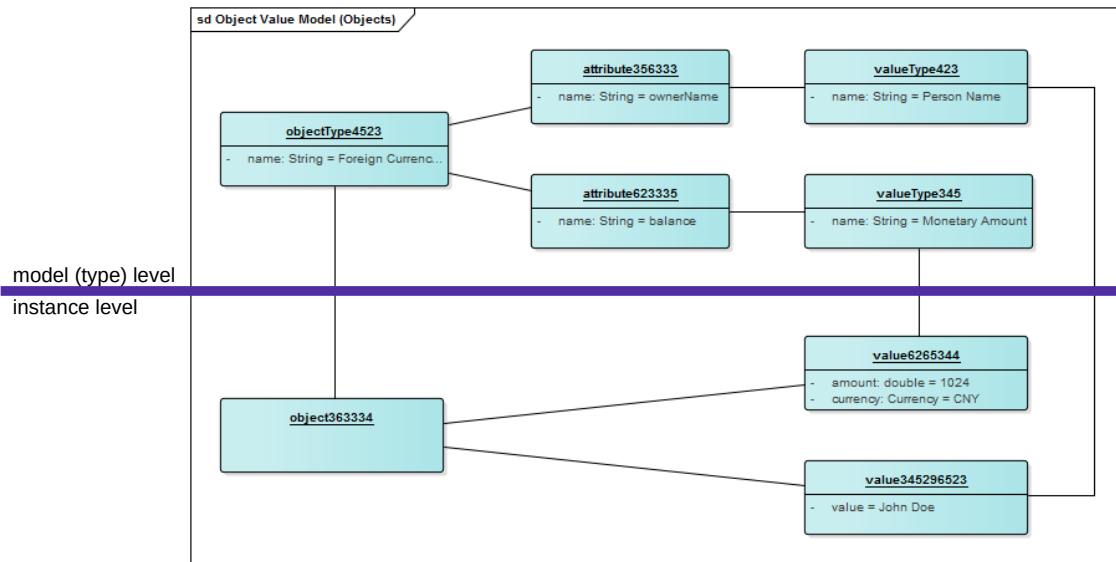
## Simple Object Value Model

model (type) level  
instance level

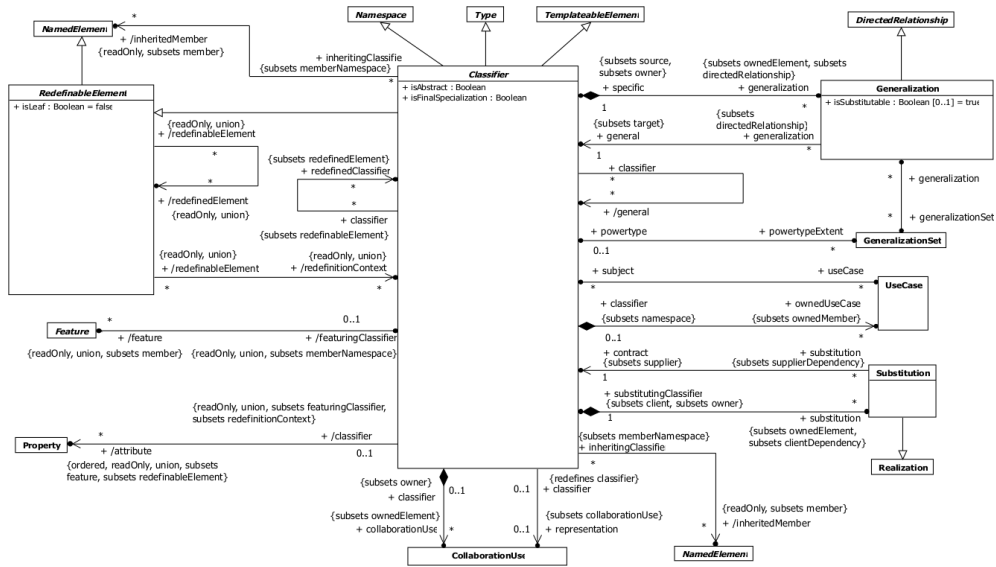




## Example of Simple Object Value Model



# 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. uml.Generalization uml.Stereotype uml.Component ...	flowers.Flower ...	...
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 flower345 : FlowerType ...
uml.Classifier uml.Generalization uml.Stereotype uml.Component ...	flowers.FlowerPhoto flowers.FlowerType flowers.Flower ...	fragrantCloud : FlowerType flower345 : FlowerType ...
flowers.FlowerType ...	flowers.Flower : FlowerType fragrantCloud : FlowerType ...	flower345 : FlowerType ...

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, similar 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



# Thank you! Questions?

**[dirk.riehle@fau.de](mailto:dirk.riehle@fau.de) – <http://osr.cs.fau.de>**

**[dirk@riehle.org](mailto:dirk@riehle.org) – <http://dirkriehle.com> – [@dirkriehle](#)**

DR

## Credits and License

- Original version
  - © 2012-2019 [Dirk Riehle](#), some rights reserved
  - Licensed under [Creative Commons Attribution 4.0 International License](#)
- Contributions
  - ...