

Subtyping and Inheritance

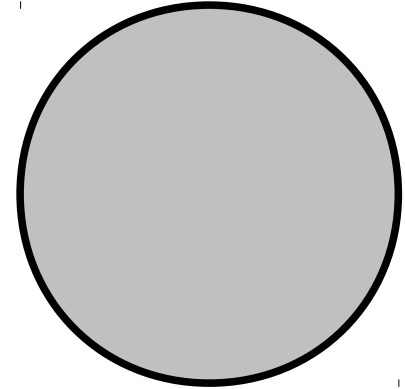
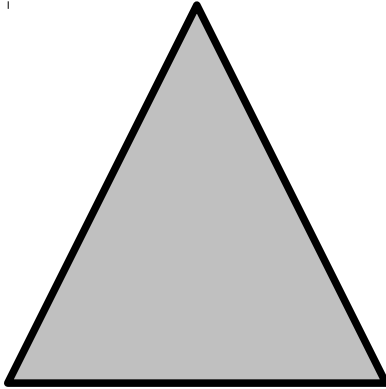
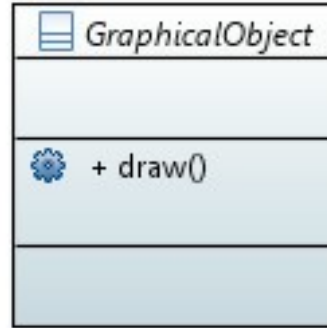
Prof. Dr. Dirk Riehle

Friedrich-Alexander University Erlangen-Nürnberg

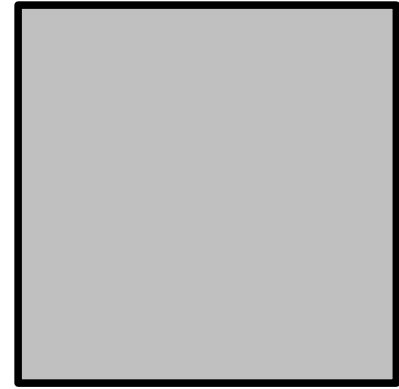
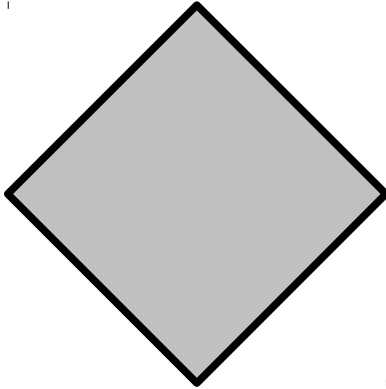
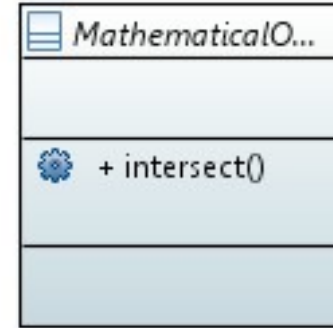
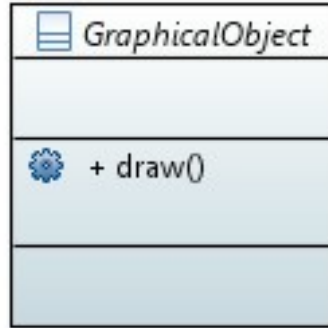
ADAP C04

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

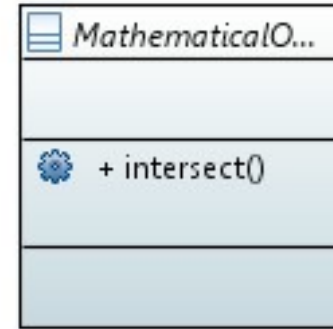
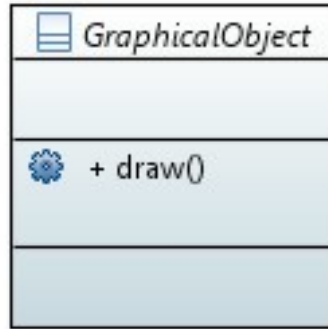
Subtyping Examples 1 / 3



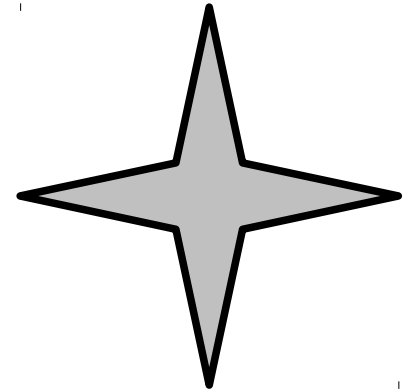
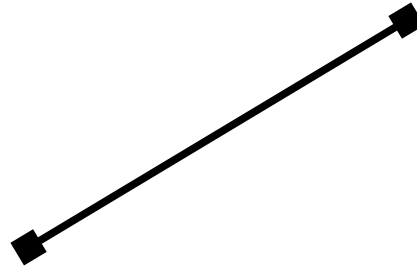
Subtyping Examples 2 / 3



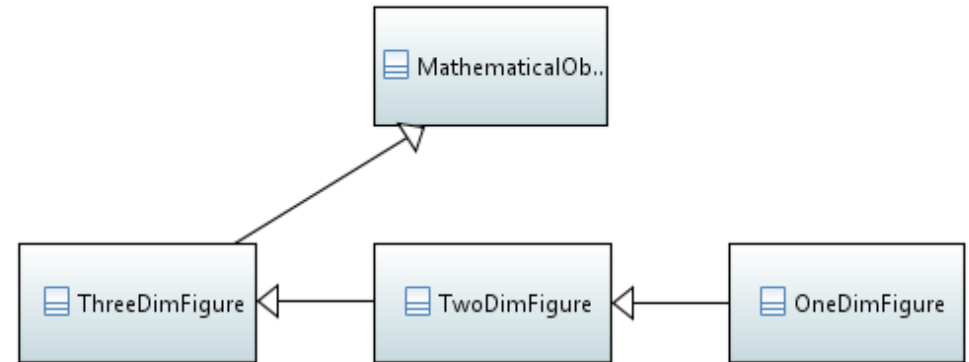
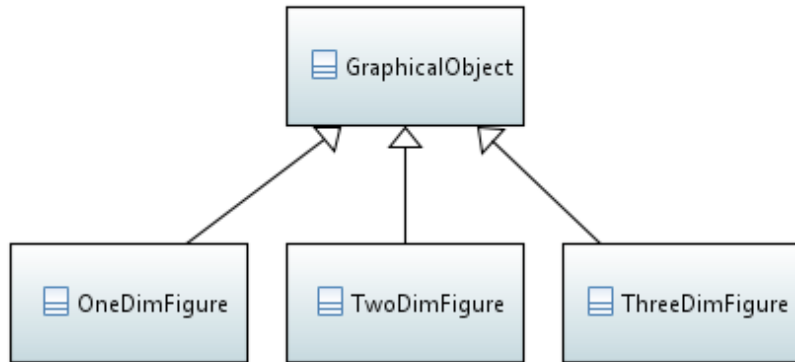
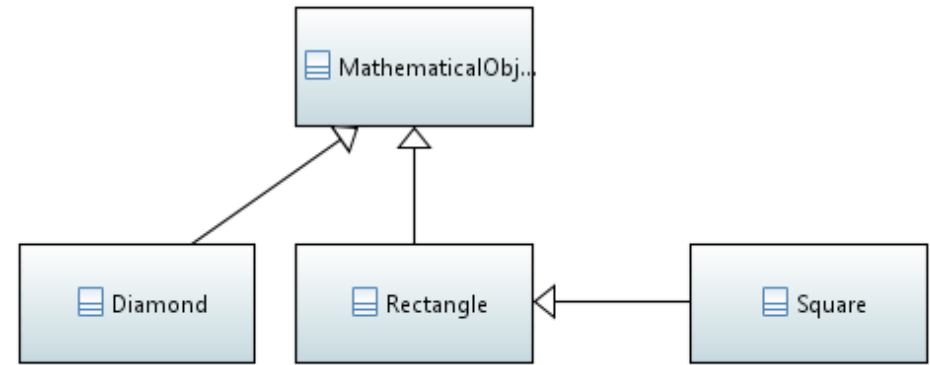
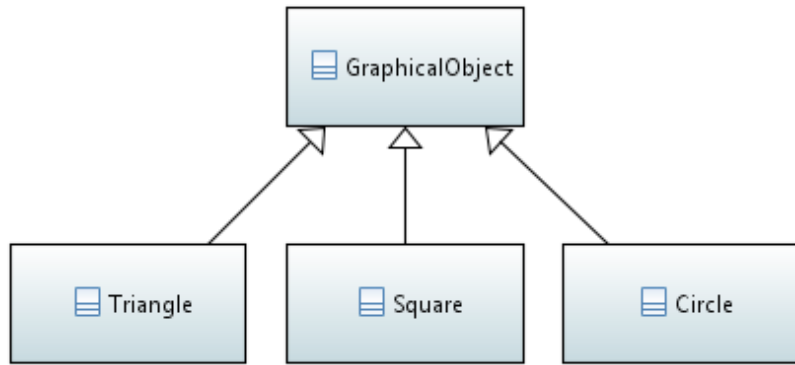
Subtyping Examples 3 / 3



×



Subtyping Examples Discussion Continued

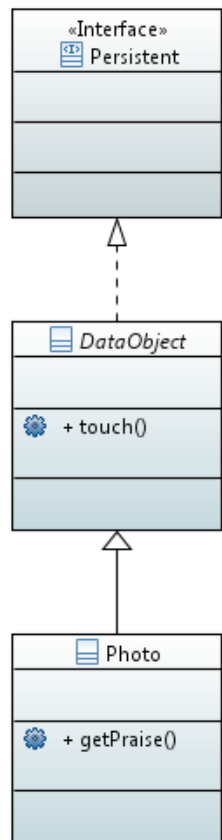


Let $\phi(x)$ be a property provable about objects x of type T . Then $\phi(y)$ should be provable for objects y of type S , where S is a subtype of T .

All properties that hold for instances of a supertype should also hold for instances of a subtype.

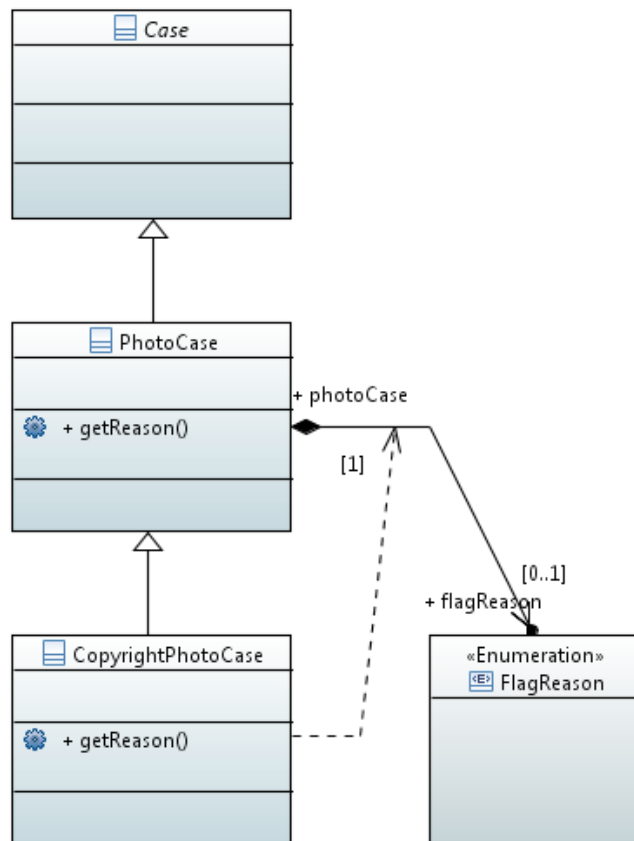
**No surprises
for a use-client!**

Subclasses as Extended Subtypes



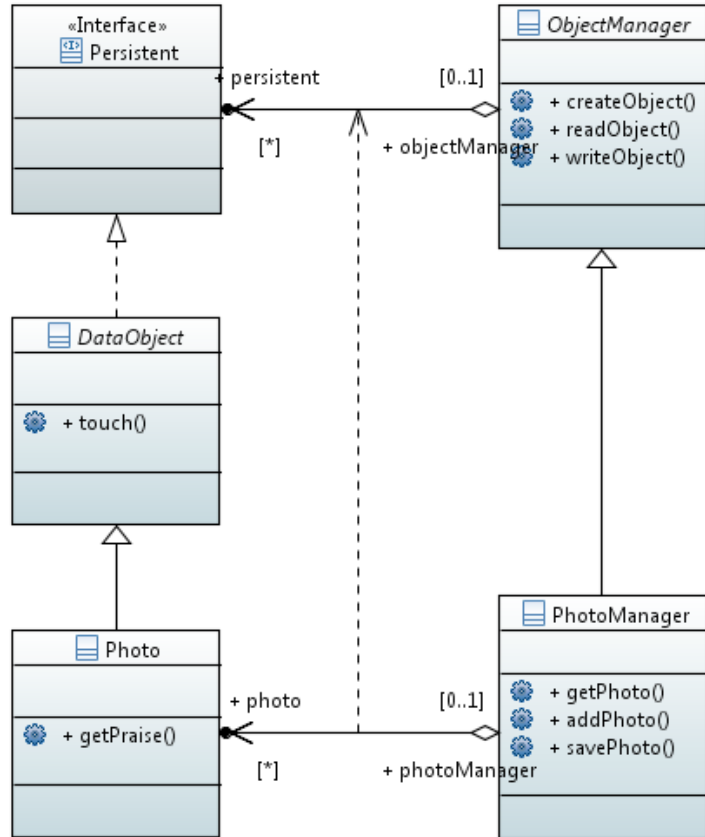
- Subclass
 - Adds methods and state
 - Does not constrain superclass
- Example
 - `public void DataObject#touch()`
 - `public float Photo#getPraise()`

Subclasses as Constrained Subtypes



- Constrained subtypes
 - Superclass defines possibility space
 - Subclass constrains behavior or return results
- Example
 - `FlagReason PhotoCase#getReason()`
 - Returns any of enum `FlagReason`
 - `FlagReason CopyrightPhotoCase#getReason()`
 - Returns only `FlagReason.COPYRIGHT`

Dual Class Hierarchies



- Constrained subtypes
 - ObjectManager
 - PhotoManager
- Extended subtypes
 - DataObject
 - Photo
- Association refinement

Covariance and Contravariance

- **Covariant redefinition**

- A method has been **covariantly** redefined in its result or argument types if those result or argument types are of a subclass of the original result or argument types

- **Contravariant redefinition**

- A method of a has been **contravariantly** redefined in its result or argument types if those result or argument types are of a superclass of the original result or argument types

Quiz: Co- and Contravariance

1. Which form of redefinition of result types violates the Liskov Substituability Principle, if any?
 - a. Contravariant redefinition
 - b. Covariant redefinition
 - c. None
 - d. Both

2. Which form of redefinition of method argument types violates the Liskov Substituability Principle, if any?
 - a. Contravariant redefinition
 - b. Covariant redefinition
 - c. None
 - d. Both

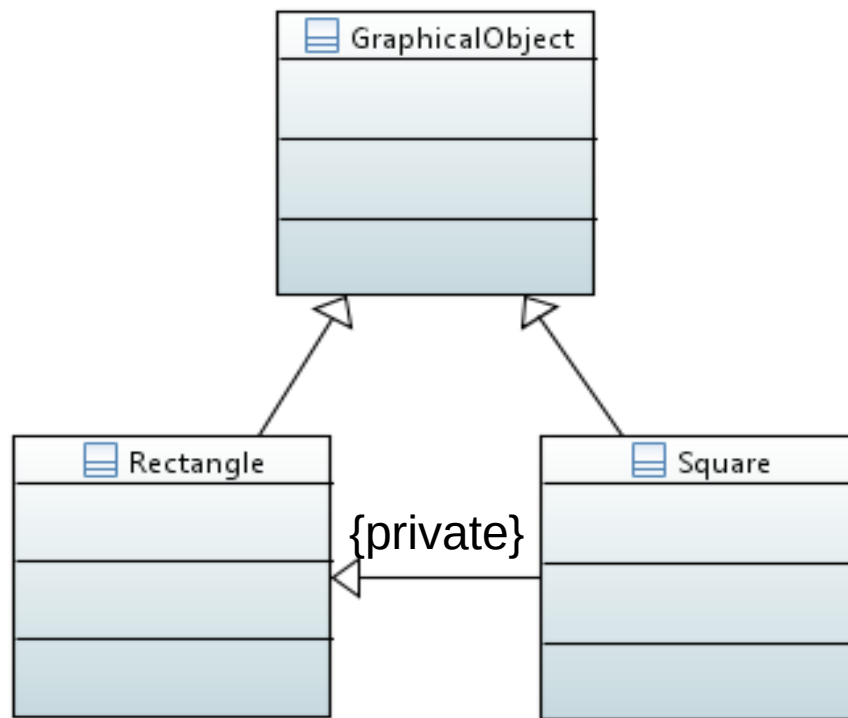
Covariance of Method Result Types

- General Java examples
 - `Object Object#clone()`
 - `MyClass MyClass#clone()`
- Wahlzeit examples
 - `Persistent ObjectManager#createObject(...)`
 - `Photo PhotoManager#createObject(...)`
- Covariant redefinition of method result types satisfies the LSP
 - Case of the constrained subtypes

Contravariance of Method Argument Types

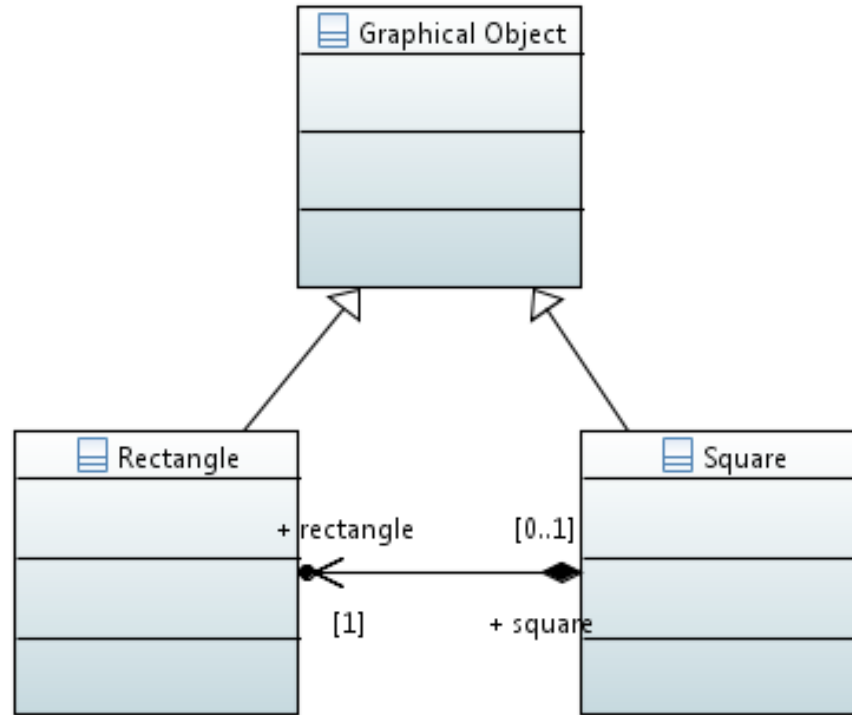
- **Not a language feature in Java (but in other languages)**
 - Hence no examples at hand
- Contravariant redefinition of method arguments satisfies the LSP
 - Case of the extended subtypes

Multiple Inheritance



Not possible in Java (not a language feature)

Implementation Delegation



Well possible in Java (and other languages)

Interface vs. Implementation Inheritance

- **Interface inheritance**

- Follows the LSP
- Can be realized in Java
 - Using Java interfaces
 - Using regular classes

- **Implementation inheritance**

- Breaks the LSP
- Can be realized in Java
 - But is generally a bad idea
 - Rather use delegation

Inheritance and Abstractness

- Inheritance
 - Relationship between two classes, a superclass and a subclass
- Abstract(ness)
 - Relationship between a class and its instances (none if abstract)

How to Make a Class Abstract (in Java)

- By declaration
 - of the class, e.g. “abstract class Counter { ... }”
 - of at least one method, e.g. “public abstract void count(...)”
- By hiding constructors
 - by declaring them protected or private
 - by making sure no implicit public empty constructor exists
- By inheritance
 - by inheriting from an abstract class and
 - not completing it
- **The best way is to explicitly declare one's intention**

**All superclasses must be abstract
(in design).**

A superclass should be abstract in implementation.

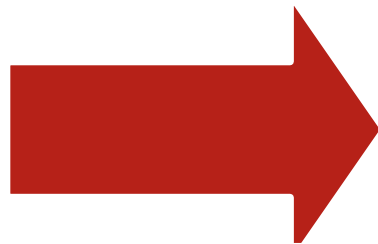
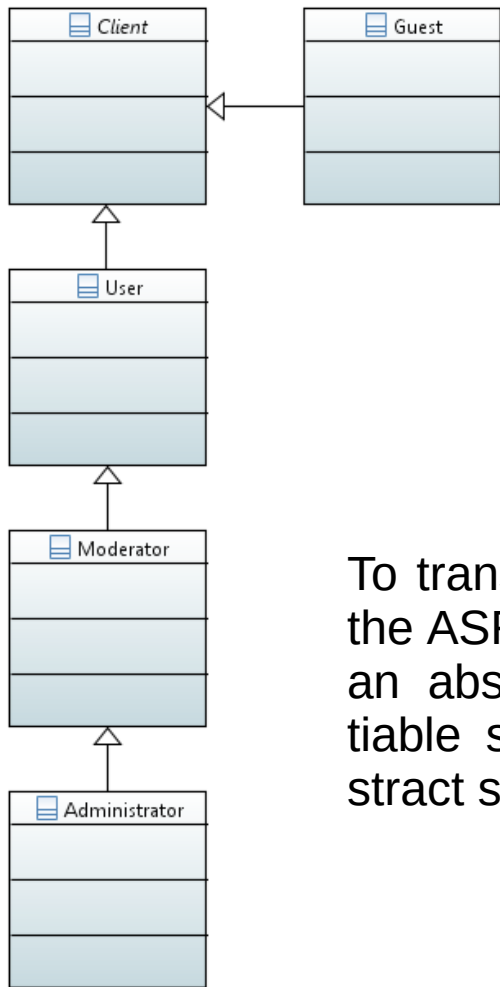
Corollaries to Abstract Superclass Rule

- Hard corollaries (rules)
 - No abstract class should subclass a concrete class
 - All subclasses should first be abstract, then concrete
- Soft corollaries (guidelines)
 - The root of a class hierarchy should be an abstract class
 - Leaf classes in an application should be concrete
- In a framework, leaf classes may be abstract
 - Because they are expecting subclasses in applications

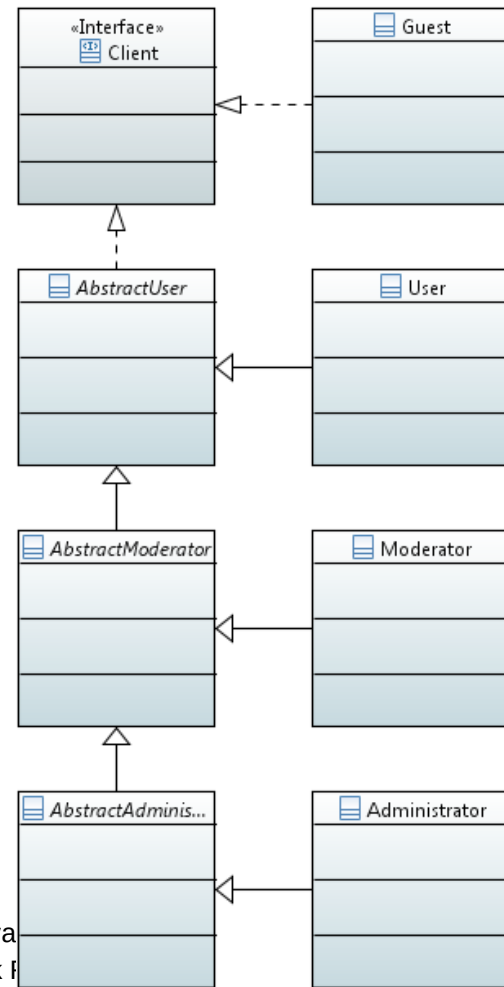
But Why? LSP applied to ASR

- The ASR helps comply with the LSP
 - The ASR automatically casts subclasses as constrained subtypes
 - Applying the ASR, developers have to think about subclasses
 - Subclasses fill in the holes defined by the abstract superclass
 - Thus, concrete subclasses constrain the abstract superclass
 - With this, the abstract superclass becomes better (re)usable

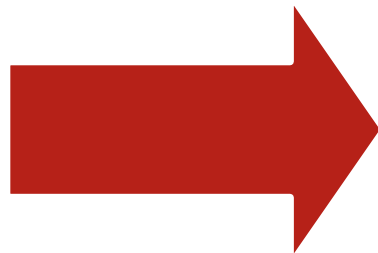
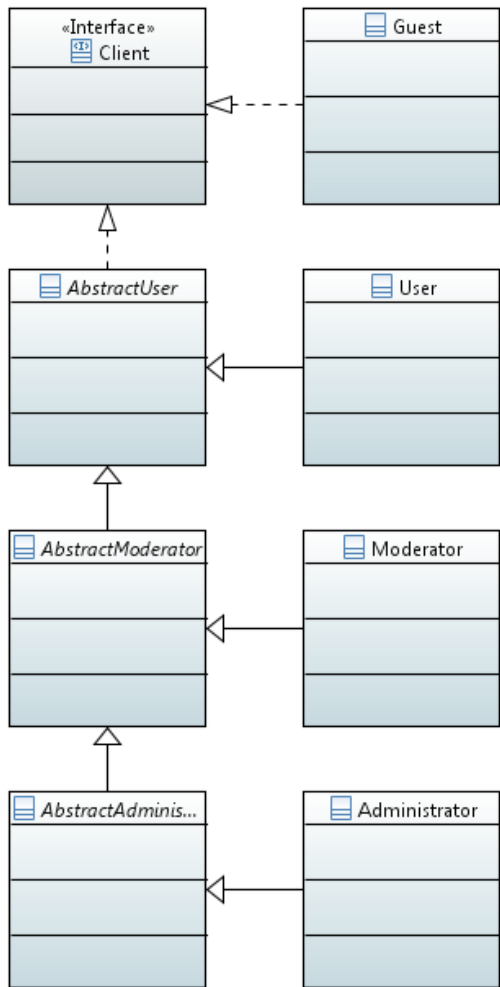
Transforming a Class Hierarchy



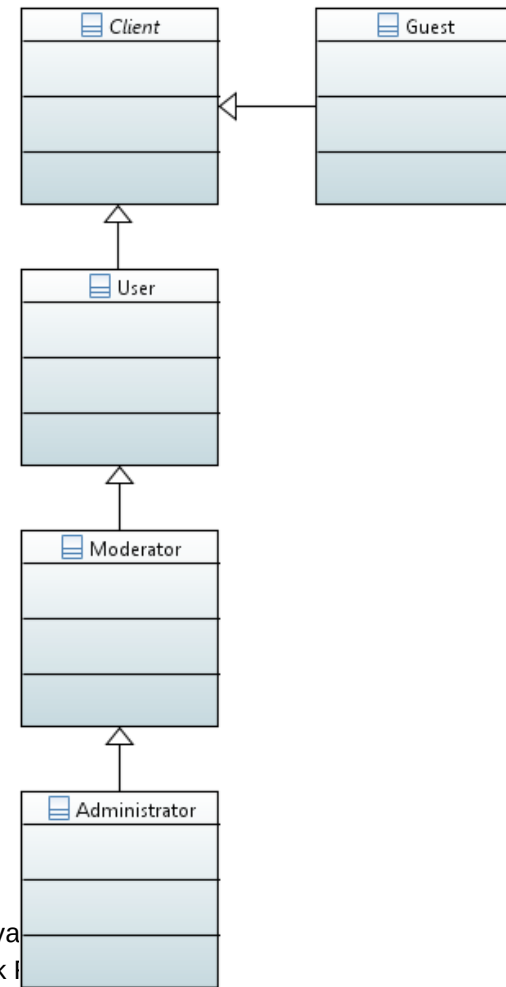
To transform a class hierarchy to conform to the ASR, split the instantiable superclass into an abstract class and introduce an instantiable subclass that completes the new abstract superclass.



Simplifying a Class Hierarchy



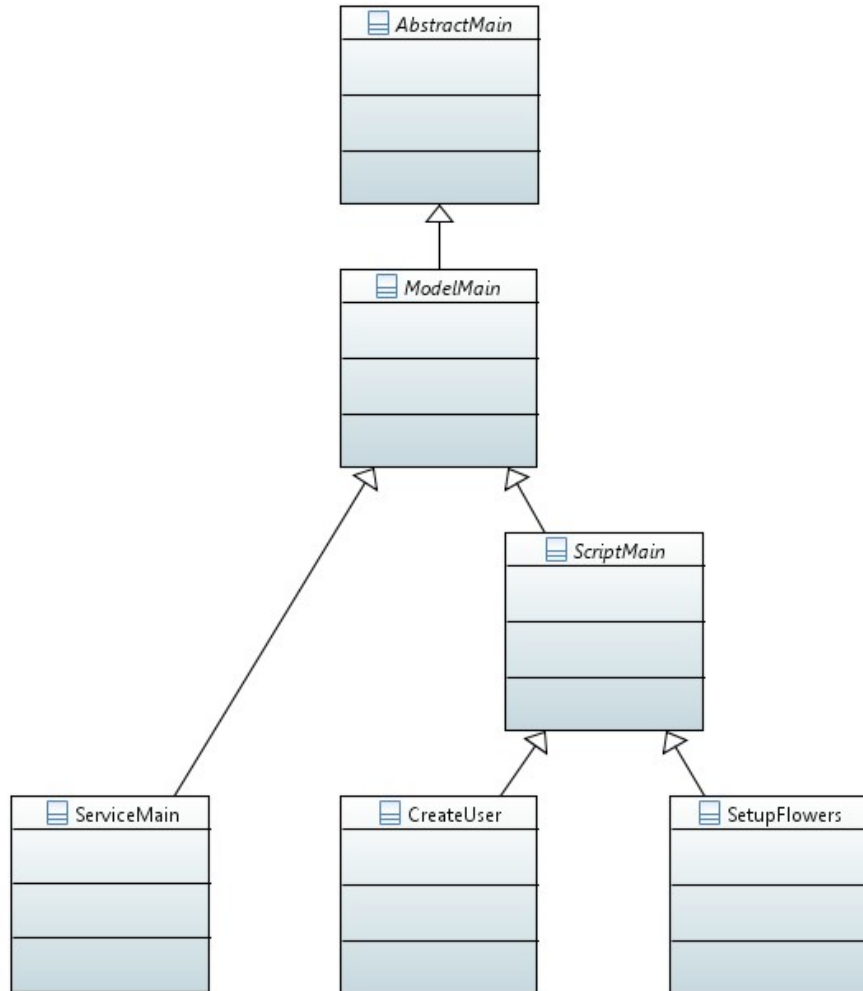
To simplify a class hierarchy, merge a default implementation with its abstract super-class.



What to Use When?

- Transform to conform in preparation for
 - increasingly complex default implementation class
 - other implementation classes as alternatives
- Simplify (and not conform) to
 - reduce number of overall classes
 - assuming implementation class is basically empty

Cascading Inheritance Interfaces



- **ServiceMain#startUp()** →
 - **ModelMain#startUp()** →
 - **AbstractMain#startUp()**
- **ServiceMain#shutDown()** →
 - **ModelMain#shutDown()** →
 - **AbstractMain#shutDown()**
- **CreateUser#execute()** →
- **ScriptMain#execute()**

Cascading Superclass Calls

```
public void ServiceMain#startUp(boolean ip, String rd) ... {  
    super.startUp(ip, rd);  
    log.info("ModelMain#startUp() completed");  
  
    log.config(LogBuilder.createSystemMessage()...);  
    initWebPartTemplateService();  
    ...  
}
```

```
protected void ModelMain#startUp(boolean ip, String rd) ... {  
    super.startUp(ip);  
    log.info("AbstractMain#startUp() completed");  
  
    log.config(LogBuilder.createSystemMessage()...);  
    initImageStore();  
    ...  
}
```

```
protected void AbstractMain#startUp(boolean ip) throws Exception {  
    isInProduction = ip;  
}
```

Traditional Run vs. ServletContext

```
public static main(String[] argv) { new FlowersMain.run() }

void FlowersMain#run() {
    ...
    startUp();
    execute();
    shutDown();
    ...
}
```

```
public void contextInitialized(ServletContextEvent sce) {
    ...
    serviceMain.startUp(true, rootDir);
    ...
}
public void contextDestroyed(ServletContextEvent sce) {
    ...
    serviceMain.shutDown();
    ...
}
```

Review / Summary of Session

- Typing and subtyping
 - Liskov Substitutability Principle (LSP)
- Class hierarchies
 - Abstract Superclass Rule (ASR)
 - Interface vs. implementation inheritance
 - Types of class hierarchies

Thank you! Questions?

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

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
 - ...