UML Object and Package Diagrams

Software Design (40007) – 2023/2024

Justus Bogner
Ivano Malavolta

Software and Sustainability research group (S2)
Department of Computer Science, Faculty of Sciences

A Quick Intro to Object Diagrams

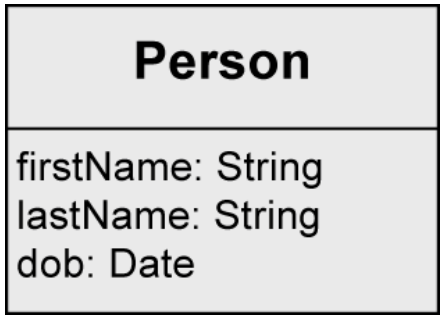
Not mandatory for the assignment! Can give bonus points.

You can find more info in chapter 4 of “UML@Classroom: An Introduction to Object-Oriented Modeling”

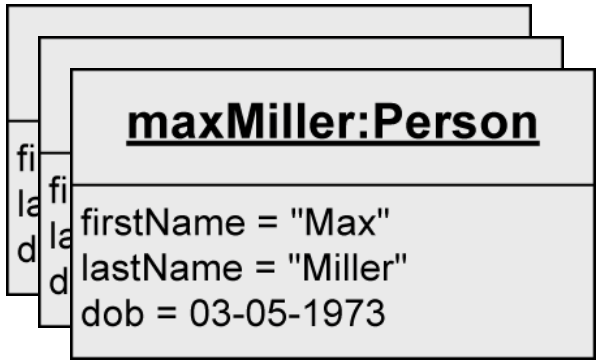
From classes to objects

- A class is a **construction plan** for a set of similar objects, i.e., objects are instances of classes
- Attributes: different value for each instance (= object)
- Operations: identical for all objects of a class
  → not depicted in object diagram

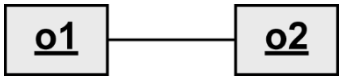
Class



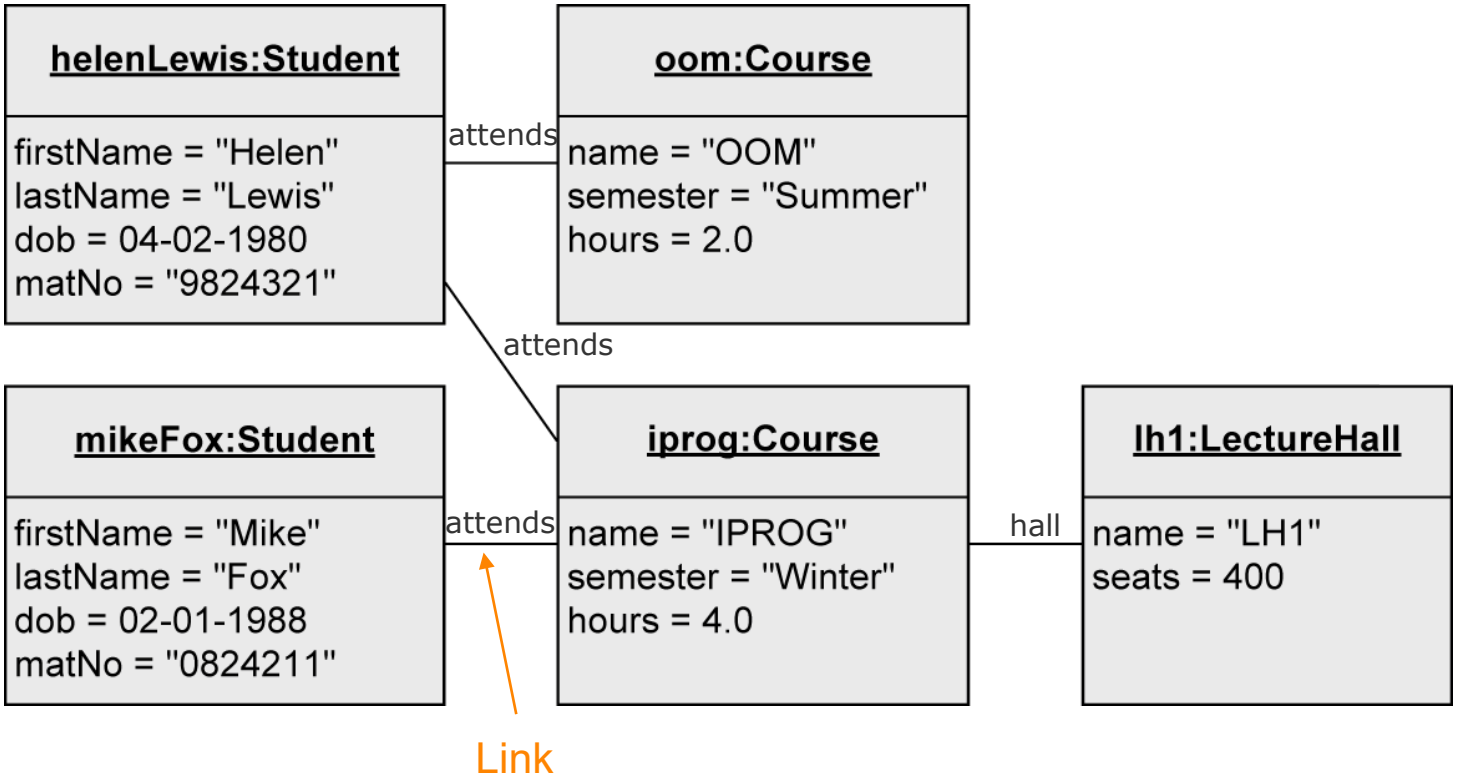
Objects



Object diagram

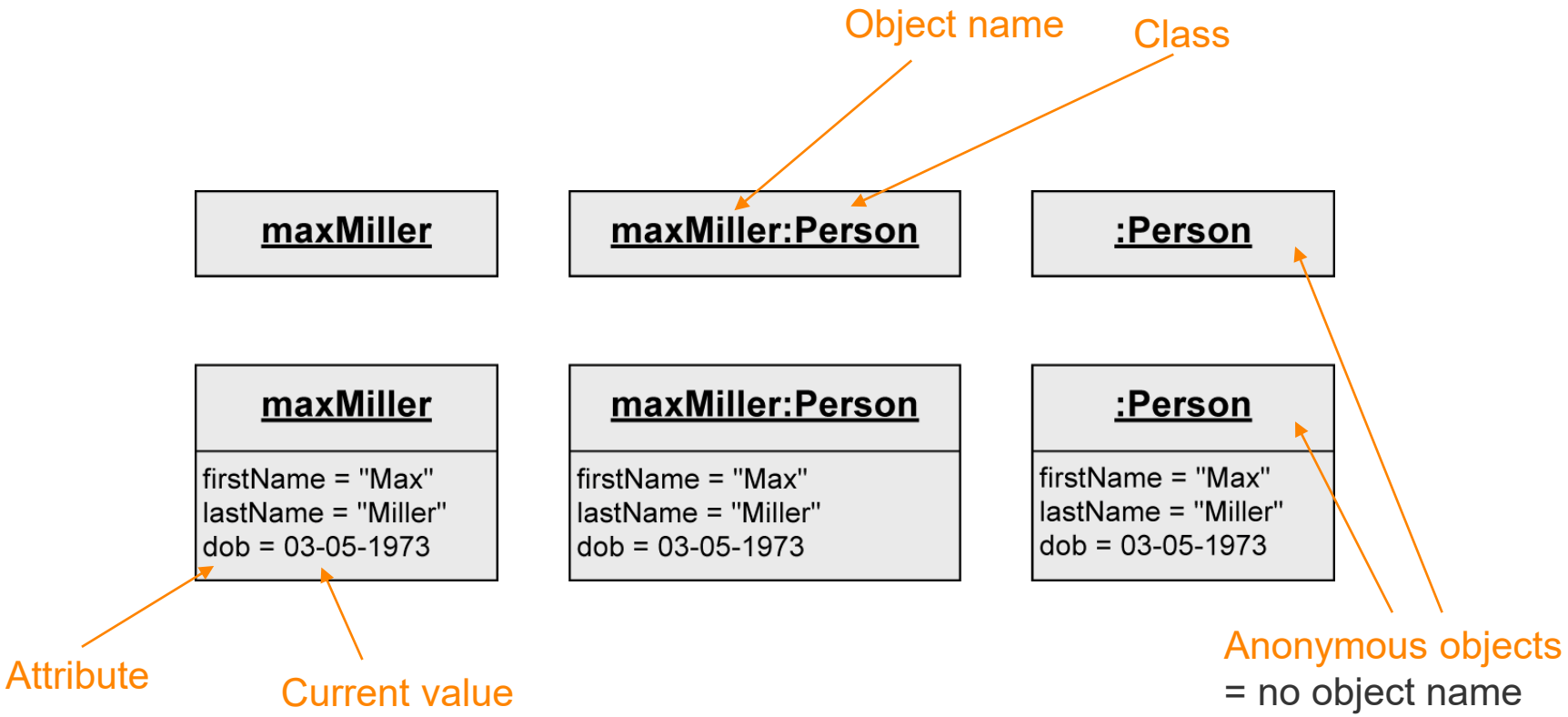


- Objects of a system and their relationships (links)
- **Snapshot of the system at a specific moment in time**



Alternative notations

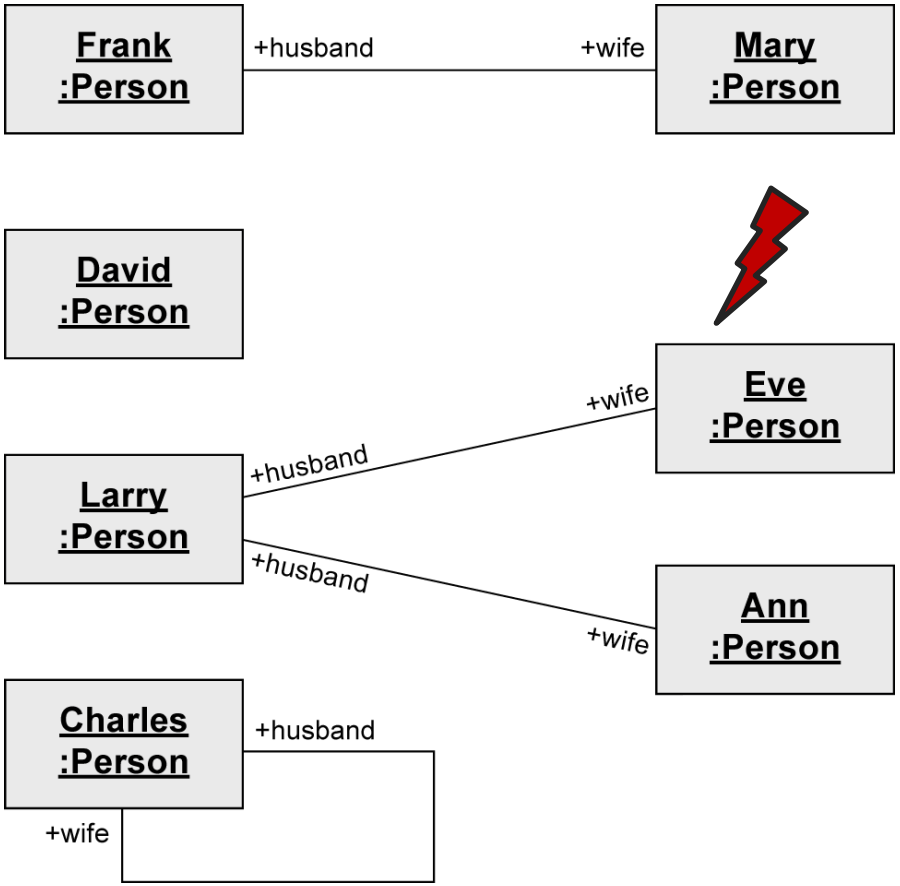
o:C



Using object diagrams for debugging

Class diagram

Object diagram

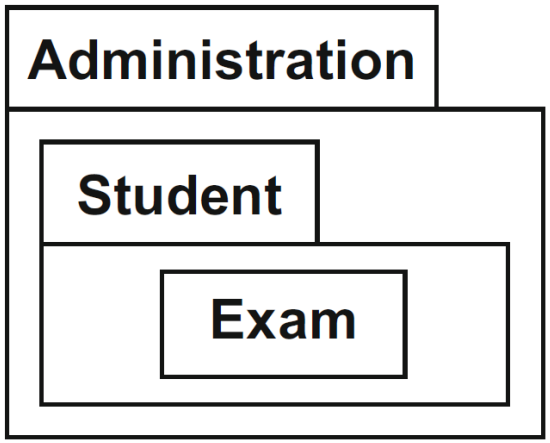


Package Diagrams

Mandatory for the assignment!

Why package diagrams?

- How to organize your classes?
- Many programming languages have *namespaces*, i.e., containers for higher-level structuring
- Java has *packages*: correspond to file system directories
- UML provides **package diagrams:** display packages that group model elements, e.g., classes, states, other packages, etc.
- Can be used for grouping various models or model elements

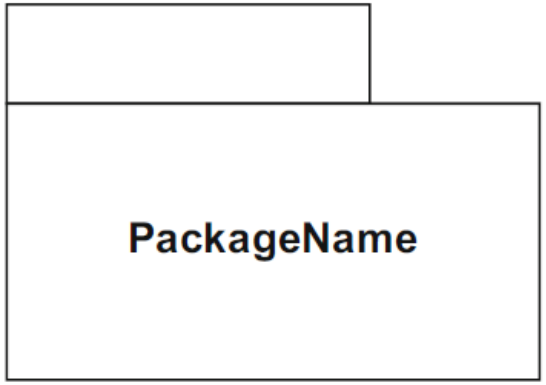


How we use package diagrams

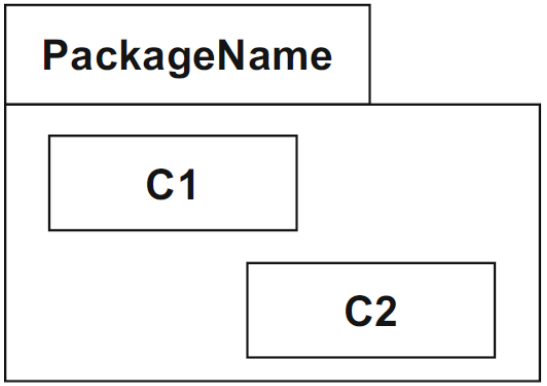
- For modeling the package structure of our Java programs, similar to *modules* → units of implementation
- *Descriptive* usage: design and document the higher-level structure for human consumers, e.g., software engineers
- Serves as a first entry point to understand the major building blocks of the project → quick overview
- Short textual description per package
- Including all classes might decrease readability
- Depending on # of classes: only include the most important ones or no classes at all

Package diagram notations

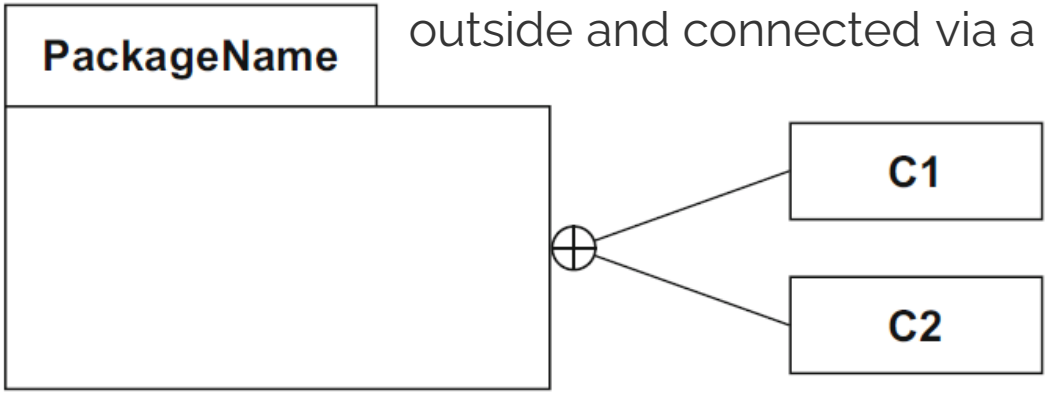
1. Displaying packages without their content:



2. With content, `name` moves to the smaller rectangle on top:



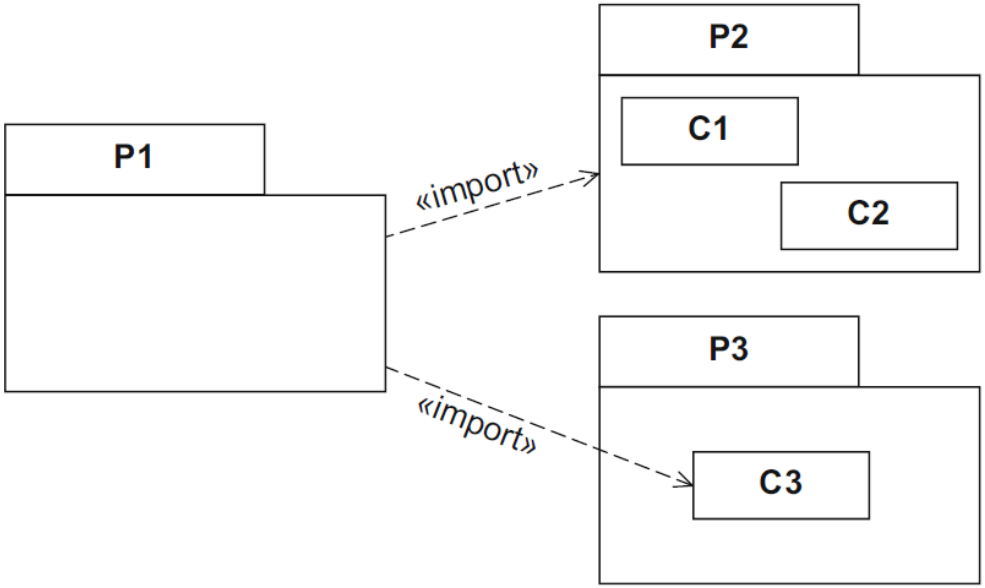
3. With a lot of content, elements can be moved outside and connected via a circle with cross:



Question: in how many packages can a class be?

Importing packages and classes

- Use `import` connectors to indicate that functionality is used in other packages



- `import` is possible for packages or individual classes
- Other connectors: `access`, `merge`, `use`
- Suggestion: focus on `import`, ignore the rest

How to name packages?

- We use package diagrams to model Java packages
- Follow common Java naming conventions!
- Only **lowercase letters and digits**
- No spaces, no special characters like hyphens
- Having no word separators encourages short names
- Oracle guide: https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html
- Java style guide from Google: https://google.github.io/styleguide/javaguide.html#s5.2.1-package-names

How to decide which packages to create?

- Two main goals for packaging:
  - Improve reasoning and navigation through the project
  - Isolate change
- Two general approaches for packaging:
  - Package by layer
  - Package by feature

Package by layer

- Organize packages according to the types of classes and the technical responsibility they have, e.g., UI, business logic, data persistence, etc.
- Advantage: easy to understand, easy to navigate for small projects
- Disadvantage: not good at isolating change
- Examples of other packages:
  - `validation`
  - `parsing`
  - `controllers`
  - `businesslogic`

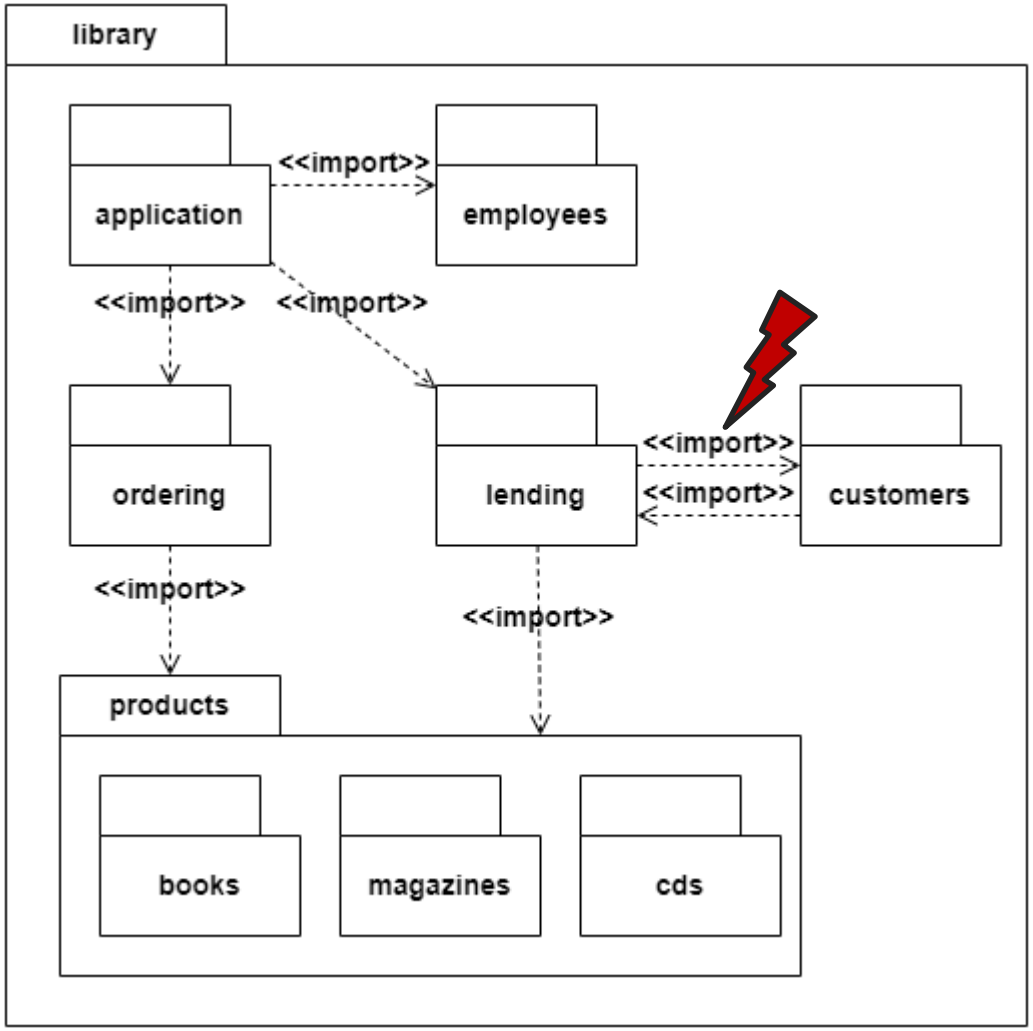
```
nl
 +- vu
     +- bookstore
         +- Application.java
         |
         +- domainentities
         |   +- Customer.java
         |   +- Order.java
         |   +- Product.java
         |
         +- persistence
         |   +- CustomerRepository.java
         |   +- OrderRepository.java
         |   +- ProductRepository.java
         |
         +- ui
            +- CustomerWidget.java
            +- OrderWidget.java
            +- ProductWidget.java
```

Package by feature

- Organize packages according to their domain-related functionality → *functional cohesion*
- Advantage: very good at isolating change
- Disadvantage: only easy to navigate if you know the domain well
- **Strongly advised for larger projects!**

```
nl
 +- vu
     +- bookstore
         +- Application.java
         |
         +- customers
         |   +- Customer.java
         |   +- CustomerRepository.java
         |   +- CustomerWidget.java
         |
         +- orders
         |   +- Order.java
         |   +- OrderRepository.java
         |   +- OrderWidget.java
         |
         +- products
            +- Product.java
            +- ProductRepository.java
            +- ProductWidget.java
```

Full example: library management system



Is this *package by layer* or *package by feature*?

Important: try to avoid cyclic dependencies!

Conclusion

Key takeaways

- **UML** = general purpose modeling language, tailored to object-oriented software
- 1 UML model, many diagrams
- **Class diagrams** for describing main entities, data structures, and operations
- They can also be used **as descriptive models**
  - for understanding the main “blocks“ of the system
  - no direct mapping to the code in this case, only reasoning
- **Object diagrams** show instances of classes, useful for describing a “snapshot” of the system at run-time
- **Package diagrams** show the high-level module structure and dependencies between packages

Readings

- UML@Classroom: An Introduction to Object-Oriented Modeling" – chapters 2 and 4
- A philosophy of Software Design, Chapters 4, 5, 6
- [optional] Concise Guide to Object-Oriented Programming, Chapter 5
- [optional] Object-Oriented Analysis, Design, and Implementation, Chapters 2 and 3